РЕФАКТОРИНГ
Улучшение существующего кода

МАРТИН ФАУЛЕР

При участии Кента Бека, Джона Бранта, Уильяма Апдайка и Дона Робертса
предисловие Эриха Гаммы Object Technology International, Inc.

Мартин Фаулер

Рефакторинг
Улучшение существующего кода

Перевод С. Маккавеева

Главный редактор	А. Галунов
Зав. редакцией	Н.Макарова
Научный редактор	Е. Шпика
Редактор	В. Овчинников
Корректура	С. Беляева
Верстка	А.Дорошенко

Фаулер М.

Рефакторинг: улучшение существующего кода. - Пер. с англ. - СПб: Символ-Плюс, 2003. - 432 с, ил.

ISBN 5-93286-045-6

Подход к улучшению структурной целостности и производительности существующих программ, называемый рефакторингом, получил развитие благодаря усилиям экспертов в области ООП, написавших эту книгу. Каждый шаг рефакторинга прост. Это может быть перемещение поля из одного класса в другой, вынесение фрагмента кода из метода и превращение его в самостоятельный метод или даже перемещение кода по иерархии классов. Каждый отдельный шаг может показаться элементарным, но совокупный эффект таких малых изменений в состоянии радикально улучшить проект или даже предотвратить распад плохо спроектированной программы.

Мартин Фаулер с соавторами пролили свет на процесс рефакторинга, описав принципы и лучшие приемы его осуществления, а также указав, где и когда следует начинать углубленное изучение кода с целью его улучшения. Основу книги составляет подробный перечень более 70 методов рефакторинга, для каждого из которых описываются мотивация и техника испытанного на практике преобразования кода с примерами на Java. Рассмотренные в книге методы позволяют поэтапно модифицировать код, внося каждый раз небольшие изменения, благодаря чему снижается риск, связанный с развитием проекта.

ISBN 5-93286-045-6
ISBN 0-201-48567-2 (англ)

© Издательство Символ-Плюс, 2003

Original English language title: Refactoring: Improving the Design of Existing Code by Martin Fowler, Copyright © 2000, All Rights Reserved. Published by arrangement with the original publisher, Pearson Education, Inc., publishing as ADDISON WESLEY LONGMAN.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота - общероссийский классификатор продукции
ОК 005-93, том 2; 953000 - книги и брошюры.
Подписано в печать 30.09.2002. Формат 70xl00Vi6 . Печать офсетная.
Объем 27 печ. л. Тираж 3000 экз. Заказ N 3385 Отпечатано с диапозитивов в Академической типографии
«Наука» РАН 199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие....7
1 Рефакторинг, первый пример....11
Исходная программа....11
Первый шаг рефакторинга....15
Декомпозиция и перераспределение метода statement....15
Замена условной логики на полиморфизм....31
Заключительные размышления....41
2 Принципы рефакторинга....42
Определение рефакторинга....42
Зачем нужно проводить рефакторинг?....43
Когда следует проводить рефакторинг?....44
Как объяснить это своему руководителю?....46
Проблемы, возникающие при проведении рефакторинга....47
Рефакторинг и проектирование....49
Рефакторинг и производительность....51
Каковы истоки рефакторинга?....52
3 Код с душком....54
Дублирование кода....54
Длинный метод....54
Большой класс....55
Длинный список параметров....56
Расходящиеся модификации....56
«Стрельба дробью»....56
Завистливые функции....57
Группы данных....57
Одержимость элементарными типами....57
Операторы типа switch....58
Параллельные иерархии наследования....58
Ленивый класс....58
Теоретическая общность....59
Временное поле....59
Цепочки сообщений....59
Посредник....59
Неуместная близость....60
Альтернативные классы с разными интерфейсами....60
Неполнота библиотечного класса....60
Классы данных....60
Отказ от наследства....61
Комментарии....61
4 Разработка тестов....62
Ценность самотестирующегося кода....62
Среда тестирования JUnit....63
Добавление новых тестов....67
5 На пути к каталогу методов рефакторинга....72
Формат методов рефакторинга....72
Поиск ссылок....73
Насколько зрелыми являются предлагаемые методы рефакторинга?....73
6 Составление методов....75
Выделение метода (Extract Method)....75
Встраивание метода (Inline Method)....80
Встраивание временной переменной (Inline Temp)....81
Замена временной переменной вызовом метода (Replace Temp with Query)....81
Введение поясняющей переменной (Introduce Explaining Variable)....84
Расщепление временной переменной (Split Temporary Variable)....87
Удаление присваиваний параметрам (Remove Assignments to Parameters)....89
Замена метода объектом методов (Replace Method with Method Object)....91
Замещение алгоритма (Substitute Algorithm)....94
7 Перемещение функций между объектами....96
Перемещение метода (Move Method)....96
Перемещение поля (Move Field)....99
Выделение класса (Extract Class)....101
Встраивание класса (Inline Class)....104
Сокрытие делегирования (Hide Delegate)....106
Удаление посредника (Remove Middle Man)....108

Введение внешнего метода (Introduce Foreign Method)....109
Введение локального расширения (Introduce Local Extension)....110
8 Организация данных....114
Самоинкапсуляция поля (Self Encapsulate Field)....114
Замена значения данных объектом (Replace Data Value with Object)....117
Замена значения ссылкой (Change Value to Reference)....119
Замена ссылки значением (Change Reference to Value)....122
Замена массива объектом (Replace Array with Object)....123
Дублирование видимых данных (Duplicate Observed Data)....126
Замена однонаправленной связи двунаправленной (Change Unidirectional Association to Bidirectional)....131
Замена двунаправленной связи однонаправленной (Change Bidirectional Association to Unidirectional)....134
Замена магического числа символической константой (Replace Magic Number with Symbolic Constant)....136
Инкапсуляция поля (Encapsulate Field)....137
Инкапсуляция коллекции (Encapsulate Collection)....138
Замена записи классом данных (Replace Record with Data Class)....144
Замена кода типа классом (Replace Type Code with Class)....144
Замена кода типа подклассами (Replace Type Code with Subclasses)....148
Замена кода типа состоянием/стратегией (Replace Type Code with State/Strategy)....150
Замена подкласса полями (Replace Subclass with Fields)....154
9 Упрощение условных выражений....157
Декомпозиция условного оператора (Decompose Conditional)....157
Консолидация условного выражения (Consolidate Conditional Expression)....158
Консолидация дублирующихся условных фрагментов (Consolidate Duplicate Conditional Fragments)....160
Удаление управляющего флага (Remove Control Flag)....161
Замена вложенных условных операторов граничным оператором (Replace Nested Conditional with Guard Clauses)....164
Замена условного оператора полиморфизмом (Replace Conditional with Polymorphism)....167
Введение объекта Null (Introduce Null Object)....171
Введение утверждения (Introduce Assertion)....175
10 Упрощение вызовов методов....179
Переименование метода (Rename Method)....179
Добавление параметра (Add Parameter)....181
Удаление параметра (Remove Parameter)....182
Разделение запроса и модификатора (Separate Query from Modifier)....182
Параметризация метода (Parameterize Method)....185
Замена параметра явными методами (Replace Parameter with Explicit Methods)....186
Сохранение всего объекта (Preserve Whole Object)....188
Замена параметра вызовом метода (Replace Parameter with Method)....191
Введение граничного объекта (Introduce Parameter Object)....193
Удаление метода установки значения (Remove Setting Method)....197
Сокрытие метода (Hide Method)....199
Замена конструктора фабричным методом (Replace Constructor with Factory Method)....199
Инкапсуляция нисходящего преобразования типа (Encapsulate Downcast)....202
Замена кода ошибки исключительной ситуацией (Replace Error Code with Exception)....203
Замена исключительной ситуации проверкой (Replace Exception with Test)....207
11 Решение задач обобщения....211
Подъем поля (Pull Up Field)....211
Подъем метода (Pull Up Method)....212
Подъем тела конструктора (Pull Up Constructor Body)....213
Спуск метода (Push Down Method)....215
Спуск поля (Push Down Field)....216
Выделение подкласса (Extract Subclass)....217
Выделение родительского класса (Extract Superclass)....221
Выделение интерфейса (Extract Interface)....224
Свертывание иерархии (Collapse Hierarchy)....226
Формирование шаблона метода (Form Template Method)....226
Замена наследования делегированием (Replace Inheritance with Delegation)....233
Замена делегирования наследованием (Replace Delegation with Inheritance)....235
12 Крупные рефакторинги....237
Разделение наследования (Tease Apart Inheritance)....238
Преобразование процедурного проекта в объекты (Convert Procedural Design to Objects)....241
Отделение предметной области от представления (Separate Domain from Presentation)....242
Выделение иерархии (Extract Hierarchy)....245
13 Рефакторинг, повторное использование и реальность....249
Проверка в реальных условиях....249

Почему разработчики не хотят применять рефакторинг к своим программам?....250
Возращаясь к проверке в реальных условиях....257
Ресурсы и ссылки, относящиеся к рефакторингу....258
Последствия повторного использования программного обеспечения и передачи технологий....258
Завершающее замечание....259
Библиография....259
14 Инструментальные средства проведения рефакторинга....261
Рефакторинг с использованием инструментальных средств....261
Технические критерии для инструментов проведения рефакторинга....262
Практические критерии для инструментов рефакторинга....263
Краткое заключение....264
15 Складывая все вместе....265
Библиография....267

ВСТУПИТЕЛЬНОЕ СЛОВО

Концепция «рефакторинга» (refactoring) возникла в кругах, связанных со Smalltalk, но вскоре нашла себе дорогу и в лагеря приверженцев других языков программирования. Поскольку рефакторинг является составной частью разработки структуры приложений (framework development), этот термин сразу появляется, когда «структурщики» начинают обсуждать свои дела. Он возникает, когда они уточняют свои иерархии классов и восторгаются тем, на сколько строк им удалось сократить код. Структурщики знают, что хорошую структуру удается создать не сразу - она должна развиваться по мере накопления опыта. Им также известно, что чаще приходится читать и модифицировать код, а не писать новый. В основе поддержки читаемости и модифицируемости кода лежит рефакторинг - как в частном случае структур (frameworks), так и для программного обеспечения в целом.

Так в чем проблема? Только в том, что с рефакторингом связан известный риск. Он требует внести изменения в работающий код, что может привести к появлению трудно находимых ошибок в программе. Неправильно осуществляя рефакторинг, можно потерять дни и даже недели. Еще большим риском чреват рефакторинг, осуществляемый без формальностей или эпизодически. Вы начинаете копаться в коде. Вскоре обнаруживаются новые возможности модификации, и вы начинаете копать глубже. Чем больше вы копаете, тем больше вскрывается нового и тем больше изменений вы производите. В конце концов, получится яма, из которой вы не сможете выбраться. Чтобы не рыть самому себе могилу, следует производить рефакторинг на систематической основе. В книге «Design Patterns»[1] мы с соавторами говорили о том, что проектные модели создают целевые объекты для рефакторинга. Однако указать цель - лишь одна часть задачи; преобразовать код так, чтобы достичь этой цели, - другая проблема.

Мартин Фаулер (Martm Fowler) и другие авторы, принявшие участие в написании этой книги, внесли большой вклад в разработку объектно-ориентированного программного обеспечения тем, что пролили свет на процесс рефакторинга. В книге описываются принципы и лучшие способы осуществления рефакторинга, а также указывается, где и когда следует начинать углубленно изучать код, чтобы улучшить его. Основу книги составляет подробный перечень методов рефакторинга. Каждый метод описывает мотивацию и технику испытанного на практике преобразования кода. Некоторые виды рефакторинга, такие как «Выделение метода» или «Перемещение поля», могут показаться очевидными, но пусть это не вводит вас в заблуждение. Понимание техники таких методов рефакторинга важно для организованного осуществления рефакторинга. С помощью описанных в этой книге методов рефакторинга можно поэтапно модифицировать код, внося каждый раз небольшие изменения, благодаря чему снижается риск, связанный с развитием проекта. Эти методы рефакторинга и их названия быстро займут место в вашем словаре разработчика.

Мой первый опыт проведения дисциплинированного «поэтапного» рефакторинга связан с программированием на пару с Кентом Беком (Kent Beck) на высоте 30 000 футов. Он обеспечил поэтапное применение методов рефакторинга, перечисленных в этой книге. Такая практика оказалась удивительно действенной. Мое доверие к полученному коду повысилось, а кроме того, я стал испытывать меньшее напряжение. Весьма рекомендую применить предлагаемые методы рефакторинга на практике: и вам и вашему коду это окажет большую пользу.

Erich Gamma Object Technology International, Inc.

[1] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д. «Приемы объектно-ориентированного проектирования. Паттерны проектирования», издательство «Питер», С-Пб, 2000 г

ПРЕДИСЛОВИЕ

Однажды некий консультант знакомился с проектом разработки программного обеспечения. Он посмотрел часть написанного кода; в центре системы была некоторая иерархия классов. Разбираясь с ней, консультант обнаружил, что она достаточно запутанна. Классы, лежащие в основе иерархии наследования, делали определенные допущения относительно того, как будут работать другие классы, и эти допущения были воплощены в наследующем коде. Однако этот код годился не для всех подклассов и довольно часто перегружался в подклассах. В родительский класс можно было бы ввести небольшие изменения и значительно сократить потребность в перегрузке кода. В других местах из-за того что назначение родительского класса не было в достаточной мере понято, дублировалось поведение, уже имеющееся в родительском классе. Еще кое-где несколько подклассов осуществляли одни и те же действия, код которых можно было бы беспрепятственно переместить вверх по иерархии классов.

Консультант порекомендовал руководителям проекта пересмотреть код и подчистить его, что не вызвало особого энтузиазма. Код вроде бы работал, а график проекта был напряженным. Руководство заявило, что займется этим как-нибудь позже.

Консультант также обратил внимание разрабатывавших иерархию программистов на обнаруженные им недостатки. Программисты смогли оценить проблему. По-настоящему они не были в ней виноваты - просто, как иногда бывает, необходим свежий взгляд. В итоге программисты потратили пару дней на доработку иерархии. По завершении этой работы они смогли удалить половину кода без ущерба для функциональности системы. Они были довольны результатом и нашли, что добавлять в иерархию новые классы и использовать имеющиеся классы в других местах системы стало легче.

Руководители проекта довольны не были. Время поджимало, а работы оставалось много. Труд, которым эти два программиста были заняты два дня, не привел к появлению в системе ни одной функции из тех многих, которые еще предстояло добавить за несколько месяцев, остающихся до поставки готового продукта. Прежний код отлично работал. Проект стал лишь несколько более «чистым» и «ясным». Итогом проекта должен быть работающий код, а не тот, который понравится университетскому ученому. Консультант предложил поправить и другие главные части системы. Такая работа могла задержать проект на одну-две недели. И все это лишь для того, чтобы код лучше выглядел, а не для того, чтобы он выполнял какие-то новые функции.

Как вы отнесетесь к этому рассказу? Был ли, по вашему мнению, прав консультант, предлагая продолжить подчистку кода? Или вы следуете старому совету инженеров - «если что-то работает, не трогай его»?

Признаюсь в некотором лукавстве: консультантом был я. Через полгода проект провалился, в значительной мере из-за того, что код стал слишком сложным для отладки или настройки, обеспечивающей приемлемую производительность.

Для повторного запуска проекта был привлечен консультант Кент Бек, и почти всю систему пришлось переписать сначала. Целый ряд вещей он организовал по-другому, причем важно, что он настоял на непрерывном исправлении кода с применением рефакторинга. Именно успех данного проекта и роль, которую сыграл в нем рефакторинг, побудили меня написать эту книгу и распространить те знания, которые Кент и другие специалисты приобрели в применении рефакторинга для повышения качества программного обеспечения.

Что такое рефакторинг?

Рефакторинг представляет собой процесс такого изменения программной системы, при котором не меняется внешнее поведение кода, но улучшается его внутренняя структура. Это способ систематического приведения кода в порядок, при котором шансы появления новых ошибок минимальны. В сущности, при проведении рефакторинга кода вы улучшаете его дизайн уже после того, как он написан.

«Улучшение кода после его написания» - непривычная фигура речи. В нашем сегодняшнем понимании разработки программного обеспечения мы сначала создаем дизайн системы, а потом пишем код. Сначала создается хороший дизайн, а затем происходит кодирование. Со временем код модифицируется, и целостность системы, соответствие ее структуры изначально созданному дизайну постепенно ухудшаются. Код медленно сползает от проектирования к хакерству.

Рефакторинг представляет собой противоположную практику. С ее помощью можно взять плохой проект, даже хаотический, и переделать его в хорошо спроектированный код. Каждый шаг этого процесса прост до чрезвычайности. Перемещается поле из одного класса в другой, изымается часть кода из метода и помещается в отдельный метод, какой-то код перемещается в иерархии в том или другом направлении. Однако суммарный эффект таких небольших изменений может радикально улучшить проект. Это прямо противоположно обычному явлению постепенного распада программы.

При проведении рефакторинга оказывается, что соотношение разных этапов работ изменяется. Проектирование непрерывно осуществляется во время разработки, а не выполняется целиком заранее. При

реализации системы становится ясно, как можно улучшить ее проект. Происходящее взаимодействие приводит к созданию программы, качество проекта которой остается высоким по мере продолжения разработки.

О чем эта книга?

Эта книга представляет собой руководство по рефакторингу и предназначена для профессиональных программистов. Автор ставил себе целью показать, как осуществлять рефакторинг управляемым и эффективным образом. Вы научитесь делать это, не внося в код ошибки и методично улучшая его структуру.

Принято помещать в начале книги введение. Я согласен с этим принципом, но было бы затруднительно начать знакомство с рефакторингом с общего изложения или определений. Поэтому я начну с примера. В главе 1 рассматривается небольшая программа, в дизайне которой есть распространенные недостатки, и с помощью рефакторинга она превращается в объектно-ориентированную программу более приемлемого вида. Попутно мы познакомимся как с процессом рефакторинга, так и несколькими полезными приемами в этом процессе. Эту главу важно прочесть, если вы хотите понять, чем действительно занимается рефакторинг.

В главе 2 более подробно рассказывается об общих принципах рефакторинга, приводятся некоторые определения и основания для осуществления рефакторинга. Обозначаются некоторые проблемы, связанные с рефакторингом. В главе 3 Кент Бек поможет мне описать, как находить «душок» в коде и как от него избавляться посредством рефакторинга. Тестирование играет важную роль в рефакторинге, поэтому в главе 4 описывается, как создавать тесты для кода с помощью простой среды тестирования Java с открытым исходным кодом.

Сердцевина книги - перечень методов рефакторинга - простирается с главы 5 по главу 12. Этот перечень ни в коей мере не является исчерпывающим, а представляет собой лишь начало полного каталога. В него входят те методы рефакторинга, которые я, работая в этой области, зарегистрировал на сегодняшний день. Когда я хочу сделать что-либо, например «Замену условного оператора полиморфизмом» (Replace Conditional with Polymorphism, 258), перечень напоминает мне, как сделать это безопасным пошаговым способом. Надеюсь, к этому разделу книги вы станете часто обращаться.

В данной книге описываются результаты, полученные многими другими исследователями. Некоторыми из них написаны последние главы. Так, в главе 13 Билл Апдайк (Bill Opdyke) рассказывает о трудностях, с которыми он столкнулся, занимаясь рефакторингом в коммерческих разработках. Глава 14 написана Доном Робертсом (Don Roberts) и Джоном Брантом (John Brant) и посвящена будущему автоматизированных средств рефакторинга. Для завершающего слова я предоставил главу 15 мастеру рефакторинга Кенту Беку.

Рефакторинг кода Java

Повсюду в этой книге использованы примеры на Java. Разумеется, рефакторинг может производиться и для других языков, и эта книга, как я думаю, будет полезна тем, кто работает с другими языками. Однако я решил сосредоточиться на Java, потому что этот язык я знаю лучше всего. Я сделал несколько замечаний по поводу рефакторинга в других языках, но надеюсь, что книги для конкретных языков (на основе имеющейся базы) напишут другие люди.

Стремясь лучше изложить идеи, я не касался особо сложных областей языка Java, поэтому воздержался от использования внутренних классов, отражения, потоков и многих других мощных возможностей Java. Это связано с желанием как можно понятнее изложить базовые методы рефакторинга.

Должен подчеркнуть, что приведенные методы рефакторинга не касаются параллельных или распределенных программ - в этих областях возникают дополнительные проблемы, решение которых выходит за рамки данной книги.

Для кого предназначена эта книга?

Книга рассчитана на профессионального программиста - того, кто зарабатывает программированием себе на жизнь. В примерах и тексте содержится масса кода, который надо прочесть и понять. Все примеры написаны на Java. Этот язык выбран потому, что он становится все более распространенным и его легко понять тем, кто имеет опыт работы с C. Кроме того, это объектно-ориентированный язык, а объектно-ориентированные механизмы очень полезны при проведении рефакторинга.

Хотя рефакторинг и ориентирован на код, он оказывает большое влияние на архитектуру системы. Руководящим проектировщикам и архитекторам важно понять принципы рефакторинга и использовать их в своих проектах. Лучше всего, если рефакторингом руководит уважаемый и опытный разработчик. Он лучше всех может понять принципы, на которых основывается рефакторинг, и адаптировать их для конкретной рабочей среды. Это особенно касается случаев применения языков, отличных от Java, т. к. придется адаптировать для них приведенные мною примеры.

Можно извлечь для себя максимальную пользу из этой книги, и не читая ее целиком.

- **Если вы хотите понять, что такое рефакторинг, прочтите главу 1**; приведенный пример должен сделать этот процесс понятным.
- **Если вы хотите понять, для чего следует производить рефакторинг**, прочтите первые две главы. Из них вы поймете, что такое рефакторинг и зачем его надо осуществлять.
- **Если вы хотите узнать, что должно подлежать рефакторингу**, прочтите главу 3. В ней рассказано о признаках, указывающих на необходимость рефакторинга.
- **Если вы хотите реально заняться рефакторингом**, прочтите первые четыре главы целиком. Затем просмотрите перечень (глава 5). Прочтите его, для того чтобы примерно представлять себе его содержание. Не обязательно разбираться во всем сразу. Когда вам действительно понадобится какой-либо метод рефакторинга, прочтите о нем подробно и используйте в своей работе. Перечень служит справочным разделом, поэтому нет необходимости читать его подряд. Следует также прочесть главы, написанные приглашенными авторами, в особенности главу 15.

Основы, которые заложили другие

Хочу в самом начале заявить, что эта книга обязана своим появлением тем, чья деятельность в последние десять лет служила развитию рефакторинга. В идеале эту книгу должен был написать кто-то из них, но время и силы для этого нашлись у меня.

Два ведущих поборника рефакторинга - Уорд Каннингем (Ward Cunningham) и Кент Бек (Kent Beck). Рефакторинг для них давно стал центральной частью процесса разработки и был адаптирован с целью использования его преимуществ. В частности, именно сотрудничество с Кентом раскрыло для меня важность рефакторинга и вдохновило на написание этой книги.

Ральф Джонсон (Ralph Johnson) возглавляет группу в Университете штата Иллинойс в Урбана-Шампань, известную практическим вкладом в объектную технологию. Ральф давно является приверженцем рефакторинга, и над этой темой работал ряд его учеников. Билл Апдайк (Bill Opdyke) был автором первого подробного печатного труда по рефакторингу, содержавшегося в его докторской диссертации. Джон Брант (John Brant) и Дон Роберте (Don Roberts) пошли дальше слов и создали инструмент - Refadoring Browser - для проведения рефакторинга программ Smalltalk.

Благодарности

Даже при использовании результатов всех этих исследований мне потребовалась большая помощь, чтобы написать эту книгу. В первую очередь, огромную помощь оказал Кент Бек. Первые семена были брошены в землю во время беседы с Кентом в баре Детройта, когда он рассказал мне о статье, которую писал для Smalltalk Report [Beck, hanoi]. Я не только позаимствовал из нее многие идеи для главы 1, но она побудила меня начать сбор материала по рефакторингу. Кент помог мне и в другом. У него возникла идея «кода с душком» (code smells), он подбадривал меня, когда было трудно и вообще очень много сделал, чтобы эта книга состоялась. Думаю, что он сам написал бы эту книгу значительно лучше, но, как я уже сказал, время нашлось у меня, и остается только надеяться, что я не навредил предмету.

После написания этой книги мне захотелось передать часть специальных знаний непосредственно от специалистов, поэтому я очень благодарен многим из этих людей за то, что они не пожалели времени и добавили в книгу часть материала. Кент Бек, Джон Брант, Уильям Ап-дайк и Дон Роберте написали некоторые главы или были их соавторами. Кроме того, Рич Гарзанити (Rich Garzaniti) и Рон Джеффрис (Ron Jeffries) добавили полезные врезки.

Любой автор скажет вам, что для таких книг, как эта, очень полезно участие технических рецензентов. Как всегда, Картер Шанклин (Carter Shanklin) со своей командой в Addison-Wesley собрали отличную бригаду дотошных рецензентов. В нее вошли:

- Кен Ауэр (Ken Auer) из Rolemodel Software, Inc.
- Джошуа Блох (Joshua Bloch) из Sun Microsystems, Java Software
- Джон Брант (John Brant) из Иллинойского университета в Урбана-Шампань
- Скотт Корли (Scott Corley) из High Voltage Software, Inc.
- Уорд Каннингэм (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 изложения и точность книги и выявили если не все, то часть ошибок, которые могут притаиться в любой рукописи. Хочу отметить пару наиболее заметных предложений, оказавших влияние на внешний вид книги. Уорд и Рон побудили меня сделать главу 1 в стиле «бок о бок». Джошуа Кериевски принадлежит идея дать в каталоге наброски кода.

Помимо официальной группы рецензентов было много неофициальных. Это люди, читавшие рукопись или следившие за работой над ней через Интернет и сделавшие ценные замечания. В их число входят Лейф Беннет (Leif Bennett), Майкл Фезерс (Michael Feathers), Майкл Финни (Michael Finney), Нейл Галарнье (Neil Galarneau), Хишем Газу-ли (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). Наверное, есть и другие, которых я не упомянул; приношу им свои извинения и благодарность.

Особенно интересными рецензентами были члены печально известной группы читателей из Университета штата Иллинойс в Урбана-Шампань. Поскольку эта книга в значительной мере служит отражением их работы, я особенно благодарен им за их труд, переданный мне в формате «real audio». В эту группу входят «Фред» Балагер (Fredrico «Fred» Balaguer), Джон Брант (John Brant), Ян Чай (Ian Chai), Брайен Фут (Brian Foote), Алехандра Гарридо (Alejandra Garrido), «Джон» Хан (Zhijiang «John» Han), Питер Хэтч (Peter Hatch), Ральф Джонсон (Ralph Johnson), «Раймонд» Лу (Songyu «Raymond» Lu), Драгос-Ан-тон Манолеску (Dragos-Anton Manolescu), Хироки Накамура (Hiroaki Nakamura), Джеймс Овертерф (James Overturf), Дон Роберте (Don Roberts), Чико Шираи (Chieko Shirai), Лес Тайрел (Les Tyrell) и Джо Йод ер (Joe Yoder).

Любые хорошие идеи должны проверяться в серьезной производственной системе. Я был свидетелем огромного эффекта, оказанного рефакторингом на систему Chrysler Comprehensive Compensation (C3), рассчитывающую заработную плату для примерно 90 тыс. рабочих фирмы Chrysler. Хочу поблагодарить всех участников этой команды: Энн Андерсон (Ann Anderson), Эда Андери (Ed Anderi), Ральфа Битти (Ralph Beattie), Кента Бека (Kent Beck), Дэвида Брайанта (David Bryant), Боба Коу (Bob Coe), Мари Д'Армен (Marie DeArment), Маргарет Фронзак (Margaret Fronczak), Рича Гарзанити (Rich Garzaniti), Денниса Гора (Dennis Gore), Брайена Хэкера (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), Эдриана Пантеа (Adrian Pantea), Мэтта Сайджена (Matt Saigeon), Дона Томаса (Don Thomas) и Дона Уэллса (Don Wells). Работа с ними, в первую очередь, укрепила во мне понимание принципов и выгод рефакторинга. Наблюдая за прогрессом, достигнутым ими, я вижу, что позволяет сделать рефакторинг, осуществляемый в большом проекте на протяжении ряда лет.

Мне помогали также 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 - удовольствие. Они предоставили мне большую поддержку и помощь.

Если говорить о поддержке, то больше всего неприятностей приносит книга тому, кто находится ближе всего к ее автору. В данном случае это моя жена Синди. Спасибо за любовь ко мне, сохранявшуюся даже тогда, когда я скрывался в своем кабинете. На протяжении всего времени работы над этой книгой меня не покидали мысли о тебе.

Martin Fowler

Melrose, Massachusetts

fowler@acm.org

http://www.martinfowler.com

http://www.refactoring.com

1 РЕФАКТОРИНГ, ПЕРВЫЙ ПРИМЕР

С чего начать описание рефакторинга? Обычно разговор о чем-то новом заключается в том, чтобы обрисовать историю, основные принципы и т. д. Когда на какой-нибудь конференции я слышу такое, меня начинает клонить ко сну. Моя голова переходит в работу в фоновом режиме с низким приоритетом, периодически производящем опрос докладчика, пока тот не начнет приводить примеры. На примерах я просыпаюсь, т. к. с их помощью могу разобраться в происходящем. Исходя из принципов очень легко делать обобщения и очень тяжело разобраться, как их применять на практике. С помощью примера все проясняется.

Поэтому я намерен начать книгу с примера рефакторинга. С его помощью вы много узнаете о том, как рефакторинг действует, и получите представление о том, как происходит его процесс. После этого можно дать обычное введение, знакомящее с основными идеями и принципами.

Однако с вводным примером у меня возникли большие сложности. Если выбрать большую программу, то читателю будет трудно разобраться с тем, как был проведен рефакторинг примера. (Я попытался было так сделать, и оказалось, что весьма несложный пример потянул более чем на сотню страниц.) Однако если выбрать небольшую программу, которую легко понять, то на ней не удастся продемонстрировать достоинства рефакторинга.

Я столкнулся, таким образом, с классической проблемой, возникающей у всех, кто пытается описывать технологии, применимые для реальных программ. Честно говоря, не стоило бы тратить силы на рефакторинг, который я собираюсь показать, для такой маленькой программы, которая будет при этом использована. Но если код, который я вам покажу, будет составной частью большой системы, то в этом случае рефакторинг сразу становится важным. Поэтому, изучая этот пример, представьте его себе в контексте значительно более крупной системы.

Исходная программа

Пример программы очень прост. Она рассчитывает и выводит отчет об оплате клиентом услуг в магазине видеопроката. Программе сообщается, какие фильмы брал в прокате клиент и на какой срок. После этого она рассчитывает сумму платежа исходя из продолжительности проката и типа фильма. Фильмы бывают трех типов: обычные, детские и новинки. Помимо расчета суммы оплаты начисляются бонусы в зависимости от того, является ли фильм новым.

Элементы системы представляются несколькими классами, показанными на диаграмме (рис. 1.1).

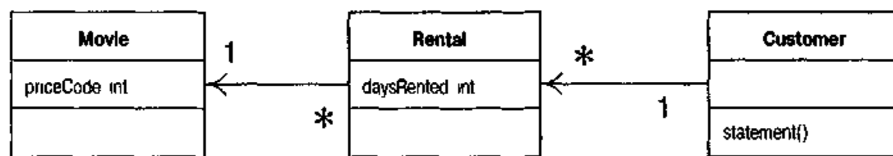


Рисунок - 1.1 Диаграмма классов для исходной программы. Нотация соответствует унифицированному языку моделирования [Fowler, UML]

Я поочередно приведу код каждого из этих классов.

Movie

Movie - класс, который представляет данные 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(Stnng 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

Rental - класс, представляющий 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

Customer - класс, представляющий клиента магазина. Как и предыдущие классы, он содержит данные и методы для доступа к ним:

```
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;
    }
```

В классе Customer есть метод, создающий отчет. На рис. 1.2. показано, с чем взаимодействует этот метод. Тело метода приведено ниже.

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

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

        //определить сумму для каждой строки
        switch (each.getMovie().getPriceCode()) {
           case Movie.REGULAR:
              thisAmount += 2;
              if (each.getDaysRented() > 2){
                 thisAmount += (each.getDaysRented() - 2) * 15;
              }
              break;

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

           case Movie.CHILDRENS:
              thisAmount += 15;
               if (each.getDaysRented() > 3){
                 thisAmount += (each.getDaysRented() - 3) * 15;
              }
              break;
        }

        // добавить очки для активного арендатора
        frequentRenterPoints ++;
        // бонус за аренду новинки на два дня
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
              each.getDaysRented() > 1) {
           frequentRenterPoints ++;
        }

        //показать результаты для этой аренды
```

```
      result += "\t" + each.getMovie().getTitle()+ "\t" +
         String.valueOf(thisAmount) + "\n";
      totalAmount += thisAmount;
   }

   //добавить нижний колонтитул
   result += "Сумма задолженности составляет" +
      String.valueOf(totalAmount) + "\n";
   result += "Вы заработали " + String.valueOf(frequentRenterPoints) +
      "очков за активность";

   return result,
  }
```

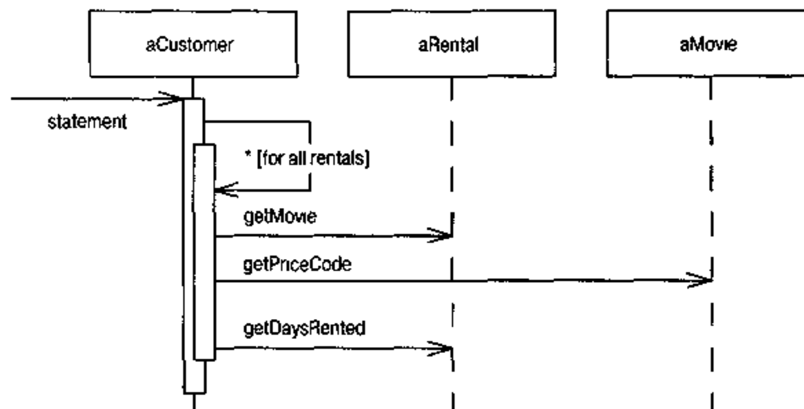


Рисунок 1.2 - Диаграмма взаимодействий для метода statement

Комментарии к исходной программе

Что вы думаете по поводу дизайна этой программы? Я бы охарактеризовал ее как слабо спроектированную и, разумеется, не объектно-ориентированную. Для такой простой программы это может оказаться не очень существенным. Нет ничего страшного, если простая программа написана на скорую руку. Но если этот фрагмент показателен для более сложной системы, то с этой программой возникают реальные проблемы. Слишком громоздкий метод statement класса Customer берет на себя слишком многое. Многое из того, что делает этот метод, в действительности должно выполняться методами других классов.

Но даже в таком виде программа работает. Может быть, это чисто эстетическая оценка, вызванная некрасивым кодом? Так оно и будет до попытки внести изменения в систему. Компилятору все равно, красив код или нет. Но в процессе внесения изменений в систему участвуют люди, которым это не безразлично. Плохо спроектированную систему трудно модифицировать. Трудно потому, что нелегко понять, где изменения нужны. Если трудно понять, что должно быть изменено, то есть большая вероятность, что программист ошибется.

В данном случае есть модификация, которую хотелось бы осуществить пользователям. Во-первых, им нужен отчет, выведенный в HTML, который был бы готов для публикации в Интернете и полностью соответствовал модным веяниям. Посмотрим, какое влияние окажет такое изменение. Если взглянуть на код, можно увидеть, что невозможно повторно использовать текущий метод statement для создания отчета в HTML. Остается только написать метод целиком заново, в основном дублируя поведение прежнего. Для такого примера это, конечно, не очень обременительно. Можно просто скопировать метод statement и произвести необходимые изменения.

Но что произойдет, если изменятся правила оплаты? Придется изменить как statement, так и htmlStatement, проследив за тем, чтобы изменения были согласованы. Проблема, сопутствующая копированию и вставке, обнаружится позже, когда код придется модифицировать. Если не предполагается в будущем изменять создаваемую программу, то можно обойтись копированием и вставкой. Если программа будет служить долго и предполагает внесение изменений, то копирование и вставка представляют собой угрозу.

Это приводит ко второму изменению. Пользователи подумывают о целом ряде изменений способов классификации фильмов, но еще не определились окончательно. Изменения коснутся как системы оплаты за

аренду фильмов, так и порядка начисления очков активным пользователям. Как опытный разработчик вы можете быть уверены, что к какой бы схеме оплаты они ни пришли, гарантировано, что через полгода это будет другая схема.

Метод statement - то место, где нужно произвести изменения, соответствующие новым правилам классификации и оплаты. Однако при копировании отчета в формат HTML необходимо гарантировать полное соответствие всех изменений. Кроме того, по мере роста сложности правил становится все труднее определить, где должны быть произведены изменения, и осуществить их, не сделав ошибки.

Может возникнуть соблазн сделать в программе как можно меньше изменений: в конце концов, она прекрасно работает. Вспомните старую поговорку инженеров: «не чините то, что еще не сломалось». Возможно, программа исправна, но доставляет неприятности. Это осложняет жизнь, потому что будет затруднительно произвести модификацию, необходимую пользователям. Здесь вступает в дело рефакторинг.

Примечание

Обнаружив, что в программу необходимо добавить новую функциональность, но код программы не структурирован удобным для добавления этой функциональности образом, сначала произведите рефакторинг программы, чтобы упростить внесение необходимых изменений, а только потом добавьте функцию.

Первый шаг рефакторинга

Приступая к рефакторингу, я всегда начинаю с одного и того же: строю надежный набор тестов для перерабатываемой части кода. Тесты важны потому, что, даже последовательно выполняя рефакторинг, необходимо исключить появление ошибок. Ведь я, как и всякий человек, могу ошибиться. Поэтому мне нужны надежные тесты.

Поскольку в результате statement возвращается строка, я создаю несколько клиентов с арендой каждым нескольких типов фильмов и генерирую строки отчетов. Далее сравниваю новые строки с контрольными, которые проверил вручную. Все тесты я организую так, чтобы можно было запускать их одной командой. Выполнение тестов занимает лишь несколько секунд, а запускаю я их, как вы увидите, весьма часто.

Важную часть тестов составляет способ, которым они сообщают свои результаты. Они либо выводят «ОК», что означает совпадение всех строк с контрольными, либо выводят список ошибок - строк, не совпавших с контрольными данными. Тесты, таким образом, являются самопроверяющимися. Это важно, потому что иначе придется вручную сравнивать получаемые результаты с теми, которые выписаны у вас на бумаге, и тратить на это время.

Проводя рефакторинг, будем полагаться на тесты. Если мы допустим в программе ошибку, об этом нам должны сообщить тесты. При проведении рефакторинга важно иметь хорошие тесты. Время, потраченное на создание тестов, того стоит, поскольку тесты гарантируют, что можно продолжать модификацию программы. Это настолько важный элемент рефакторинга, что я подробнее остановлюсь на нем в главе 4.

Примечание

Перед началом рефакторинга убедитесь, что располагаете надежным комплектом тестов. Эти тесты должны быть самопроверяющимися.

Декомпозиция и перераспределение метода statement

Очевидно, что мое внимание было привлечено в первую очередь к непомерно длинному методу statement. Рассматривая такие длинные методы, я приглядываюсь к способам разложения их на более мелкие составные части. Когда фрагменты кода невелики, облегчается управление. С такими фрагментами проще работать и перемещать их.

Первый этап рефакторинга в данной главе показывает, как я разделяю длинный метод на части и перемещаю их в более подходящие классы. Моя цель состоит в том, чтобы облегчить написание метода вывода отчета в HTML с минимальным дублированием кода.

Первый шаг состоит в том, чтобы логически сгруппировать код и использовать «Выделение метода» (Extract Method). Очевидный фрагмент для выделения в новый метод составляет здесь команда switch.

При выделении метода, как и при любом другом рефакторинге, я должен знать, какие неприятности могут случиться. Если плохо выполнить выделение, можно внести в программу ошибку. Поэтому перед выполнением рефакторинга нужно понять, как провести его безопасным образом. Ранее я уже неоднократно проводил такой рефакторинг и записал безопасную последовательность шагов.

Сначала надо выяснить, есть ли в данном фрагменте переменные с локальной для данного метода областью видимости - локальные переменные и параметры. В этом сегменте кода таких переменных две: each и thisAmount. При этом each не изменяется в коде, a thisAmount - изменяется. Все немодифицируемые

переменные можно передавать как параметры. С модифицируемыми переменными сложнее. Если такая переменная только одна, ее можно вернуть из метода. Временная переменная инициализируется нулем при каждой итерации цикла и не изменяется, пока до нее не доберется switch. Поэтому значение этой переменной можно просто вернуть из метода.

На следующих двух страницах показан код до и после проведения рефакторинга. Первоначальный код показан слева, результирующий код - справа. Код, извлеченный мной из оригинала, и изменения в новом коде выделены полужирным шрифтом. Данного соглашения по левой и правой страницам я буду придерживаться в этой главе и далее.

```
public String statement() {
 double totalAmount = 0;
 int frequentRenterPoints = 0;
 Enumeration rentals = _rentals.elements();
 String result = "Учет аренды для " + getName() + "\n";
 while (rentals.hasMoreElements()) {
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement();

    // определить сумму 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.getDaysRentedQ > 3){
             thisAmount += (each.getDaysRented() - 3) * 1.5;
          }
          break;
    }

    // добавить очки для активного арендатора
    frequentRenterPoints ++;
    // добавить бонус за аренду новинки на два дня
    if ((each getMovie().getPnceCode() == Movie NEW_RELEASE) &&
          each getDaysRented() > 1){
       frequentRenterPoints ++;
    }
    // показать результаты для этой аренды
    result += "\t" + each.getMovie().getTitle()+ "\t" +
          String.valueOf(thisAmount) + "\n";
```

```
        totalAmount += thisAmount;
        }

        // добавить нижний колонтитул
        result += "Сумма задолженности составляет " +
                  String.valueOf(totalAmount) +
                  "\n";
        result += "Вы заработали " + String.valueOf(frequentRenterPoints) +
                "очков за активность ";
        return result
      }
    ---------------------------------------------------------------------------------
-
    public String statement() {
     double totalAmount = 0;
     int frequentRenterPoints = 0;
     Enumeration rentals = _rentals.elements();
     String result = "Учет аренды для" + getName() + "\n";

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

        // добавить очки для активного арендатора
        frequentRenterPoints ++;
        // добавить бонус за аренду новинки на два дня
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
              each.getDaysRented() > 1) {
           frequentRenterPoints ++;
        }

        // показать результаты для этой аренды
        result += "\t" + each.getMovie().getTitle()+ "\t" +
              String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
     }

     // добавить нижний колонтитул
     result += "Сумма задолженности составляет " +
           String.valueOf(totalAmount) + "\n";
     result += "Вы заработали " + String.valueOf(frequentRenterPomts) +
              "очков за активность";

     return result;
    }
```

```
private int amountFor(Rental each) {
 int thisAmount = 0;
 switch (each getMovie().getPriceCode()) {
   case Movie.REGULAR:
      thisAmount += 2;
      if (each getDaysRentedO > 2) {
         thisAmount += each.getDaysRented() - 2) * 1 5;
      }
      break;

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

   case Movie.CHILDRENS:
      thisAmount += 15;
      if (each.getDaysRented() > 3) {
         thisAmount += (each.getDaysRented() - 3) * 1 5;
      }
      break;
 }
 return thisAmount;
}
```

После внесения изменений я компилирую и тестирую код. Старт оказался не слишком удачным - тесты «слетели». Пара цифр в тестах оказалась неверной. После нескольких секунд размышлений я понял, что произошло. По глупости я задал тип возвращаемого значения amountFor как int вместо 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) * 15;
      }
      break;

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

   case Movie.CHILDRENS:
      thisAmount += 15;
      if (each.getDaysRented() > 3) {
         thisAmount += (each.getDaysRented() - 3) * 15;
      }
```

```
        break;
    }
    return thisAmount;
  }
```

Такие глупые ошибки я делаю часто, и выследить их бывает тяжело. В данном случае Java преобразует double в int без всяких сообщений путем простого округления [Java Spec]. К счастью, обнаружить это было легко, потому что модификация была незначительной, а набор тестов - хорошим. Этот случай иллюстрирует сущность процесса рефакторинга. Поскольку все изменения достаточно небольшие, ошибки отыскиваются легко. Даже такому небрежному программисту, как я, не приходится долго заниматься отладкой.

Примечание

При применении рефакторинга программа модифицируется небольшими шагами. Ошибку нетрудно обнаружить.

Поскольку я работаю на Java, приходится анализировать код и определять, что делать с локальными переменными. Однако с помощью специального инструментария подобный анализ проводится очень просто. Такой инструмент существует для Smalltalk, называется Refactoring Browser и весьма упрощает рефакторинг. Я просто выделяю код, выбираю в меню Extract Method (Выделение метода), ввожу имя метода, и все готово. Более того, этот инструмент не делает такие глупые ошибки, какие мы видели выше. Мечтаю о появлении его версии для Java!

Разобрав исходный метод на куски, можно работать с ними отдельно. Мне не нравятся некоторые имена переменных в amountFor, и теперь хорошо бы 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) * 15;
          }
          break;

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

          case Movie.CHILDRENS:
             thisAmount += 15;
             if (each.getDaysRented() > 3) {
                thisAmount += (each.getDaysRented90 - 3) * 15;
             }
             break;
       }
    return thisAmount;
  }
```

А вот код после переименования:

```
  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) * 15;
         }
         break;

      case Movie.NEW_RELEASE:
         result += aRental.getDaysRented() * 3;
         break;

      case Movie.CHILDRENS:
         result += 15;
         if (aRental.getDaysRented() > 3) {
            result += (aRental.getDaysRented() - 3) * 15;
         }
         break;
   }
   return result;
  }
```

Закончив переименование, я компилирую и тестирую код, чтобы проверить, не испортилось ли что-нибудь.

Стоит ли заниматься переименованием? Без сомнения, стоит. Хороший код должен ясно сообщать о том, что он делает, и правильные имена переменных составляют основу понятного кода. Не бойтесь изменять имена, если в результате код становится более ясным. С помощью хороших средств поиска и замены сделать это обычно несложно. Строгий контроль типов и тестирование выявят возможные ошибки. Запомните:

Примечание

Написать код, понятный компьютеру, может каждый, но только хорошие программисты пишут код, понятный людям.

Очень важно, чтобы код сообщал о своей цели. Я часто провожу рефакторинг, когда просто читаю некоторый код. Благодаря этому свое понимание работы программы я отражаю в коде, чтобы впоследствии не забыть понятое.

Перемещение кода расчета суммы

Глядя на amountFor, можно заметить, что в этом методе используются данные класса, представляющего аренду, но не используются данные класса, представляющего клиента.

```
  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) * 15;
            }
            break;

         case Movie.NEW_RELEASE:
```

```
            result += aRental.getDaysRented() * 3;
            break;

         case Movie.CHILDRENS:
            result += 15;
            if (aRental.getDaysRented() > 3) {
               result += (aRental.getDaysRented() - 3) * 15;
            }
            break;
      }
      return result;
   }
```

В результате сразу возникает подозрение в неправильном выборе объекта для метода. В большинстве случаев метод должен связываться с тем объектом, данные которого он использует, поэтому его необходимо переместить в класс, представляющий аренду. Для этого я применяю «Перемещение метода» (Move Method). При этом надо сначала скопировать код в класс, представляющий аренду, настроить его в соответствии с новым местоположением и скомпилировать, как показано ниже:

```
   class Rental {
    double getCharge() {
      double result = 0;
      switch (getMovie().getPriceCode() {
         case Movie.REGULAR:
            result += 2;
            if (getDaysRented() > 2){
               result += (getDaysRented() - 2) * 15;
            }
            break;

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

         case Movie.CHILDRENS:
            result += 15;
            if (getDaysRented() > 3) {
               result += (getDaysRented() - 3) * 15;
            }
            break;
      }
    return result;
   }
```

В данном случае подгонка к новому местонахождению означает удаление параметра. Кроме того, при перемещении метода я его переименовал.

Теперь можно проверить, работает ли метод. Для этого я изменяю тело метода Customer amountFor так, чтобы обработка передавалась новому методу.

```
   class Customer{
    private double amountFor(Rental aRental) {
```

```
      return aRental.getCharge();
   }
  }
```

Теперь можно выполнить компиляцию и посмотреть, не нарушилось ли что-нибудь.

Следующим шагом будет нахождение всех ссылок на старый метод и настройка их на использование нового метода:

```
  class Customer {
   public String statement() {
      double totalAmount = 0;
      int frequentRenterPoints = 0;
      Enumeration rentals = _rentals.elements();
      String result = "Учет аренды для " + getName() + "\n"
      while (rentals.hasMoreElements()) {
         double thisAmount = 0;
         Rental each = (Rental)rentals.nextElement();
         thisAmount = amountFor(each);
         // добавить очки для активного арендатора
         frequentRenterPoints ++;
         // добавить бонус за аренду новинки на два дня
         if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
               each.getDaysRented() > 1) {
            frequentRenterPoints ++;
         }
         //показать результаты для этой аренды
         result += "\t" + each.getMovie().getTitle()+ "\t" +
            String.valueOf(thisAmount) + "\n";
         totalAmount += thisAmount;
      }
      //добавить нижний колонтитул
      result += "Сумма задолженности составляет " +
         String.valueOf(totalAmount) + "\n";
      result += "Вы заработали" + String.valueOf(frequentRenterPoints) +
         "очков за активность";

      return result;
   }
  }
```

В данном случае этот шаг выполняется просто, потому что мы только что создали метод и он находится только в одном месте. В общем случае, однако, следует выполнить поиск по всем классам, которые могут использовать этот метод:

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

```
        while (rentals.hasMoreElements()) {
            double thisAmount = 0;
            Rental each = (Rental) rentals.nextElement();
            thisAmount = each.getCharge();
            // добавить очки для активного арендатора
            frequentRenterPoints ++;
            // добавить бонус за аренду новинки на два дня
            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                    each.getDaysRented() > 1) {
                frequentRenterPoints ++;
            }
            //показать результаты для этой аренды
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                String.valueOf(thisAmount) + "\n";
            totalAmount += thisAmount;
        }
        //добавить нижний колонтитул
        result += "Сумма задолженности составляет" + String.valueOf(totalAmount) + "\n";
        result += "Вы заработали" + String.value0f(frequentRenterPoints) +
            "очков за активность";

        return result;
    }
   }
```

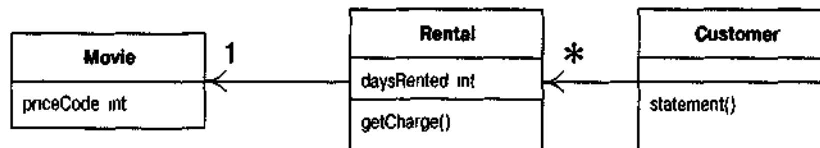


Рисунок 1.3 - Диаграмма классов после перемещения метода, вычисляющего сумму оплаты

После внесения изменений (рис. 1.3) нужно удалить старый метод. Компилятор сообщит, не пропущено ли чего. Затем я запускаю тесты и проверяю, не нарушилась ли работа программы.

Иногда я сохраняю старый метод, но для обработки в нем привлекается новый метод. Это полезно, если метод объявлен с модификатором видимости public, а изменять интерфейс других классов я не хочу.

Конечно, хотелось бы еще кое-что сделать с Rental.getCharge, но оставим его на время и вернемся к Customer statement.

```
   public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Учет аренды для " + getName() + "\n";
    while (rentals. hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();
        thisAmount = each.getCharge();
        // добавить очки для активного арендатора
        frequentRenterPoints ++;
```

```
      // добавить бонус за аренду новинки на два дня
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
          each.getDaysRentedO > 1) {
        frequentRenterPoints ++;
      }
      //показать результаты для этой аренды
      result += "\t" + each.getMovie().getTitle()+ "\t" +
        String.valueOf(thisAmount) + "\n";
      totalAmount += thisAmount;
    }
    //добавить нижний колонтитул
    result += "Сумма задолженности составляет " + String.valueOf(totalAmount) + "\n";
    result += "Вы заработали " + String.valueOf(frequentRenterPoints) +
      " очков за активность";

    return result;
   }
```

Теперь мне приходит в голову, что переменная thisAmount стала избыточной. Ей присваивается результат each.charge(), который впоследствии не изменяется. Поэтому можно исключить thisAmount, применяя «Замену временной переменной вызовом метода» (Replace Temp with Query):

```
   public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Учет аренды для " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      // добавить очки для активного арендатора
      frequentRenterPoints ++;
      // добавить бонус за аренду новинки на два дня
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
          each.getDaysRented() > 1) {
        frequentRenterPoints ++;
      }
      //показать результаты для этой аренды
      result += "\t" + each.getMovie().getTitle()+ "\t" +
        String.valueOf(each.getCharge()) + "\n";
      totalAmount += each.getCharge();
    }
    //добавить нижний колонтитул
    result += "Сумма задолженности составляет " +
      String.valueOf(totalAmount) + "\n";
    result += "Вы заработали " + String.valueOf(frequentRenterPoints) +
      "очков за активность";

   return result;
```

```
    }
```

Проделав это изменение, я выполняю компиляцию и тестирование, чтобы удостовериться, что ничего не нарушено.

Я стараюсь по возможности избавляться от таких временных переменных. Временные переменные вызывают много проблем, потому что из-за них приходится пересылать массу параметров там, где этого можно не делать. Можно легко забыть, для чего эти переменные введены. Особенно коварными они могут оказаться в длинных методах. Конечно, приходится расплачиваться снижением производительности: теперь сумма оплаты стала у нас вычисляться дважды. Но это можно оптимизировать в классе аренды, и оптимизация оказывается значительно эффективнее, когда код правильно разбит на классы. Я буду говорить об этом далее в разделе «Рефакторинг и производительность».

Выделение начисления бонусов в метод

На следующем этапе аналогичная вещь производится для подсчета бонусов. Правила зависят от типа пленки, хотя вариантов здесь меньше, чем для оплаты. Видимо, разумно переложить ответственность на класс аренды. Сначала надо применить процедуру «Выделение метода» (Extract Method) к части кода, осуществляющей начисление бонусов (выделена полужирным шрифтом):

```
public String statement() {
 double totalAmount = 0;
 int frequentRenterPoints = 0;
 Enumeration rentals = _rentals.elements();
 String result = "Учет аренды для " + getName() + "\n";
 while (rentals.hasMoreElements()) {
    Rental each = (Rental) rentals.nextElement();
    // добавить очки для активного арендатора
    frequentRenterPoints ++;
    // добавить бонус за аренду новинки на два дня
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
         each.getDaysRented() > 1) {
       frequentRenterPoints ++;
    }
    //показать результаты для этой аренды
    result += "\t" + each.getMovie().getTitle()+ "\t" +
       String.valueOf(each.getCharge()) + "\n";
    totalAmount += each.getCharge();
 }
 //добавить нижний колонтитул
 result += "Сумма задолженности составляет " +
    String.valueOf(totalAmount) + "\n";
 result += "Вы заработали " + String.valueOf(frequentRenterPoints) +
    "очков за активность";

return result;
}
```

И снова ищем переменные с локальной областью видимости. Здесь опять фигурирует переменная each, которую можно передать в качестве параметра. Есть и еще одна временная переменная - frequentRenterPoints. В данном случае frequentRenterPoints имеет значение, присвоенное ей ранее. Однако в теле выделенного метода нет чтения значения этой переменной, поэтому не следует передавать ее в качестве параметра, если пользоваться присваиванием со сложением.

Я произвел выделение метода, скомпилировал и протестировал код, а затем произвел перемещение и снова выполнил компиляцию и тестирование. При рефакторинге лучше всего продвигаться маленькими шагами, чтобы не столкнуться с неприятностями.

```
class Customer {
 public String statement() {
 double totalAmount = 0;
 int frequentRenterPoints = 0;
 Enumeration rentals = _rentals.elements();
 String result = ”Учет аренды для ” + getName() + “\n”;
 while (rentals.hasMoreElements()) {
    Rental each = (Rental) rentals.nextElement();
    frequentRenterPoints += each.getFrequentRenterPoints();
    //показать результаты для этой аренды
    result += “\t” + each.getMovie().getTitle()+ “\t” +
       String.valueOf(each.getCharge()) + “\n”;
    totalAmount += each.getCharge()
 }
 //добавить нижний колонтитул
 result += “Сумма задолженности составляет ” +
    String.valueOf(totalAmount) + “\n”;
 result += “Вы заработали ” + String.valueOf(frequentRenterPoints) +
    очков за активность”;
 return result;
}

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

Я подытожу только что произведенные изменения с помощью диаграмм унифицированного языка моделирования (UML) для состояния «до» и «после» (рисунках 1.4 - 1.7). Как обычно, слева находятся диаграммы, отражающие ситуацию до внесения изменений, а справа - после внесения изменений.

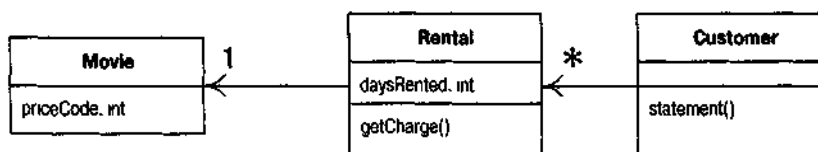


Рисунок 1.4 - Диаграмма классов до выделения кода начисления бонусов и его перемещения

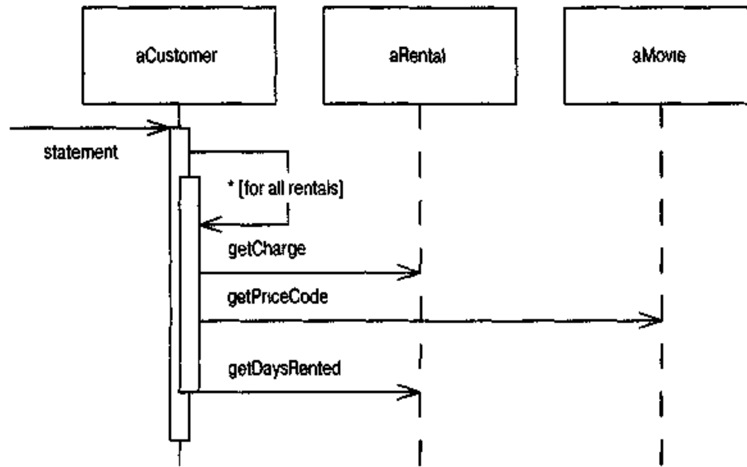


Рисунок 1.5 - Диаграмма последовательности до выделения кода начисления бонусов и его перемещения

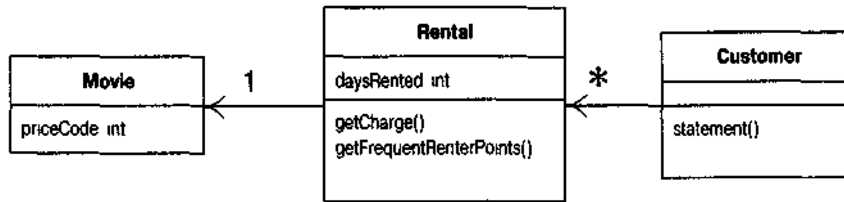


Рисунок 1.6 - Диаграмма классов после выделения кода начисления бонусов и его перемещения

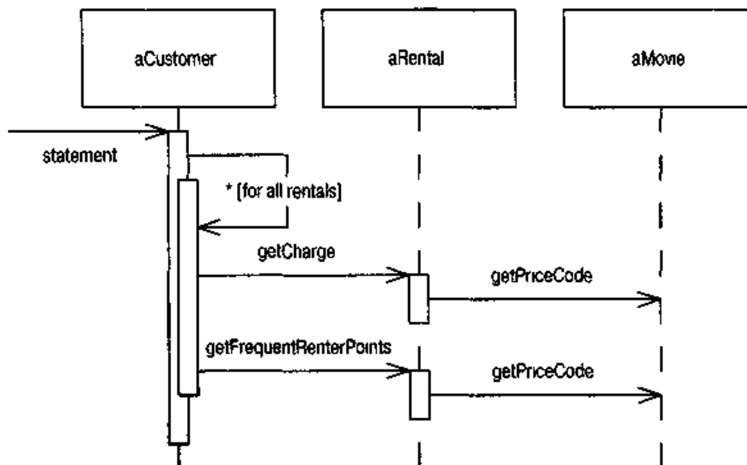


Рисунок 1.7 - Диаграмма последовательности после выделения кода начисления бонусов и его перемещения

Удаление временных переменных

Как уже говорилось, из-за временных переменных могут возникать проблемы. Они используются только в своих собственных методах и приводят к появлению длинных и сложных методов. В нашем случае есть две временные переменные, участвующие в подсчете итоговых сумм по арендным операциям данного клиента. Эти итоговые суммы нужны для обеих версий - ASCII и HTML. Я хочу применить процедуру «Замены временной переменной вызовом метода» (Replace Temp with Query), чтобы заменить totalAmount и frequentRenterPoints на вызов метода. Замена временных переменных вызовами методов способствует более понятному архитектурному дизайну без длинных и сложных методов:

```
class Customer {
 public String statement() {
 double totalAmount = 0;
 int frequentRenterPoints = 0;
 Enumeration rentals = _rentals.elements();
 String result = ”Учет аренды для ” + getName() + “\n”;
 while (rentals.hasMoreElements()) {
    Rental each = (Rental) rentals.nextElement();
```

```
      frequentRenterPoints += each.getFrequentRenterPoints();
      //показать результаты для этой аренды
      result += "\t" + each.getMovie().getTitle()+ "\t" +
        String.valueOf(each.getCharge()) + "\n";
      totalAmount += each.getCharge()
    }
    //добавить нижний колонтитул
    result += "Сумма задолженности составляет " +
      String.valueOf(totalAmount) + "\n";
    result += "Вы заработали " + String.valueOf(frequentRenterPoints) +
      очков за активность";
    return result;
  }
```

Я начал с замены временной переменной totalAmount на вызов метода getTotalCharge класса клиента:

```
  class Customer {
    public String statement() {
      int frequentRenterPoints = 0;
      Enumeration rentals = _rentals.elements();
      String result = "Учет аренды для " + getName() + "\n";
      while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        frequentRenterPoints += each.getFrequentRenterPoints();
        //показать результаты для этой аренды
        result += "\t" + each.getMovie().getTitle()+ "\t" +
          String.valueOf(each.getCharge()) + "\n";
      }
      //добавить нижний колонтитул
      result += "Сумма задолженности составляет " +
        String.valueOf(getTotalCharge()) + "\n";
      result += "Вы заработали " + String.valueOf(frequentRenterPoints) +
        "очков за активность";
      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 самый простой случай «Замены временной переменной вызовом метода» (Replace Temp with Query): присваивание totalAmount выполнялось в цикле, поэтому придется копировать цикл в метод запроса.

После компиляции и тестирования результата рефакторинга то же самое я проделал для frequentRenterPoints:

```
class Customer {
 public String statement() {
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Учет аренды для " + getName() + "\n";
    while (rentals.hasMoreElements()) {
       Rental each = (Rental) rentals.nextElement();
       frequentRenterPoints += each.getFrequentRenterPoints();
       //показать результаты для этой аренды
       result += "\t" + each.getMovie().getTitle()+ "\t" +
          String.valueOf(each.getCharge()) + "\n";
    }
    //добавить нижний колонтитул
    result += "Сумма задолженности составляет " +
       String.valueOf(getTotalCharge()) + "\n";
    result += "Вы заработали " + String.valueOf(frequentRenterPoints) +
       "очков за активность";
    return result
 }
---------------------------------------------------------------------------------
public String statement() {
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Учет аренды для " + getName() + "\n";
    while (rentals.hasMoreElements()) {
       Rental each = (Rental) rentals.nextElement();
       frequentRenterPoints += each.getFrequentRenterPoints();
       //показать результаты для этой аренды
       result += "\t" + each.getMovie().getTitle()+ "\t" +
          String.valueOf(each.getCharge()) + "\n";
    }
    //добавить нижний колонтитул
    result += "Сумма задолженности составляет " +
       String.valueOf(getTotalCharge()) + "\n";
    result += "Вы заработали " + String.valueOf(getFrequentRenterPoints()) +
       "очков за активность";
    return result
 }
private int getTotalFrequentRenterPoints(){
 int result = 0;
 Enumeration rentals = _rentals.elements();
 while (rentals.hasMoreElements()) {
    Rental each = (Rental) rentals.nextEleraent();
    result += each.getFrequentRenterPoints();
 }
 return result;
```

```
    }
```

На рисунках 1.8 - 1.11 показаны изменения, внесенные в процессе рефакто-ринга в диаграммах классов и диаграмме взаимодействия для метода statement.

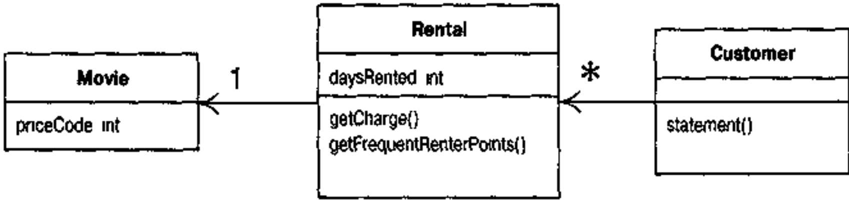


Рисунок 1.8 - Диаграмма классов до выделения подсчета итоговых сумм

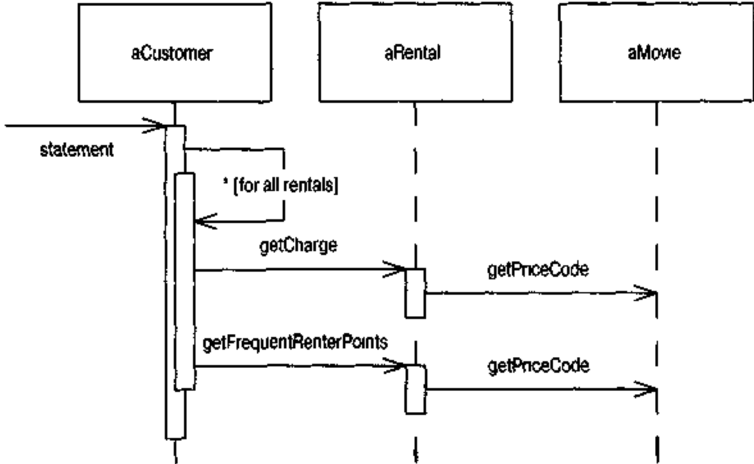


Рисунок 1.9 - Диаграмма последовательности до выделения подсчета итоговых сумм

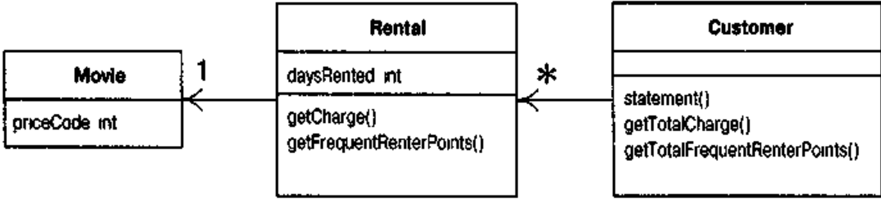


Рисунок 1.10 - Диаграмма классов после выделения подсчета итоговых сумм

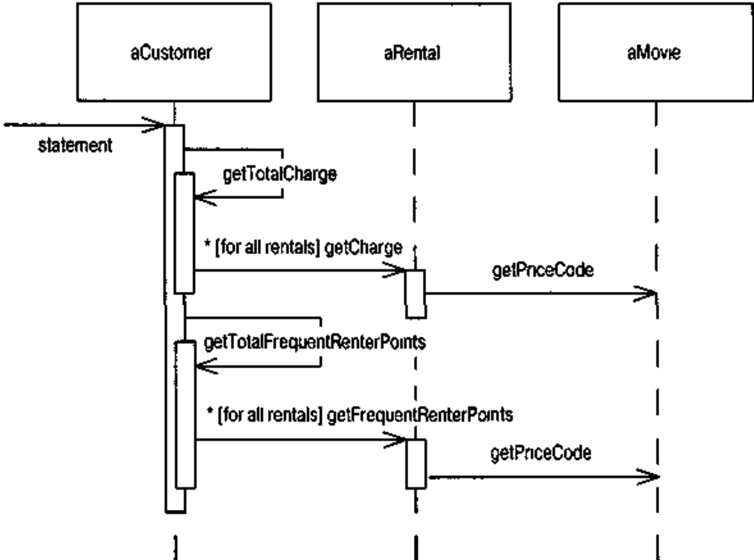


Рисунок 1.11 - Диаграмма последовательности после выделения подсчета итоговых сумм

Стоит остановиться и немного поразмышлять о последнем рефакторинге. При выполнении большинства процедур рефакторинга объем кода уменьшается, но в данном случае все наоборот. Дело в том, что в Java 1.1 требуется много команд для организации суммирующего цикла. Даже для простого цикла суммирования с

одной строкой кода для каждого элемента нужны шесть вспомогательных строк. Это идиома, очевидная для любого программиста, но, тем не менее, она состоит из большого количества строк.

Также вызывает беспокойство, связанное с такого вида рефакторингом, возможное падение производительности. Прежний код выполнял цикл while один раз, новый код выполняет его три раза. Долго выполняющийся цикл while может снизить производительность. Многие программисты отказались бы от подобного рефакторинга лишь по данной причине. Но обратите внимание на слово «может». До проведения профилирования невозможно сказать, сколько времени требует цикл для вычислений и происходит ли обращение к циклу достаточно часто, чтобы повлиять на итоговую производительность системы. Не беспокойтесь об этих вещах во время проведения рефакторинга. Когда вы приступите к оптимизации, тогда и нужно будет об этом беспокоиться, но к тому времени ваше положение окажется значительно более выгодным для ее проведения и у вас будет больше возможностей для эффективной оптимизации.

Созданные мной методы доступны теперь любому коду в классе Customer. Их нетрудно добавить в интерфейс класса, если данная информация потребуется в других частях системы. При отсутствии таких методов другим методам приходится разбираться с устройством класса аренды и строить циклы. В сложной системе для этого придется написать и сопровождать значительно больший объем кода.

Вы сразу почувствуете разницу, увидев htmlStatement. Сейчас я перейду от рефакторинга к добавлению методов. Можно написать такой код htmlStatement и добавить соответствующие тесты:

```
public String htmlStatement() {
 Enumeration rentals = _rentals.elements();
 String result = "<H1>Операции аренды для <EM>" + getName() + </EM></H1><P>\n";
 while (rentals hasMoreElements()) {
    Rental each = (Rental) rentals.nextElement();
    // показать результаты по каждой аренде
    result += each.getMovie().getTitle() +
       String.valueOf(each.getCharge()) + "<BR>\n";
 }
 //добавить нижний колонтитул
 result += "<P>Ваша задолженность составляет <EM>" +
    String.valueOf(getTotalCharge()) + "</EM><P>\n";
 result += "На этой аренде вы заработали <EM>" +
    String.valueOf(getTotalFrequentRenterPoints()) + "</EM> очков за активность<P>";
 return result;
}
```

Выделив расчеты, я смог создать метод htmlStatement и при этом повторно использовать весь код для расчетов, имевшийся в первоначальном методе statement. Это было сделано без копирования и вставки, поэтому если правила расчетов изменятся, переделывать код нужно будет только в одном месте. Любой другой вид отчета можно подготовить быстро и просто. Рефакторинг не отнял много времени. Большую часть времени я выяснял, какую функцию выполняет код, а это пришлось бы делать в любом случае.

Часть кода скопирована из ASCII-версии, в основном из-за организации цикла. При дальнейшем проведении рефакторинга это можно было бы улучшить. Выделение методов для заголовка, нижнего колонтитула и строки с деталями - один из путей, которыми можно пойти. Как это сделать, можно узнать из примера для «Формирования шаблона метода» (Form Template Method). Но теперь пользователи выдвигают новые требования. Они собираются изменить классификацию фильмов. Пока неясно, как именно, но похоже, что будут введены новые категории, а существующие могут измениться. Для этих новых категорий должен быть установлен порядок оплаты и начисления бонусов. В данное время осуществление такого рода изменений затруднено. Чтобы модифицировать классификацию фильмов, придется изменять условные операторы в методах начисления оплаты и бонусов. Снова седлаем коня рефакторинга.

Замена условной логики на полиморфизм

Первая часть проблемы заключается в блоке switch. Организовывать переключение в зависимости от атрибута другого объекта - неудачная идея. Если уж без оператора switch не обойтись, то он должен основываться на ваших собственных, а не чужих данных.

```
class Rental {
```

```
 double getCharge() {
   double result = 0;
   switch (getMovie().getPriceCode()) {
     case Movie.REGULAR:
       result += 2;
       if (getDaysRented() > 2) {
         result += (getDaysRented() - 2) * 15;
       }
       break;

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

     case Movie.CHILDRENS:
       result += 15;
       if (getDaysRented() > 3) {
         result += (getDaysRented() - 3) * 15;
       }
       break;
   }
   return result;
 }
}
```

Это подразумевает, что getCharge должен переместиться 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) * 15;
       }
       break;

     case Movie.NEW_RELEASE:
       result += daysRented * 3;
       break;

     case Movie.CHILDRENS:
       result += 15;
       if (daysRented > 3) {
         result += (daysRented - 3) * 15;
       }
       break;
```

```
        }
        return result;
    }
  }
```

Для того чтобы этот код работал, мне пришлось передавать в него продолжительность аренды, которая, конечно, представляет собой данные из Rental. Метод фактически использует два элемента данных - продолжительность аренды и тип фильма. Почему я предпочел передавать продолжительность аренды в Movie, а не тип фильма в Rental? Потому что предполагаемые изменения касаются только введения новых типов. Данные о типах обычно оказываются более подверженными изменениям. Я хочу, чтобы волновой эффект изменения типов фильмов был минимальным, поэтому решил рассчитывать сумму оплаты в Movie.

Я поместил этот метод в Movie и изменил getCharge для Rental так, чтобы использовался новый метод (рисунки 1.12 и 1.13):

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

Переместив метод getCharge, я произведу то же самое с методом начисления бонусов. В результате оба метода, зависящие от типа, будут сведены в класс, который содержит этот тип:

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

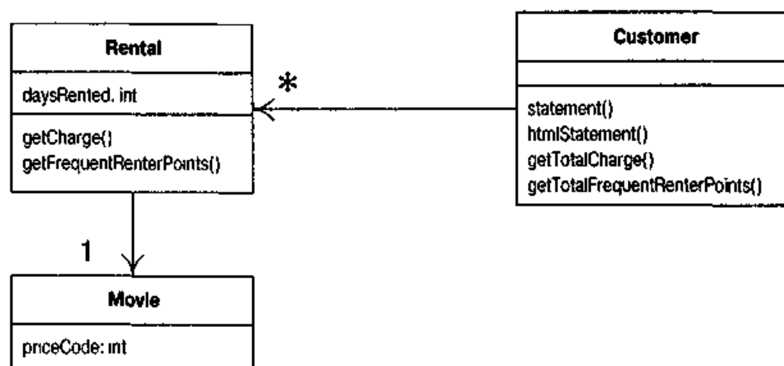


Рисунок 1.12 - Диаграмма классов до перемещения методов в Movie

```
  class Rental {
   int getFrequentRenterPoints() {
      return _movie.getFrequentRenterPoints(_daysRented)
    }
  }
  class Movie {
   int getFrequentRenterPoints(int daysRented) {
      if ((getPriceCode() == Movie.NEWRELEASE) && daysRented > 1) {
         return 2;
```

```
        } else {
        return 1;
     }
   }
```

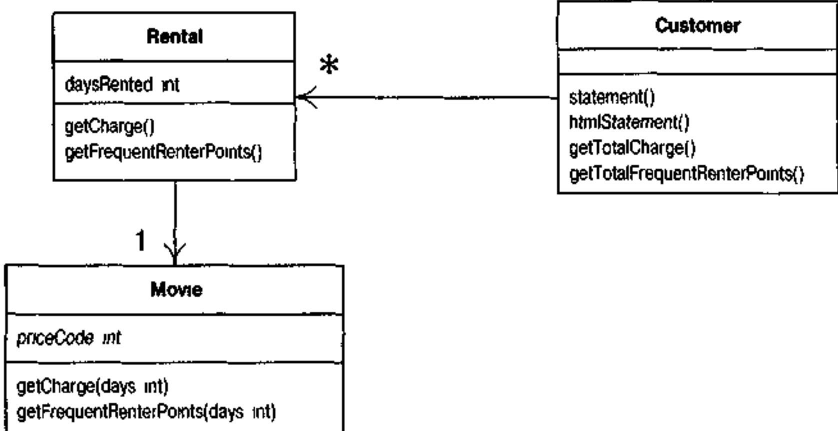


Рисунок 1.13 - Диаграмма классов после перемещения методов в Movie

Наконец-то... наследование

У нас есть несколько типов фильмов, каждый из которых по-своему отвечает на один и тот же вопрос. Похоже, здесь найдется работа для подклассов. Можно завести три подкласса Movie, в каждом из которых будет своя версия метода начисления платы (рисунок 1.14).

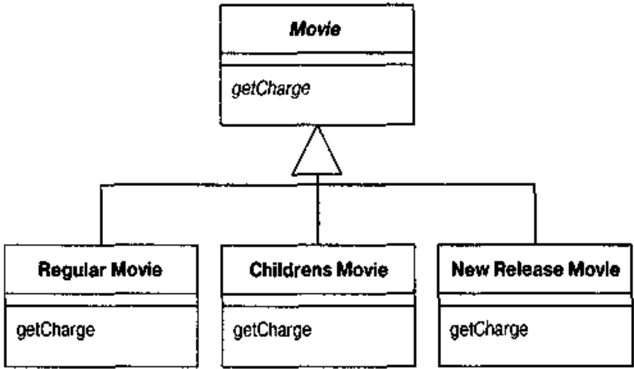


Рисунок 1.14 - Применение наследования для Movie

Такой прием позволяет заменить оператор switch полиморфизмом. К сожалению, у этого решения есть маленький недостаток - оно не работает. Фильм за время своего существования может изменить тип, объект же, пока он жив, изменить свой класс не может. Однако выход есть - паттерн «Состояние» (State pattern [Gang of Four]). При этом классы приобретают следующий вид (рисунок 1.15):

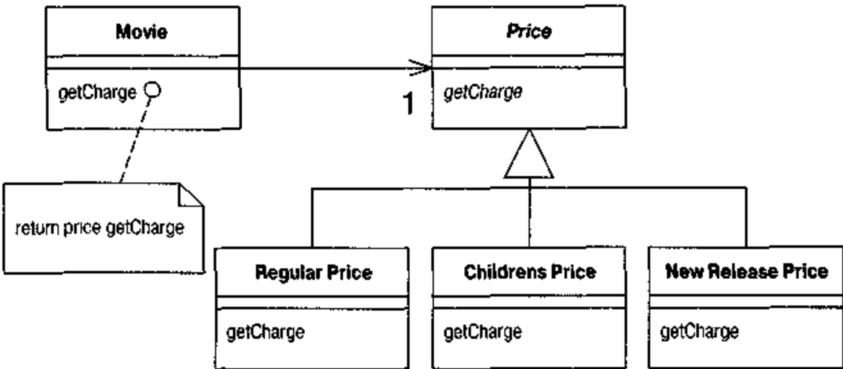


Рисунок 1.15 - Использование паттерна «состояние» с Movie

Добавляя уровень косвенности, можно порождать подклассы Price и изменять Price в случае необходимости.

У тех, кто знаком с паттернами «банды четырех», может возникнуть вопрос: «Что это - состояние или стратегия?» Представляет ли класс Price алгоритм расчета цены (и тогда я предпочел бы назвать его Pricer или PricingStrategy) или состояние фильма (Star Trek X - новинка проката). На данном этапе выбор паттерна (и имени) отражает способ, которым вы хотите представлять себе структуру. В настоящий момент я представляю это себе как состояние фильма. Если позднее я решу, что стратегия лучше передает мои намерения, я произведу рефакторинг и поменяю имена.

Для ввода схемы состояний я использую три операции рефакторинга. Сначала я перемещу поведение кода, зависящего от типа, в паттерн «Состояние» с помощью «Замены кода типа состоянием/стратегией» (Replace Type Code with State /Strategy). Затем можно применить «Перемещение метода» (Move Method), чтобы перенести оператор switch в класс Price. Наконец, с помощью «Замены условного оператора полиморфизмом» (Replace Conditional with Polymorphism) я исключу оператор switch.

Начну с «Замены кода типа состоянием/стратегией» (Replace Type Code with State/Strategy). На этом первом шаге к коду типа следует применить «Самоинкапсуляцию поля» (Self Encapsulate Field), чтобы гарантировать выполнение любых действий с кодом типа через методы get и set. Поскольку код по большей части получен из других классов, то в большинстве методов уже используется метод get. Однако в конструкторах осуществляется доступ к коду цены:

```
class Movie {
 public Movie(String name, int priceCode) {
    _title = name
    _priceCode = priceCode }
 }
}
```

Вместо этого можно прибегнуть к методу set.

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

Провожу компиляцию и тестирование, проверяя, что ничего не нарушилось. Теперь добавляю новые классы. Обеспечиваю в объекте Price поведение кода, зависящее от типа, с помощью абстрактного метода в 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;
 }
}
```

В этот момент можно компилировать новые классы.

Теперь требуется изменить методы доступа класса Movie, чтобы код класса цены использовал новый класс:

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

Это означает, что надо заменить поле кода цены полем цены и изменить функции 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 можно выполнить компиляцию и тестирование, а более сложные методы так и не узнают, что мир изменился.

Затем применяем к getCharge «Перемещение метода» (Move Method):

```
class Movie {
 double getCharge(int daysRented) {
    double result = 0;
    switch (getPriceCode()) {
       case Movie.REGULAR:
          result += 2;
          if (daysRented > 2) {
             result += (daysRented - 2) * 15;
          }
          break;
```

```
        case Movie.NEW_RELEASE:
          result += daysRented * 3;
          break;

        case Movie.CHILDRENS:
          result += 15;
          if (daysRented > 3) {
            result += (daysRented - 3) * 15;
          }
          break;
      }
      return result;
    }
  }
```

Перемещение производится 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) * 15;
          }
          break;

        case Movie.NEW_RELEASE:
          result += daysRented * 3;
          break;

        case Movie.CHILDRENS:
          result += 15;
          if (daysRented > 3) {
            result += (daysRented - 3) * 15;
          }
          break;
      }
      return result;
    }
  }
```

После перемещения можно приступить к «Замене условного оператора полиморфизмом» (Replace Conditional with Polymorphism):

```
class Price {
 double getCharge(int daysRented) {
    double result = 0;
    switch (getPriceCode()) {
       case Movie.REGULAR:
          result += 2;
          if (daysRented > 2) {
             result += (daysRented - 2) * 15;
          }
          break;

       case Movie.NEW_RELEASE:
          result += daysRented * 3;
          break;

       case Movie.CHILDRENS:
          result += 15;
          if (daysRented > 3) {
             result += (daysRented - 3) * 15;
          }
          break;
    }
    return result;
 }
}
```

Это я делаю, беря по одной ветви предложения case и создавая замещающий метод. Начнем с RegularPrice:

```
class RegularPrice {
 double getCharge(int daysRented){
    double result = 2;
    if (daysRented > 2) {
       result += (daysRented - 2) * 15;
    }
    return result;
 }
}
```

В результате замещается родительское предложение case, которое я просто оставляю как есть. После компиляции и тестирования этой ветви я беру следующую и снова компилирую и тестирую. (Чтобы проверить, действительно ли выполняется код подкласса, я умышленно делаю какую-нибудь ошибку и выполняю код, убеждаясь, что тесты «слетают». Это вовсе не значит, что я параноик.)

```
class ChildrensPrice {
 double getCharge(int daysRented) {
    double result = 15;
    if (daysRented > 3) {
       result += (daysRented - 3) * 15;
    }
```

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

Проделав это со всеми ветвями, я объявляю абстрактным метод Price getCharge:

```
  class Price {
   abstract double getCharge(int daysRented);
  }
```

Такую же процедуру я выполняю для getFrequentRenterPoints:

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

Сначала я перемещаю метод в 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 объявляю метод родительского класса абстрактным. Вместо этого я создаю замещающий метод в новых версиях и оставляю определение метода в родительском классе в качестве значения по умолчанию:

```
  Class NewReleasePrice {
```

```
 int getFrequentRenterPoints(int daysRented) {
    return (daysRented > 1) ? 2:1;
 }
}
Class Price {
 int getFrequentRenterPoints(int daysRented){
    return 1;
 }
}
```

Введение паттерна «Состояние» потребовало немалого труда. Стоит ли этого достигнутый результат? Преимущество в том, что если изменить поведение Price, добавить новые цены или дополнительное поведение, зависящее от цены, то реализовать изменения будет значительно легче. Другим частям приложения ничего не известно об использовании паттерна «Состояние». Для маленького набора функций, имеющегося в данный момент, это не играет большой роли. В более сложной системе, где зависящих от цены поведений с десяток и более, разница будет велика. Все изменения проводились маленькими шагами. Писать код таким способом долго, но мне ни разу не пришлось запускать отладчик, поэтому в действительности процесс прошел весьма быстро. Мне потребовалось больше времени для того, чтобы написать эту часть книги, чем для внесения изменений в код.

Второй главный рефакторинг завершен. Теперь будет значительно проще изменить структуру классификации фильмов или изменить правила начисления оплаты и систему начисления бонусов.

На рисунках 1.16 и 1.17 показано, как паттерн «Состояние», который я применил, работает с информацией по ценам.

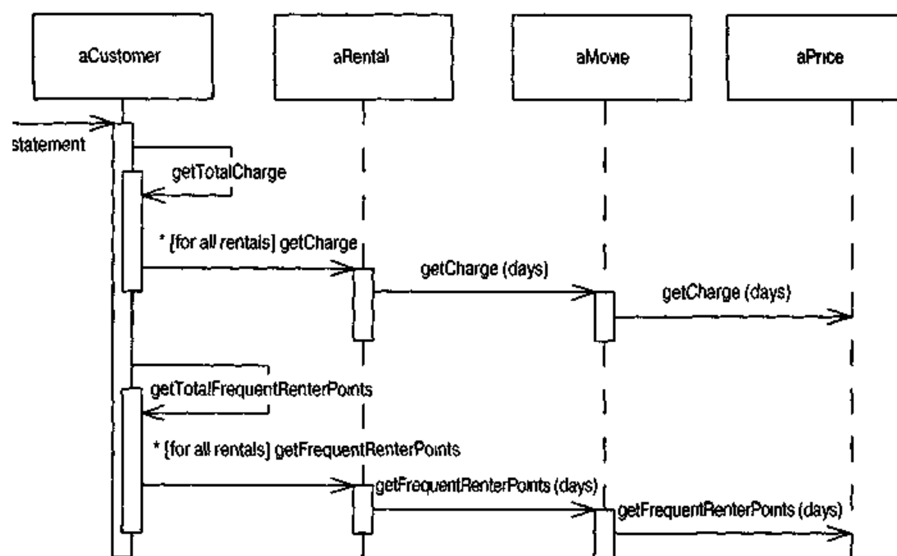


Рисунок 1.16 - Диаграмма взаимодействия с применением паттерна «Состояние»

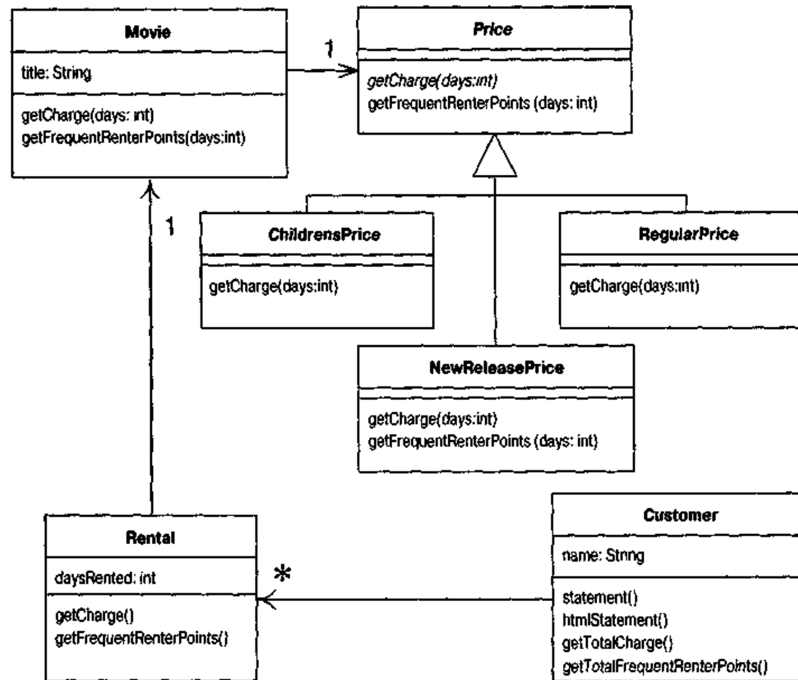


Рисунок 1.17 - Диаграмма классов после добавления паттерна «Состояние»

Заключительные размышления

Это простой пример, но надеюсь, что он дал вам почувствовать, что такое рефакторинг. Было использовано несколько видов рефакторинга, в том числе «Выделение метода» (Extract Method), «Перемещение метода» (Move Method) и «Замена условного оператора полиморфизмом» (Replace Conditional with Polymorphism). Все они приводят к более правильному распределению ответственности между классами системы и облегчают сопровождение кода. Это не похоже на код в процедурном стиле и требует некоторой привычки. Но привыкнув к такому стилю, трудно возвращаться к процедурным программам.

Самый важный урок, который должен преподать данный пример, это ритм рефакторинга: тестирование, малые изменения, тестирование, малые изменения, тестирование, малые изменения. Именно такой ритм делает рефакторинг быстрым и надежным.

Если вы дошли вместе со мной до этого места, то уже должны понимать, для чего нужен рефакторинг. Теперь можно немного заняться истоками, принципами и теорией (хотя и в ограниченном объеме).

2 ПРИНЦИПЫ РЕФАКТОРИНГА

Приведенный пример должен был дать хорошее представление о том, чем занимается рефакторинг. Теперь пора сделать шаг назад и рассмотреть основные принципы рефакторинга и некоторые проблемы, о которых следует знать при проведении рефакторинга.

Определение рефакторинга

Я всегда с некоторой настороженностью отношусь к определениям, потому что у каждого они свои, но когда пишешь книгу, приходится делать собственный выбор. В данном случае мои определения основаны на работе, проделанной группой Ральфа Джонсона (Ralph Johnson) и рядом коллег.

Первое, что следует отметить, это существование двух различных определений слова «рефакторинг» в зависимости от контекста. Это раздражает (меня - несомненно), но служит еще одним примером проблем, возникающих при работе с естественными языками.

Первое определение соответствует форме существительного.

Примечание

Рефакторинг (Refactoring) (сущ.): изменение во внутренней структуре программного обеспечения, имеющее целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого поведения.

Примеры рефакторинга можно найти в каталоге, например, «Выделение метода» (Extract Method) и «Подъем поля» (Pull Up Field). Как таковой, рефакторинг обычно представляет собой незначительные изменения в ПО, хотя один рефакторинг может повлечь за собой другие. Например, «Выделение класса» (Extract Class) обычно влечет за собой «Перемещение метода» (Move Method, 154) и «Перемещение поля» (Move Field).

Существует также определение рефакторинга как глагольной формы.

Примечание

Производить рефакторинг (Refactor) (глаг.): изменять структуру программного обеспечения, применяя ряд рефакторингов, не затрагивая его поведения.

Итак, можно в течение нескольких часов заниматься рефакторингом, и за это время произвести десяток-другой рефакторингов.

Мне иногда задают вопрос: «Заключается ли рефакторинг просто в том, что код приводится в порядок?» В некотором отношении - да, но я полагаю, что рефакторинг - это нечто большее, поскольку он предоставляет технологию приведения кода в порядок, осуществляемого в более эффективном и управляемом стиле. Я заметил, что, начав применять рефакторинг, я стал делать это значительно эффективнее, чем прежде. Это и понятно, ведь теперь я знаю, какие методы рефакторинга производить, умею применять их так, чтобы количество ошибок оказывалось минимальным, и при каждой возможности провожу тестирование.

Следует более подробно остановиться на некоторых местах в моих определениях. Во-первых, цель рефакторинга - упростить понимание и модификацию программного обеспечения. Можно выполнить много изменений в программном обеспечении, в результате которых его видимое поведение изменится незначительно или вообще не изменится. Рефакторингом будут только такие изменения, которые сделаны с целью облегчения понимания исходного кода. Противоположным примером может служить оптимизация производительности. Как и рефакторинг, оптимизация производительности обычно не изменяет поведения компонента (за исключением скорости его работы); она лишь изменяет его внутреннее устройство. Цели, однако, различны. Оптимизация производительности часто затрудняет понимание кода, но она необходима для достижения желаемого результата.

Второе обстоятельство, которое я хочу отметить, заключается в том, что рефакторинг не меняет видимого поведения программного обеспечения. Оно продолжает выполнять прежние функции. Никто - ни конечный пользователь, ни программист - не сможет сказать по внешнему виду, что что-то изменилось.

С одного конька на другой

Это второе обстоятельство связано с метафорой Кента Бека по поводу двух видов деятельности. Применение рефакторинга при разработке программного обеспечения разделяет время между двумя разными видами деятельности - вводом новых функций и изменением структуры. Добавление новых функций не должно менять структуру существующего кода: просто вводятся новые возможности. Прогресс можно оценить, добавляя тесты и добиваясь их нормальной работы. При проведении рефакторинга вы стремитесь не добавлять функции, а только улучшать структуру кода. При этом не добавляются новые тесты (если только не

обнаруживается пропущенная ранее ситуация); тесты изменяются только тогда, когда это абсолютно необходимо, чтобы проверить изменения в интерфейсе.

В процессе разработки программного обеспечения может оказаться необходимым часто переключаться между двумя видами работы. Попытавшись добавить новую функцию, можно обнаружить, что это гораздо проще сделать, если изменить структуру кода. Тогда следует на некоторое время переключиться на рефакторинг. Улучшив структуру кода, можно добавлять новую функцию. А добившись ее работы, можно заметить, что она написана способом, затрудняющим ее понимание, тогда вы снова переключаетесь и занимаетесь рефакторингом. Все это может происходить в течение десяти минут, но в каждый момент вы должны понимать, которым из видов работы заняты.

Зачем нужно проводить рефакторинг?

Я не стану провозглашать рефакторинг лекарством от всех болезней. Это не «серебряная пуля». Тем не менее это ценный инструмент - этакие «серебряные плоскогубцы», позволяющие крепко ухватить код. Рефакторинг - это инструмент, который можно и должно использовать для нескольких целей.

Рефакторинг улучшает композицию программного обеспечения

Без рефакторинга композиция программы приходит в негодность. По мере внесения в код изменений, связанных с реализацией краткосрочных целей или производимых без полного понимания организации кода, последний утрачивает свою структурированность. Разобраться в проекте, читая код, становится все труднее. Рефакторинг напоминает наведение порядка в коде. Убираются фрагменты, оказавшиеся не на своем месте. Утрата кодом структурности носит кумулятивный характер. Чем сложнее разобраться во внутреннем устройстве кода, тем труднее его сохранить и тем быстрее происходит его распад. Регулярно проводимый рефакторинг помогает сохранять форму кода.

Плохо спроектированный код обычно занимает слишком много места, часто потому, что он выполняет в нескольких местах буквально одно и то же. Поэтому важной стороной улучшения композиции является удаление дублирующегося кода. Важность этого связана с модификациями кода в будущем. Сокращение объема кода не сделает систему более быстрой, потому что изменение размера образа программы в памяти редко имеет значение. Однако объем кода играет существенную роль, когда приходится его модифицировать. Чем больше кода, тем труднее правильно его модифицировать, и разбираться приходится в его большем объеме. При изменении кода в некотором месте система ведет себя несоответственно расчетам, поскольку не модифицирован другой участок, который делает то же самое, но в несколько ином контексте. Устраняя дублирование, мы гарантируем, что в коде есть все, что нужно, и притом только в одном месте, в чем и состоит суть хорошего проектирования.

Рефакторинг облегчает понимание программного обеспечения

Во многих отношениях программирование представляет собой общение с компьютером. Программист пишет код, указывающий компьютеру, что необходимо сделать, и тот в ответ делает в точности то, что ему сказано. Со временем разрыв между тем, что требуется от компьютера, и тем, что вы ему говорите, сокращается. В таком режиме суть программирования состоит в том, чтобы точно сказать, что требуется от компьютера. Но программа адресована не только компьютеру. Пройдет некоторое время, и кому-нибудь понадобится прочесть ваш код, чтобы внести какие-то изменения. Об этом пользователе кода часто забывают, но он-то и есть главный. Станет ли кто-нибудь волноваться из-за того, что компьютеру для компиляции потребуется несколько дополнительных циклов? Зато важно, что программист может потратить неделю на модификацию кода, которая заняла бы у него лишь час, будь он в состоянии разобраться в коде.

Беда в том, что когда вы бьетесь над тем, чтобы заставить программу работать, то совсем не думаете о разработчике, который будет заниматься ею в будущем. Надо сменить ритм, чтобы внести в код изменения, облегчающие его понимание. Рефакторинг помогает сделать код более легким для чтения. При проведении рефакторинга вы берете код, который работает, но не отличается идеальной структурой. Потратив немного времени на рефакторинг, можно добиться того, что код станет лучше информировать о своей цели. В таком режиме суть программирования состоит в том, чтобы точно сказать, что вы имеете в виду.

Эти соображения навеяны не только альтруизмом. Таким будущим разработчиком часто являюсь я сам. Здесь рефакторинг имеет особую важность. Я очень ленивый программист. Моя лень проявляется, в частности, в том, что я не запоминаю деталей кода, который пишу. На самом деле, опасаясь перегрузить свою голову, я умышленно стараюсь не запоминать того, что могу найти. Я стараюсь записывать все, что в противном случае пришлось бы запомнить, в код. Благодаря этому я меньше волнуюсь из-за того, что Old Peculier[2] [Jackson] убивает клетки моего головного мозга.

У этой понятности есть и другая сторона. После рефакторинга мне легче понять незнакомый код. Глядя на незнакомый код, я пытаюсь понять, как он работает. Смотрю на одну строку, на другую и говорю себе: «Ага,

[2] Сорт пива – прим. редактора

вот что делает этот фрагмент». Занимаясь рефакторингом, я не останавливаюсь на этом мысленном замечании, а изменяю этот код так, чтобы он лучше отражал мое понимание его, а затем проверяю, правильно ли я его понял, выполняя новый код и убеждаясь в его работоспособности.

Уже с самого начала я провожу такой рефакторинг небольших деталей. По мере того как код становится более ясным, я обнаруживаю в его конструкции то, чего не замечал раньше. Если бы я не модифицировал код, то, вероятно, вообще этого не увидел бы, потому что не настолько сообразителен, чтобы представить все это зрительно в уме. Ральф Джонсон (Ralph Johnson) сравнивает эти первые шаги рефакторинга с мытьем окна, которое позволяет видеть дальше. Я считаю, что при изучении кода рефакторинг приводит меня к более высокому уровню понимания, которого я иначе бы не достиг.

Рефакторинг помогает найти ошибки

Лучшее понимание кода помогает мне выявить ошибки. Должен признаться, что в поиске ошибок я не очень искусен. Есть люди, способные прочесть большой фрагмент кода и увидеть в нем ошибки, я - нет. Однако я заметил, что при проведении рефакторинга кода я глубоко вникаю в него, пытаясь понять, что он делает, и достигнутое понимание возвращаю обратно в код. После прояснения структуры программы некоторые сделанные мной допущения становятся настолько ясными, что я не могу не увидеть ошибки.

Это напоминает мне высказывание Кента Бека, которое он часто повторяет: «Я не считаю себя замечательным программистом. Я просто хороший программист с замечательными привычками». Рефакторинг очень помогает мне писать надежный код.

Рефакторинг позволяет быстрее писать программы

В конечном счете все перечисленное сводится к одному: рефакторинг способствует ускорению разработки кода.

Кажется, что это противоречит интуиции. Когда я рассказываю о рефакторинге, всем становится ясно, что он повышает качество кода. Совершенствование конструкции, улучшение читаемости, уменьшение числа ошибок - все это повышает качество. Но разве в результате всего этого не снижается скорость разработки?

Я совершенно уверен, что хороший дизайн системы важен для быстрой разработки программного обеспечения. Действительно, весь смысл хорошего дизайна в том, чтобы сделать возможной быструю разработку. Без него можно некоторое время быстро продвигаться, но вскоре плохой дизайн становится тормозом. Время будет тратиться не на добавление новых функций, а на поиск и исправление ошибок. Модификация занимает больше времени, когда приходится разбираться в системе и искать дублирующийся код. Добавление новых функций требует большего объема кодирования, когда на исходный код наложено несколько слоев заплаток.

Хороший дизайн важен для сохранения скорости разработки программного обеспечения. Благодаря рефакторингу программы разрабатываются быстрее, т. к. он удерживает композицию системы от распада. С его помощью можно даже улучшить дизайн.

Когда следует проводить рефакторинг?

Рассказывая о рефакторинге, я часто слышу вопрос о том, когда он должен планироваться. Надо ли выделять для проведения рефакторинга две недели после каждой пары месяцев работы?

Как правило, я против откладывания рефакторинга «на потом». На мой взгляд, это не тот вид деятельности. Рефакторингом следует заниматься постоянно понемногу. Надо не решать проводить рефакторинг, а проводить его, потому что необходимо сделать что-то еще, а поможет в этом рефакторинг.

Правило трех ударов

Вот руководящий совет, который дал мне Дон Роберте (Don Roberts). Делая что-то в первый раз, вы просто это делаете. Делая что-то аналогичное во второй раз, вы морщитесь от необходимости повторения, но все-таки повторяете то же самое. Делая что-то похожее в третий раз, вы начинаете рефакторинг.

Примечание

После трех ударов начинайте рефакторинг.

Применяйте рефакторинг при добавлении новой функции

Чаще всего я начинаю рефакторинг, когда в некоторое программное обеспечение требуется добавить новую функцию. Иногда при этом причиной рефакторинга является желание лучше понять код, который надо модифицировать. Этот код мог написать кто-то другой, а мог и я сам. Всякий раз, когда приходится думать о том, что делает некий код, я задаю себе вопрос о том, не могу ли я изменить его структуру так, чтобы организация кода стала более очевидной. После этого я провожу рефакторинг кода. Отчасти это делается на тот

случай, если мне снова придется с ним работать, но в основном потому, что мне большее становится понятным, если в процессе работы я делаю код более ясным.

Еще одна причина, которая в этом случае побуждает к проведению рефакторинга, - это дизайн, не способствующий легкому добавлению новой функции. Глядя на дизайн кода, я говорю себе: «Если бы я спроектировал этот код так-то и так-то, добавить такую функцию было бы просто». В таком случае я не переживаю о прежних промахах, а исправляю их - путем проведения рефакторинга. Отчасти это делается с целью облегчения дальнейших усовершенствований, но в основном потому, что я считаю это самым быстрым способом. Рефакторинг - процесс быстрый и ровный. После рефакторинга добавление новой функции происходит значительно более гладко и занимает меньше времени.

Применяйте рефакторинг, если требуется исправить ошибку

При исправлении ошибок польза рефакторинга во многом заключается в том, что код становится более понятным. Я смотрю на код, пытаясь понять его, я произвожу рефакторинг кода, чтобы лучше понять. Часто оказывается, что такая активная работа с кодом помогает найти в нем ошибки. Можно взглянуть на это и так: если мы получаем сообщение об ошибке, то это признак необходимости рефакторинга, потому что код не был достаточно ясным и мы не смогли увидеть ошибку.

Применяйте рефакторинг при разборе кода

В некоторых организациях регулярно проводится разбор кода, а в других - нет, и напрасно. Благодаря разбору кода знания становятся достоянием всей команды разработчиков. При этом более опытные разработчики передают свои знания менее опытным. Разборы помогают большему числу людей разобраться с большим числом аспектов крупной программной системы. Они также очень важны для написания понятного кода. Мой код может казаться понятным мне, но не моей команде. Это неизбежно - очень трудно поставить себя на место того, кто не знаком с вашей работой. Разборы также дают возможность большему числу людей высказать полезные мысли. Столько хороших идей у меня и за неделю не появится. Вклад, вносимый коллегами, облегчает мне жизнь, поэтому я всегда стараюсь чаще посещать разборы.

Почему рефакторинг приносит результаты

Кент Бек

Ценность программ имеет две составляющие - то, что они могут делать для нас сегодня, и то, что они смогут делать завтра. В большинстве случаев мы сосредоточиваемся при программировании на том, что требуется от программы сегодня. Исправляя ошибку или добавляя новую функцию, мы повышаем ценность программы сегодняшнего дня путем расширения ее возможностей.

Занимаясь программированием долгое время, нельзя не заметить, что функции, выполняемые системой сегодня, отражают лишь одну сторону вопроса. Если сегодня можно выполнить ту работу, которая нужна сегодня, но так, что это не гарантирует выполнение той работы, которая понадобится завтра, то вы проиграли. Хотя, конечно, вы знаете, что вам нужно сегодня, но не вполне уверены в том, что потребуется завтра. Это может быть одно, может быть и другое, а может быть и то, что вы не могли себе вообразить.

Я знаю достаточно, чтобы выполнить сегодняшнюю задачу. Я знаю недостаточно, чтобы выполнить завтрашнюю. Но если я буду работать только на сегодняшний день, завтра я не смогу работать вообще.

Рефакторинг - один из путей решения описанной проблемы. Обнаружив, что вчерашнее решение сегодня потеряло смысл, мы его изменяем. Теперь мы можем выполнить сегодняшнюю задачу. Завтра наше сегодняшнее представление о задаче покажется наивным, поэтому мы и его изменим. Из-за чего бывает трудно работать с программами? В данный момент мне приходят в голову четыре причины:

1 Программы, трудные для чтения, трудно модифицировать.

2 Программы, в логике которых есть дублирование, трудно модифицировать.

3 Программы, которым нужны дополнительные функции, что требует изменений в работающем коде, трудно модифицировать.

4 Программы, реализующие сложную логику условных операторов, трудно модифицировать.

Итак, нам нужны программы, которые легко читать, вся логика которых задана в одном и только одном месте, модификация которых не ставит под угрозу существующие функции и которые позволяют выражать условную логику возможно более простым способом.

Рефакторинг представляет собой процесс улучшения работающей программы не путем изменения ее функций, а путем усиления в ней указанных качеств, позволяющих продолжить разработку с высокой скоростью.

Я обнаружил, что рефакторинг помогает мне разобраться в коде, написанном не мной. До того как я стал применять рефакторинг, я мог прочесть код, в какой-то мере понять его и внести предложения. Теперь, когда у

меня возникают идеи, я смотрю, нельзя ли их тут же реализовать с помощью рефакторинга. Если да, то я провожу рефакторинг. Проделав это несколько раз, я лучше понимаю, как будет выглядеть код после внесения в него предлагаемых изменений. Не надо напрягать воображение, чтобы увидеть будущий код, это можно сделать уже теперь. В результате появляются уже идеи следующего уровня, которых у меня не возникло бы, не проведи я рефакторинг.

Кроме того, рефакторинг способствует получению более конкретных результатов от разбора кода. В его процессе не только возникают новые предложения, но многие из них тут же реализуются. В результате это мероприятие дает ощущение большей успешности.

Для этой технологии группы разбора должны быть невелики. Мой опыт говорит в пользу того, чтобы над кодом совместно работали один рецензент и сам автор кода. Рецензент предлагает изменения, и они вместе с автором решают, насколько легко провести соответствующий рефакторинг. Если изменение небольшое, они модифицируют код.

При разборе больших проектов часто бывает лучше собрать несколько разных мнений в более представительной группе. При этом показ кода может быть не лучшим способом. Я предпочитаю использовать диаграммы, созданные на UML, и проходить сценарии с помощью CRC-карт (Class Responsibility Collaboration cards). Таким образом, разбор дизайна кода я провожу в группах, а разбор самого кода - с отдельными рецензентами.

Эта идея активного разбора кода доведена до своего предела в практике программирования парами (Pair Programming) в книге по экстремальному программированию [Beck, XP]. При использовании этой технологии все серьезные разработки выполняются двумя разработчиками на одной машине. В результате в процесс разработки включается непрерывный разбор кода, а также рефакторинг.

Как объяснить это своему руководителю?

Вопрос, который мне задают чаще всего, - как разговаривать о рефакторинге с руководителем? Если руководитель технически грамотен, ознакомление его с этим предметом не составит особого труда. Если руководитель действительно ориентирован на качество, то следует подчеркивать аспекты качества. В этом случае хорошим способом введения рефакторинга будет его использование в процессе разбора кода. Бесчисленные исследования показывают, что технические разборы играют важную роль для сокращения числа ошибок и обусловленного этим ускорения разработки. Последние высказывания на эту тему можно найти в любых книгах, посвященных разбору, экспертизе или технологии разработки программного обеспечения. Они должны убедить большинство руководителей в важности разборов. Отсюда один шаг до того, чтобы сделать рефакторинг способом введения в код замечаний, высказываемых при разборе.

Конечно, многие говорят, что главное для них качество, а на самом деле главное для них - выполнение графика работ. В таких случаях я даю несколько спорный совет: не говорите им ничего!

Подрывная деятельность? Не думаю. Разработчики программного обеспечения - это профессионалы. Наша работа состоит в том, чтобы создавать эффективные программы как можно быстрее. По моему опыту, рефакторинг значительно способствует быстрому созданию приложений. Если мне надо добавить новую функцию, а проект плохо согласуется с модификацией, то быстрее сначала изменить его структуру, а потом добавлять новую функцию. Если требуется исправить ошибку, то необходимо сначала понять, как работает программа, и я считаю, что быстрее всего можно сделать это с помощью рефакторинга. Руководитель, подгоняемый графиком работ, хочет, чтобы я сделал свою работу как можно быстрее; как мне это удастся - мое дело. Самый быстрый путь - рефакторинг, поэтому я и буду им заниматься.

Косвенность и рефакторинг

Кент Бек

Вычислительная техника - это дисциплина, в которой считается, что все проблемы можно решить благодаря введению одного или нескольких уровней косвенности.

Деннис Де Брюле

Учитывая одержимость разработчиков программного обеспечения косвенностью, не следует удивляться тому, что рефакторинг, как правило, вводит в программу дополнительную косвенность. Рефакторинг обычно разделяет большие объекты, как и большие методы, на несколько меньших.

Однако рефакторинг - меч обоюдоострый. Каждый раз при разделении чего-либо надвое количество объектов управления растет. При этом может также быть затруднено чтение программы, потому что один объект делегирует полномочия другому, который делегирует их третьему. Поэтому желательно минимизировать косвенность.

Но не следует спешить. Косвенность может окупиться, например, следующими способами:

1 Позволить совместно использовать логику. Например, подметод, вызываемый из разных мест, или метод родительского класса, доступный всем подклассам.

2 Изолировать изменения в коде. Допустим, я использую объект в двух разных местах и мне надо изменить его поведение в одном из этих двух случаев. Если изменить объект, это может повлиять на оба случая, поэтому я сначала создаю подкласс и пользуюсь им в том случае, где нужны изменения. В результате можно модифицировать родительский класс без риска непреднамеренного изменения во втором случае.

3 Кодировать условную логику. В объектах есть сказочный механизм - полиморфные сообщения, гибко, но ясно выражающие условную логику. Преобразовав явные условные операторы в сообщения, часто можно одновременно уменьшить дублирование, улучшить понятность и увеличить гибкость.

Вот суть игры для рефакторинга: как, сохранив текущее поведение системы, повысить ее ценность - путем повышения ее качества или снижения стоимости? Как правило, следует взглянуть на программу и найти места, где преимущества, достигаемые косвенностью, отсутствуют. Ввести необходимую косвенность, не меняя поведения системы. В результате ценность программы возрастает, потому что у нее теперь больше качеств, которые будут оценены завтра. Сравните это с тщательным предварительным проектированием. Теоретическое проектирование представляет собой попытку заложить в систему все ценные свойства еще до написания какого-либо кода. После этого код можно просто навесить на крепкий каркас. Проблема при этом в том, что можно легко ошибиться в предположениях. Применение рефакторинга исключает опасность все испортить. После него программа ведет себя так же, как и прежде. Сверх того, есть возможность добавить в код ценные качества.

Есть и вторая, более редкая разновидность этой игры. Найдите косвенность, которая не окупает себя, и уберите ее из программы. Часто это относится к переходным методам, служившим решению какой-то задачи, теперь отпавшей. Либо это может быть компонент, задуманный как совместно используемый или полиморфный, но в итоге нужный лишь в одном месте. При обнаружении паразитной косвенности следует ее удалять. И снова ценность программы повысится не благодаря присутствию в ней перечисленных выше четырех качеств, а благодаря тому, что нужно меньше косвенности, чтобы получить тот же итог от этих качеств.

Проблемы, возникающие при проведении рефакторинга

При изучении новой технологии, значительно повышающей производительность труда, бывает нелегко увидеть, что есть случаи, когда она неприменима. Обычно она изучается в особом контексте, часто являющемся отдельным проектом. Трудно увидеть, почему технология может стать менее эффективной и даже вредной. Десять лет назад такая ситуация была с объектами. Если бы меня спросили тогда, в каких случаях не стоит пользоваться объектами, ответить было бы трудно.

Не то чтобы я не считал, что применение объектов имеет свои ограничения - я достаточный циник для этого. Я просто не знал, каковы эти ограничения, хотя каковы преимущества, мне было известно.

Теперь такая же история происходит с рефакторингом. Нам известно, какие выгоды приносит рефакторинг. Нам известно, что они могут ощутимо изменить нашу работу. Но наш опыт еще недостаточно богат, чтобы увидеть, какие ограничения присущи рефакторингу.

Этот раздел короче, чем мне бы того хотелось, и, скорее, экспериментален. По мере того как о рефакторинге узнает все большее число людей, мы узнаем о нем все больше. Для читателя это должно означать, что несмотря на мою полную уверенность в необходимости применения рефакторинга ради реально достигаемых с его помощью выгод, надо следить за его ходом. Обращайте внимание на трудности, возникающие при проведении рефакторинга. Сообщайте нам о них. Больше узнав о рефакторинге, мы найдем больше решений возникающих проблем и сможем выявить те из них, которые трудно поддаются решению.

Базы данных

Одной из областей применения рефакторинга служат базы данных. Большинство деловых приложений тесно связано с поддерживающей их схемой базы данных. Это одна из причин, по которым базу данных трудно модифицировать. Другой причиной является миграция данных. Даже если система тщательно разбита по слоям, чтобы уменьшить зависимости между схемой базы данных и объектной моделью, изменение схемы базы данных вынуждает к миграции данных, что может оказаться длительной и рискованной операцией.

В случае необъектных баз данных с этой задачей можно справиться, поместив отдельный программный слой между объектной моделью и моделью базы данных. Благодаря этому можно отделить модификации двух разных моделей друг от друга. Внося изменения в одну модель, не обязательно изменять другую, надо лишь модифицировать промежуточный слой. Такой слой вносит дополнительную сложность, но дает значительную гибкость. Даже без проведения рефакторинга это очень важно в ситуациях, когда имеется несколько баз данных или сложная модель базы данных, управлять которой вы не имеете возможности.

Не обязательно начинать с создания отдельного слоя. Можно создать этот слой, заметив, что части объектной модели становятся переменчивыми. Благодаря этому достигаются наилучшие возможности для осуществления модификации.

Объектные модели одновременно облегчают задачу и усложняют ее. Некоторые объектно-ориентированные базы данных позволяют автоматически переходить от одной версии объекта к другой. Это сокращает необходимые усилия, но влечет потерю времени, связанную с осуществлением перехода. Если переход не автоматизирован, его приходится осуществлять самостоятельно, что требует больших затрат. В этом случае следует более осмотрительно изменять структуры данных классов. Можно свободно перемещать методы, но необходимо проявлять осторожность при перемещении полей. Надо применять методы для доступа к данным, чтобы создавать иллюзию их перемещения, в то время как в действительности его не происходило. При достаточной уверенности в том, где должны находиться данные, можно переместить их с помощью одной операции. Модифицироваться должны только методы доступа, что снижает риск появления ошибок.

Изменение интерфейсов

Важной особенностью объектов является то, что они позволяют изменять реализацию программного модуля независимо от изменений в интерфейсе. Можно благополучно изменить внутреннее устройство объекта, никого при этом не потревожив, но интерфейс имеет особую важность - если изменить его, может случиться всякое.

В рефакторинге беспокойство вызывает то, что во многих случаях интерфейс действительно изменяется. Такие простые вещи, как «Переименование метода» (Rename Method), целиком относятся к изменению интерфейса. Как это соотносится с бережно хранимой идеей инкапсуляции?

Поменять имя метода нетрудно, если доступен весь код, вызывающий этот метод. Даже если метод открытый, но можно добраться до всех мест, откуда он вызывается, и модифицировать их, метод можно переименовывать. Проблема возникает только тогда, когда интерфейс используется кодом, который не доступен для изменений. В таких случаях я говорю, что интерфейс опубликован (на одну ступень дальше открытого интерфейса). Если интерфейс опубликован, изменять его и просто редактировать точки вызова небезопасно. Необходима несколько более сложная технология.

При таком взгляде вопрос изменяется. Теперь задача формулируется следующим образом: как быть с рефакторингами, изменяющими опубликованные интерфейсы?

Коротко говоря, при изменении опубликованного интерфейса в процессе рефакторинга необходимо сохранять как старый интерфейс, так и новый, - по крайней мере до тех пор, пока пользователи не смогут отреагировать на модификацию. К счастью, это не очень трудно. Обычно можно сделать так, чтобы старый интерфейс продолжал действовать. Попытайтесь устроить так, чтобы старый интерфейс вызывал новый. При этом, изменив название метода, сохраните прежний. Не копируйте тело метода - дорогой дублирования кода вы пройдете прямиком к проклятию. На Java следует также воспользоваться средством пометки кода как устаревшего (необходимо объявить метод, который более не следует использовать, с модификатором deprecated). Благодаря этому обращающиеся к коду будут знать, что ситуация изменилась.

Хорошим примером этого процесса служат классы коллекций Java. Новые классы Java 2 заменяют собой те, которые предоставлялись первоначально. Однако когда появились классы Java 2, JavaSoft было приложено много усилий для обеспечения способа перехода к ним.

Защита интерфейсов обычно осуществима, но связана с усилиями. Необходимо создать и сопровождать эти дополнительные методы, по крайней мере, временно. Эти методы усложняют интерфейс, затрудняя его применение. Однако есть альтернатива: не публикуйте интерфейс. Никто не говорит здесь о полном запрете; ясно, что опубликованные интерфейсы должны быть. Если вы создаете API для внешнего использования, как это делает Sun, то тогда у вас должны быть опубликованные интерфейсы. Я говорю об этом потому, что часто вижу, как группы разработчиков злоупотребляют публикацией интерфейсов. Я знаком с тремя разработчиками, каждый из которых публиковал интерфейсы для двух других. Это порождало лишние «движения», связанные с сопровождением интерфейсов, в то время как проще было бы взять основной код и провести необходимое редактирование. К такому стилю работы тяготеют организации с гипертрофированным отношением к собственности на код. Публикация интерфейсов полезна, но не проходит безнаказанно, поэтому воздерживайтесь от нее без особой необходимости. Это может потребовать изменения правил в отношении владения кодом, чтобы люди могли изменять чужой код для поддержки изменений в интерфейсе. Часто это целесообразно делать при программировании парами.

Примечание

Не публикуйте интерфейсы раньше срока Измените политику в отношении владения кодом, чтобы облегчить рефакторинг

Есть одна особая область проблем, возникающих в связи с изменением интерфейсов в Java: добавление исключительной ситуации в предложения throw объявлений методов. Это не изменение сигнатуры, поэтому ситуация не решается с помощью делегирования. Однако компилятор не позволит осуществить компиляцию. С

данной проблемой бороться тяжело. Можно создать новый метод с новым именем, позволить вызывать его старому методу и превратить обрабатываемую исключительную ситуацию в необрабатываемую. Можно также генерировать необрабатываемую исключительную ситуацию, хотя при этом теряется возможность проверки. Если так поступить, можно предупредить тех, кто пользуется методом, что исключительная ситуация в будущем станет обрабатываемой, дав им возможность установить обработчики в своем коде. По этой причине я предпочитаю определять родительский класс исключительной ситуации для пакета в целом (например, SQLException для java.sql) и объявлять эту исключительную ситуацию только в предложениях throw опубликованных методов. Используя такой подход, возможно создать иерархию классов исключительных ситуаций и использовать только родительский класс в предложениях throw опубликованных интерфейсов, в то же время можно будет использовать его подклассы в более специфичных ситуациях посредством операций приведения и проверки типа объекта исключительной ситуации. Благодаря этому я при желании могу определять подклассы исключительных ситуаций, что не затрагивает вызывающего, которому известен только общий случай.

Изменения дизайна, вызывающие трудности при рефакторинге

Можно ли с помощью рефакторинга исправить любые недостатки проектирования или в дизайне системы могут быть заложены такие базовые решения, которые впоследствии не удастся изменить путем проведения рефакторинга? Имеющиеся в этой области данные недостаточны. Конечно, часто рефакторинг оказывается эффективным, но иногда возникают трудности. Мне известен проект, в котором, хотя и с трудом, но удалось при помощи рефакторинга привести архитектуру системы, созданной без учета защиты данных, к архитектуре с хорошей защитой.

На данном этапе мой подход состоит в том, чтобы представить себе возможный рефакторинг. Рассматривая различные варианты дизайна системы, я пытаюсь определить, насколько трудным окажется рефакторинг одного дизайна в другой. Если трудностей не видно, то я, не слишком задумываясь о выборе, останавливаюсь на самом простом дизайне, даже если он не охватывает все требования, которые могут возникнуть в дальнейшем. Однако если я не могу найти простых способов рефакторинга, то продолжаю работать над дизайном. Мой опыт показывает, что такие ситуации редки.

Когда рефакторинг не нужен?

В некоторых случаях рефакторинг вообще не нужен. Основной пример - необходимость переписать программу с нуля. Иногда имеющийся код настолько запутан, что подвергнуть его рефакторингу, конечно, можно, но проще начать все с самого начала. Такое решение принять нелегко, и я признаюсь, что не могу предложить достаточно надежные рекомендации по этому поводу.

Явный признак необходимости переписать код - его неработоспособность. Это обнаруживается только при его тестировании, когда ошибок оказывается так много, что сделать код устойчивым не удается. Помните, что перед началом рефакторинга код должен выполняться в основном корректно.

Компромиссное решение состоит в создании компонентов с сильной инкапсуляцией путем рефакторинга значительной части программного обеспечения. После этого для каждого компонента в отдельности можно принять решение относительно изменения его структуры при помощи рефакторинга или воссоздания заново. Это многообещающий подход, но имеющихся у меня данных недостаточно, чтобы сформулировать четкие правила его осуществления. Когда основная система является унаследованной, такой подход, несомненно, становится привлекательным .

Другой случай, когда следует воздерживаться от рефакторинга, это близость даты завершения проекта. Рост производительности, достигаемый благодаря рефакторингу, проявит себя слишком поздно - после истечения срока. Правильна в этом смысле точка зрения Уорда Каннингема (Ward Cunningham). Незавершенный рефакторинг он сравнивает с залезанием в долги. Большинству компаний для нормальной работы нужны кредиты. Однако вместе с долгами появляются и проценты, то есть дополнительная стоимость обслуживания и расширения, обусловленная чрезмерной сложностью кода. Выплату каких-то процентов можно вытерпеть, но если платежи слишком велики, вы разоритесь. Важно управлять своими долгами, выплачивая их часть посредством рефакторинга.

Однако приближение срока окончания работ - единственный случай, когда можно отложить рефакторинг, ссылаясь на недостаток времени. Опыт работы над несколькими проектами показывает, что проведение рефакторинга приводит к росту производительности труда. Нехватка времени обычно сигнализирует о необходимости рефакторинга.

Рефакторинг и проектирование

Рефакторинг играет особую роль в качестве дополнения к проектированию. Когда я только начинал учиться программированию, я просто писал программу и как-то доводил ее до конца. Со временем мне стало ясно, что если заранее подумать об архитектуре программы, то можно избежать последующей дорогостоящей

переработки. Я все более привыкал к этому стилю предварительного проектирования. Многие считают, что проектирование важнее всего, а программирование представляет собой механический процесс. Аналогией проекта служит технический чертеж, а аналогией кода - изготовление узла. Но программа весьма отличается от физического механизма. Она значительно более податлива и целиком связана с обдумыванием. Как говорит Элистер Кокберн (Alistair Cockburn): «При наличии готового дизайна я думаю очень быстро, но в моем мышлении полно пробелов».

Существует утверждение, что рефакторинг может быть альтернативой предварительному проектированию. В таком сценарии проектирование вообще отсутствует. Первое решение, пришедшее в голову, воплощается в коде, доводится до рабочего состояния, а потом обретает требуемую форму с помощью рефакторинга. Такой подход фактически может действовать. Мне встречались люди, которые так работают и получают в итоге систему с очень хорошей архитектурой. Тех, кто поддерживает «экстремальное программирование» [Beck, XP], часто изображают пропагандистами такого подхода.

Подход, ограничивающийся только рефакторингом, применим, но не является самым эффективным. Даже «экстремальные» программисты сначала разрабатывают некую архитектуру будущей системы. Они пробуют разные идеи с помощью CRC-карт или чего-либо подобного, пока не получат внушающего доверия первоначального решения. Только после первого более или менее удачного «выстрела» приступают к кодированию, а затем к рефакторингу. Смысл в том, что при использовании рефакторинга изменяется роль предварительного проектирования. Если не рассчитывать на рефакторинг, то ощущается необходимость как можно лучше провести предварительное проектирование. Возникает чувство, что любые изменения проекта в будущем, если они потребуются, окажутся слишком дорогостоящими. Поэтому в предварительное проектирование вкладывается больше времени и усилий - во избежание таких изменений впоследствии.

С применением рефакторинга акценты смещаются. Предварительное проектирование сохраняется, но теперь оно не имеет целью найти единственно правильное решение. Все, что от него требуется, - это найти приемлемое решение. По мере реализации решения, с углублением понимания задачи становится ясно, что наилучшее решение отличается от того, которое было принято первоначально. Но в этом нет ничего страшного, если в процессе участвует рефакторинг, потому что модификация не обходится слишком дорого.

Важным следствием такого смещения акцентов является большее стремление к простоте проекта. До введения рефакторинга в свою работу я всегда искал гибкие решения. Для каждого технического требования я рассматривал возможности его изменения в течение срока жизни системы. Поскольку изменения в проекте были дорогостоящими, я старался создать проект, способный выдержать изменения, которые я мог предвидеть. Недостаток гибких решений в том, что за гибкость приходится платить. Гибкие решения сложнее обычных. Создаваемые по ним программы в целом труднее сопровождать, хотя и легче перенацеливать в том направлении, которое предполагалось изначально. И даже такие решения не избавляют от необходимости разбираться, как модифицировать проект. Для одной-двух функций это сделать не очень трудно, но изменения происходят по всей системе. Если предусматривать гибкость во всех этих местах, то вся система становится значительно сложнее и дороже в сопровождении. Весьма разочаровывает, конечно, то, что вся эта гибкость и не нужна. Потребуется лишь какая-то часть ее, но невозможно заранее сказать какая. Чтобы достичь гибкости, приходится вводить ее гораздо больше, чем требуется в действительности.

Рефакторинг предоставляет другой подход к рискам модификации. Возможные изменения все равно надо пытаться предвидеть, как и рассматривать гибкие решения. Но вместо реализации этих гибких решений следует задаться вопросом: «Насколько сложно будет с помощью рефакторинга преобразовать обычное решение в гибкое?» Если, как чаще всего случается, ответ будет «весьма несложно», то надо просто реализовать обычное решение.

Рефакторинг позволяет создавать более простые проекты, не жертвуя гибкостью, благодаря чему процесс проектирования становится более легким и менее напряженным. Научившись в целом распознавать то, что легко поддается рефакторингу, о гибкости решений даже перестаешь задумываться. Появляется уверенность в возможности применения рефакторинга, когда это понадобится. Создаются самые простые решения, которые могут работать, а гибкие и сложные решения по большей части не потребуются.

Чтобы ничего не создать, требуется некоторое время

Рон Джеффрис

Система полной выплаты компенсации Chrysler Comprehensive Compensation работала слишком медленно. Хотя разработка еще не была завершена, это стало нас беспокоить, потому что стало замедляться выполнение тестов.

Кент Бек, Мартин Фаулер и я решили исправить ситуацию. В ожидании нашей встречи я, будучи хорошо знаком с системой, размышлял о том, что же может ее тормозить. Я подумывал о нескольких причинах и поговорил с людьми о тех изменениях, которые могли потребоваться. Нам пришло в голову несколько хороших идей относительно возможности ускорить систему.

После этого мы измерили производительность с помощью профайлера Кента. Ни одна из предполагаемых мною причин, как оказалось, не имела отношения к проблеме. Мы обнаружили, что

половину времени система тратила на создание экземпляров классов дат. Еще интереснее, что все эти объекты содержали одни и те же данные.

Изучив логику создания дат, мы нашли некоторые возможности оптимизировать процесс их создания. Везде использовалось преобразование строк, даже если не было ввода внешних данных. Делалось это просто для удобства ввода кода. Похоже, что это можно было оптимизировать.

Затем мы взглянули, как используются эти даты. Оказалось, что значительная часть их участвовала в создании диапазонов дат - объектов, содержащих дату «с» и дату «по». Еще немного осмотревшись, мы поняли, что большинство этих диапазонов пусты!

При работе с диапазонами дат нами было условлено, что любой диапазон, конечная дата которого предшествует начальной, пуст. Это удобное соглашение, хорошо согласующееся с тем, как работает класс. Начав использовать данное соглашение, мы вскоре поняли, что код, создающий диапазоны дат, начинающиеся после своего конца, непонятен, в связи с чем мы выделили эту функцию в фабричный метод (Gang of Four "Factory method"), создающий пустые диапазоны дат.

Это изменение должно было сделать код более четким, но мы были вознаграждены за него неожиданным образом. Мы создали пустой диапазон в виде константы и сделали так, чтобы фабричный метод не создавал каждый раз объект заново, а возвращал эту константу. После такой модификации скорость системы удвоилась, чего было достаточно для приемлемого выполнения тестов. Все это отняло у нас примерно пять минут.

О том, что может быть плохо в хорошо известном нам коде, мы размышляли со многими участниками проекта (Кент и Мартин отрицают свое участие в этих обсуждениях). Мы даже прикинули некоторые возможные усовершенствования, не производя предварительных измерений.

Мы были полностью неправы. Если не считать интересной беседы, пользы не было никакой.

Вывод из этого следующий. Даже если вы точно знаете, как работает система, не занимайтесь гаданием, а проведите замеры. Полученная информация в девяти случаях из десяти покажет, что ваши догадки были ошибочны!

Рефакторинг и производительность

С рефакторингом обычно связан вопрос о его влиянии на производительность программы. С целью облегчить понимание работы программы часто осуществляется модификация, приводящая к замедлению выполнения программы. Это важный момент. Я не принадлежу к той школе, которая пренебрегает производительностью в пользу чистоты проекта или в надежде на рост мощности аппаратной части. Программное обеспечение отвергалось как слишком медленное, а более быстрые машины устанавливают свои правила игры. Рефакторинг, несомненно, заставляет программу выполняться медленнее, но при этом делает ее более податливой для настройки производительности. Секрет создания быстрых программ, если только они не предназначены для работы в жестком режиме реального времени, состоит в том, чтобы сначала написать программу, которую можно настраивать, а затем настроить ее так, чтобы достичь приемлемой скорости.

Мне известны три подхода к написанию быстрых программ. Наиболее трудный из них связан с ресурсами времени и часто применяется в системах с жесткими требованиями к выполнению в режиме реального времени. В этой ситуации при декомпозиции проекта каждому компоненту выделяется бюджет ресурсов - по времени и памяти. Компонент не должен выйти за рамки своего бюджета, хотя разрешен механизм обмена временными ресурсами. Такой механизм жестко сосредоточен на соблюдении времени выполнения. Это важно в таких системах, как, например, кардиостимуляторы, в которых данные, полученные с опозданием, всегда ошибочны. Данная технология избыточна в системах другого типа, например в корпоративных информационных системах, с которыми я обычно работаю.

Второй подход предполагает постоянное внимание. В этом случае каждый программист в любой момент времени делает все от него зависящее, чтобы поддерживать высокую производительность программы. Это распространенный и интуитивно привлекательный подход, однако он не так хорош на деле. Модификация, повышающая производительность, обычно затрудняет работу с программой. Это замедляет создание программы. На это можно было бы пойти, если бы в результате получалось более быстрое программное обеспечение, но обычно этого не происходит. Повышающие скорость усовершенствования разбросаны по всей программе, и каждое из них касается только узкой функции, выполняемой программой.

С производительностью связано то интересное обстоятельство, что при анализе большинства программ обнаруживается, что большая часть времени расходуется небольшой частью кода. Если в равной мере оптимизировать весь код, то окажется, что 90% оптимизации произведено впустую, потому что оптимизировался код, который выполняется не слишком часто. Время, ушедшее на ускорение программы, и время, потерянное из-за ее непонятности - все это израсходовано напрасно.

Третий подход к повышению производительности программы основан как раз на этой статистике. Он предполагает создание программы с достаточным разложением ее на компоненты без оглядки на достигаемую производительность вплоть до этапа оптимизации производительности, который обычно наступает на довольно поздней стадии разработки и на котором осуществляется особая процедура настройки программы.

Начинается все с запуска программы под профайлером, контролирующим программу и сообщающим, где расходуются время и память. Благодаря этому можно обнаружить тот небольшой участок программы, в котором находятся узкие места производительности. На этих узких местах сосредоточиваются усилия, и осуществляется та же самая оптимизация, которая была бы применена при подходе с постоянным вниманием. Но благодаря тому, что внимание сосредоточено на выявленных узких местах, удается достичь больших результатов при значительно меньших затратах труда. Но даже в этой ситуации необходима бдительность. Как и при проведении рефакторинга, изменения следует вносить небольшими порциями, каждый раз компилируя, тестируя и запуская профайлер. Если производительность не увеличилась, изменениям дается обратный ход. Процесс поиска и ликвидации узких мест продолжается до достижения производительности, которая удовлетворяет пользователей. Мак-Коннелл [McConnell] подробно рассказывает об этой технологии.

Хорошее разделение программы на компоненты способствует оптимизации такого рода в двух отношениях. Во-первых, благодаря ему появляется время, которое можно потратить на оптимизацию. Имея хорошо структурированный код, можно быстрее добавлять новые функции и выиграть время для того, чтобы заняться производительностью. (Профилирование гарантирует, что это время не будет потрачено зря.) Во-вторых, хорошо структурированная программа обеспечивает более высокое разрешение для анализа производительности. Профайлер указывает на более мелкие фрагменты кода, которые легче настроить. Благодаря большей понятности кода легче осуществить выбор возможных вариантов и разобраться в том, какого рода настройка может оказаться действенной.

Я пришел к выводу, что рефакторинг позволяет мне писать программы быстрее. На некоторое время он делает программы более медленными, но облегчает настройку программ на этапе оптимизации. В конечном счете достигается большой выигрыш.

Каковы истоки рефакторинга?

Мне не удалось проследить, когда именно появился термин рефакторинг, Хорошие программисты, конечно, всегда хотя бы какое-то время посвящают приведению своего кода в порядок. Они занимаются этим, поскольку поняли, что аккуратный код проще модифицировать, чем сложный и запутанный, а хорошим программистам известно, что сразу написать хороший код удается редко.

Рефакторинг идет еще дальше. В этой книге рефакторинг пропагандируется как ведущий элемент процесса разработки программного обеспечения в целом. Первыми, кто понял важность рефакторинга, были Уорд Каннигем и Кент Бек, с 1980-х годов работавшие со Smalltalk. Среда Smalltalk уже тогда была весьма открыта для рефакторинга. Она очень динамична и позволяет быстро писать функционально богатые программы. У Smalltalk очень короткий цикл «компиляция-компоновка-выполнение», благодаря чему можно быстро модифицировать программы. Этот язык также является объектно-ориентированным, предоставляя мощные средства для уменьшения влияния, оказываемого изменениями, скрываемыми за четко определенными интерфейсами. Уорд и Кент потрудились над созданием технологии разработки программного обеспечения, приспособленной для применения в такого рода среде. (Сейчас Кент называет такой стиль «экстремальным программированием» (Extreme Programming) [Beck, XP].) Они поняли значение рефакторинга для повышения производительности своего труда и с тех пор применяют ее в сложных программных проектах, а также совершенствуют саму технологию.

Идеи Уорда и Кента всегда оказывали сильное влияние на сообщество Smalltalk, и понятие рефакторинга стало важным элементом культуры Smalltalk. Другая крупная фигура в сообществе Smalltalk - Ральф Джонсон, профессор Университета штата Иллинойс в Урбана-Шампань, известный как член «банды четырех». К числу областей, в которых сосредоточены наибольшие интересы Ральфа, относится создание шаблонов (frameworks) разработки программного обеспечения. Он исследовал вопрос о применении рефакторинга для создания эффективной и гибкой среды разработки.

Билл Апдайк был одним из докторантов Ральфа. Он проявляет особый интерес к шаблонам. Он заметил потенциальную ценность рефакторинга и обратил внимание, что ее применение совсем не ограничивается рамками Smalltalk. Он имел опыт разработки программного обеспечения телефонных станций, для которого характерны нарастание сложности со временем и трудность модификации. Докторская работа Билла рассматривала рефакторинг с точки зрения разработчика инструментальных средств. Билл изучил виды рефакторинга, которые могли оказаться полезными при разработке шаблонов C++, и исследовал методы рефакторинга, которые должны сохранять семантику, доказательства сохранения ими семантики и возможности реализации этих идей в инструментальных средствах. Докторская диссертация Билла [Opdyke] является на сегодняшний день самой существенной работой по рефакторингу. Он также написал для этой книги главу 13.

Помню встречу с Биллом на конференции OOPSLA в 1992 году. Мы сидели в кафе и обсуждали некоторые стороны моей работы по созданию концептуальной основы в здравоохранении. Билл рассказал мне о своих исследованиях, и я тогда подумал, что они интересны, но не представляют большого значения. Как я был неправ!

Джон Брант и Дон Роберте значительно продвинули вперед идеи инструментальных средств рефакторинга, создав Refactoring Browser, инструмент для рефакторинга в Smalltalk. Их участие в этой книге представлено главой 14, глубже описывающей инструменты рефакторинга.

А что же я? Я всегда имел склонность к хорошему коду, но никогда не думал, что это так важно. Затем я стал работать в одном проекте с Кентом и увидел, как он применяет рефакторинг. Я понял, какое влияние он оказывает на производительность труда и качество результатов. Этот опыт убедил меня в том, что рефакторинг представляет собой очень важную технологию. Однако меня разочаровало отсутствие книг, которые можно было бы дать работающему программисту, а ни один из отмеченных выше экспертов не собирался писать такую книгу. В результате это пришлось сделать мне с их помощью.

Оптимизация системы зарплаты

Рич Гарзанити

Мы уже долгое время разрабатывали систему полной выплаты компенсации Chrysler Comprehensive Compensation, когда стали переводить ее на GemStone. Естественно, в результате мы обнаружили, что программа работает недостаточно быстро. Мы пригласили Джима Хонгса (Jim Haungs), большого специалиста, помочь нам в оптимизации системы.

Проведя некоторое время с командой и разобравшись, как работает система, Джим воспользовался ProfMonitor производства GemStone и написал профилирующий инструмент, который подключил к нашим функциональным тестам. Этот инструмент показывал количество создаваемых объектов и место их создания.

К нашему удивлению, самым зловредным оказалось создание строк. Рекорд поставило многократное создание строк из 12 000 байт. При этом возникали особые проблемы, потому что обычные средства уборки мусора GemStone с такими большими строками не справлялись. Из-за большого размера GemStone выгружал строки на диск, как только они создавались. Оказалось, что эти строки создавались нашей системой ввода/вывода, причем сразу по три для каждой выходной записи!

Для начала мы стали буферизовать 12 000-байтные строки, что в основном решило проблему. Затем мы изменили среду так, чтобы запись велась прямо в файловый поток, в результате чего избежали создания хотя бы одной строки.

После того как длинные строки перестали нам мешать, профайлер Джима нашел аналогичные проблемы с маленькими строками - 800 байт, 500 байт и т. д. Преобразование их для использования с файловыми потоками решило и эти проблемы.

С помощью таких приемов мы уверенно повышали производительность системы. Во время разработки было похоже, что для расчета зарплаты потребуется более 1000 часов. Когда мы были действительно готовы к работе, фактически потребовалось 40 часов. Через месяц продолжительность была снижена до 18 часов, а после ввода в эксплуатацию - до 12. После года эксплуатации и усовершенствования системы для новой группы служащих время прогона было сокращено до 9 часов.

Самое крупное усовершенствование связано с запуском программы в нескольких потоках на многопроцессорной машине. При разработке системы применение потоков не предполагалось, но благодаря ее хорошей структурированности мы модифицировали ее для использования потоков за три дня. В настоящее время расчет зарплаты выполняется за пару часов.

До того как Джим обеспечил нас инструментом измерения фактических показателей работы системы, у нас было немало идей по поводу причин неудовлетворительной работы системы. Но время для реализации наших хороших идей еще не наступило. Реальные замеры указали другое направление работы и имели значительно большее значение.

3 КОД С ДУШКОМ

Авторы: Кент Бек и Мартин Фаулер

Если что-то плохо пахнет, это что-то надо поменять.

- Мнение бабушки Бек, высказанное при обсуждении проблем детского воспитания

К этому моменту у вас должно быть хорошее представление о том, как действует рефакторинг. Но если вы знаете «как», это не значит, что вы знаете «когда». Решение о том, когда начать рефакторинг и когда остановить его, так же важно, как умение управлять механизмом рефакторинга.

И тут возникает затруднительное положение. Легко объяснить, как удалить переменную экземпляра или создать иерархию - это простые вопросы; попытка же показать, когда это следует делать, не столь тривиальна. Вместо того чтобы взывать к неким туманным представлениям об эстетике программирования (как, честно говоря, часто поступаем мы, консультанты), я попытался подвести под это более прочную основу.

Я размышлял над этим сложным вопросом, когда встретился в Цюрихе с Беком. Возможно, он тогда находился под впечатлением ароматов, исходящих от своей новорожденной дочки, потому что выразил представление о том, когда проводить рефакторинг, на языке запахов. Вы скажете: «Запах? И что, это лучше туманных представлений об эстетике?» Да, лучше. Мы просмотрели очень много кода, написанного для проектов, степень успешности которых охватывалась широким диапазоном - от весьма удачных до полудохлых. При этом мы научились искать в коде определенные структуры, которые предполагают возможность рефакторинга (иногда просто вопиют о ней). («Мы» в этой главе отражает тот факт, что она написана Кентом и мной совместно. Определить действительного автора можно так: смешные шутки принадлежат мне, а все остальные - ему.)

Чего мы не будем пытаться сделать, так это дать точные критерии своевременности рефакторинга. Наш опыт показывает, что никакие системы показателей не могут соперничать с человеческой интуицией, основанной на информации. Мы только приведем симптомы неприятностей, устранимых путем рефакторинга. У вас должно развиться собственное чувство того, какое количество следует считать «чрезмерным» для атрибутов класса или строк кода метода.

Эта глава и таблица «Источники неприятных запахов» в конце книги должны помочь, когда неясно, какие методы рефакторинга применять. Прочтите эту главу (или пробегите глазами таблицу) и попробуйте установить, какой запах вы чувствуете, а затем обратитесь к предлагаемым нами методам рефакторинга и посмотрите, не помогут ли они вам. Возможно, запахи не совпадут полностью, но надо надеяться, что общее направление будет выбрано правильно.

Дублирование кода

Парад дурных запахов открывает дублирующийся код. Увидев одинаковые кодовые структуры в нескольких местах, можно быть уверенным, что если удастся их объединить, 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 надо лишь применить «Выделение метода» (Extract Method) и вызывать код созданного метода из обеих точек.

Другая задача с дублированием часто встречается, когда одно и то же выражение есть в двух подклассах, находящихся на одном уровне. Устранить такое дублирование можно с помощью «Выделения метода» (Extract Method) для обоих классов с последующим «Подъемом поля» (Pull Up Field). Если код похож, но не совпадает полностью, нужно применить «Выделение метода» (Extract Method) для отделения совпадающих фрагментов от различающихся. После этого может оказаться возможным применить «Формирование шаблона метода» (Form Template Method). Если оба метода делают одно и то же с помощью разных алгоритмов, можно выбрать более четкий из этих алгоритмов и применить «Замещение алгоритма» (Substitute Algorithm).

Если дублирующийся код находится в двух разных классах, попробуйте применить «Выделение класса» (Extract Class) в одном классе, а затем использовать новый компонент в другом. Бывает, что в действительности метод должен принадлежать только одному из классов и вызываться из другого класса либо метод должен принадлежать третьему классу, на который будут ссылаться оба первоначальных. Необходимо решить, где оправдано присутствие этого метода, и обеспечить, чтобы он находился там и нигде более.

Длинный метод

Программы, использующие объекты, живут долго и счастливо, когда методы этих объектов короткие. Программистам, не имеющим опыта работы с объектами, часто кажется, что никаких вычислений не происходит, а программы состоят из нескончаемой цепочки делегирования действий. Однако, тесно общаясь с

такой программой на протяжении нескольких лет, вы начинаете понимать, какую ценность представляют собой все эти маленькие методы. Все выгоды, которые дает косвенность - понятность, совместное использование и выбор, - поддерживаются маленькими методами.

Уже на заре программирования стало ясно, что чем длиннее процедура, тем труднее понять, как она работает. В старых языках программирования вызов процедур был связан с накладными расходами, которые удерживали от применения маленьких методов. Современные объектно-ориентированные языки в значительной мере устранили издержки вызовов внутри процесса. Однако издержки сохраняются для того, кто читает код, поскольку приходится переключать контекст, чтобы увидеть, чем занимается процедура. Среда разработки, позволяющая видеть одновременно два метода, помогает устранить этот шаг, но главное, что способствует пониманию работы маленьких методов, это толковое присвоение им имен. Если правильно выбрать имя метода, нет необходимости изучать его тело.

В итоге мы приходим к тому, что следует активнее применять декомпозицию методов. Мы придерживаемся эвристического правила, гласящего, что если ощущается необходимость что-то прокомментировать, надо написать метод. В таком методе содержится код, который требовал комментариев, но его название отражает назначение кода, а не то, как он решает свою задачу. Такая процедура может применяться к группе строк или всего лишь к одной строке кода. Мы прибегаем к ней даже тогда, когда обращение к коду длиннее, чем код, который им замещается, при условии, что имя метода разъясняет назначение кода. Главным здесь является не длина метода, а семантическое расстояние между тем, что делает метод, и тем, как он это делает.

В 99% случаев, чтобы укоротить метод, требуется лишь «Выделение метода» (Extract Method). Найдите те части метода, которые кажутся согласованными друг с другом, и образуйте новый метод.

Когда в методе есть масса параметров и временных переменных, это мешает выделению нового метода. При попытке «Выделения метода» (Extract Method) в итоге приходится передавать столько параметров и временных переменных в качестве параметров, что результат оказывается ничуть не проще для чтения, чем оригинал. Устранить временные переменные можно с помощью «Замены временной переменной вызовом метода» (Replace Temp with Query). Длинные списки параметров можно сократить с помощью приемов «Введение граничного объекта» (Introduce Parametr Object) и «Сохранение всего объекта» (Preserve Whole Object).

Если даже после этого остается слишком много временных переменных и параметров, приходится выдвигать тяжелую артиллерию - необходима «Замена метода объектом метода» (Replace Method with Method Object).

Как определить те участки кода, которые должны быть выделены в отдельные методы? Хороший способ - поискать комментарии: они часто указывают на такого рода семантическое расстояние. Блок кода с комментариями говорит о том, что его действие можно заменить методом, имя которого основывается на комментарии. Даже одну строку имеет смысл выделить в метод, если она нуждается в разъяснениях.

Условные операторы и циклы тоже свидетельствуют о возможности выделения. Для работы с условными выражениями подходит «Декомпозиция условных операторов» (Decompose Conditional). Если это цикл, выделите его и содержащийся в нем код в отдельный метод.

Большой класс

Когда класс пытается выполнять слишком много работы, это часто проявляется в чрезмерном количестве имеющихся у него атрибутов. А если класс имеет слишком много атрибутов, недалеко и до дублирования кода.

Можно применить «Выделение класса» (Extract Class), чтобы связать некоторое количество атрибутов. Выбирайте для компонента атрибуты так, чтобы они имели смысл для каждого из них. Например, «depositAmount» (сумма задатка) и «depositCurrency» (валюта задатка) вполне могут принадлежать одному компоненту. Обычно одинаковые префиксы или суффиксы у некоторого подмножества переменных в классе наводят на мысль о создании компонента. Если разумно создание компонента как подкласса, то более простым оказывается «Выделение подкласса» (Extract Subclass).

Иногда класс не использует постоянно все свои переменные экземпляра. В таком случае оказывается возможным применить «Выделение класса» (Extract Class) или «Выделение подкласса» (Extract Subclass) несколько раз.

Как и класс с чрезмерным количеством атрибутов, класс, содержащий слишком много кода, создает питательную среду для повторяющегося кода, хаоса и гибели. Простейшее решение (мы уже говорили, что предпочитаем простейшие решения?) - устранить избыточность в самом классе. Пять методов по сотне строк в длину иногда можно заменить пятью методами по десять строк плюс еще десять двухстрочных методов, выделенных из оригинала.

Как и для класса с кучей атрибутов, обычное решение для класса с чрезмерным объемом кода состоит в том, чтобы применить «Выделение класса» (Extract Class) или «Выделение подкласса» (Extract Subclass). Полезно установить, как клиенты используют класс, и применить «Выделение интерфейса» (Extract Interface) для каждого из этих вариантов. В результате может выясниться, как расчленить класс еще далее.

Если большой класс является классом GUI, может потребоваться переместить его данные и поведение в отдельный объект предметной области. При этом может оказаться необходимым хранить копии некоторых данных в,двух местах и обеспечить их согласованность. «Дублирование видимых данных» (Duplicate Observed Data) предлагает путь, которым можно это осуществить. В данном случае, особенно при использовании старых компонентов Abstract Windows Toolkit (AWT), можно в последующем удалить класс GUI и заменить его компонентами Swing.

Длинный список параметров

Когда-то при обучении программированию рекомендовали все необходимые подпрограмме данные передавать в виде параметров. Это можно было понять, потому что альтернативой были глобальные переменные, а глобальные переменные пагубны и мучительны. Благодаря объектам ситуация изменилась, т. к. если какие-то данные отсутствуют, всегда можно попросить их у другого объекта. Поэтому, работая с объектами, следует передавать не все, что требуется методу, а столько, чтобы метод мог добраться до всех необходимых ему данных. Значительная часть того, что необходимо методу, есть в классе, которому он принадлежит. В объектно-ориентированных программах списки параметров обычно гораздо короче, чем в традиционных программах.

И это хорошо, потому что в длинных списках параметров трудно разбираться, они становятся противоречивыми и сложными в использовании, а также потому, что их приходится вечно изменять по мере того, как возникает необходимость в новых данных. Если передавать объекты, то изменений требуется мало, потому что для получения новых данных, скорее всего, хватит пары запросов.

«Замена параметра вызовом метода» (Replace Parameter with Method) уместна, когда можно получить данные в одном параметре путем вызова метода объекта, который уже известен. Этот объект может быть полем или другим параметром. «Сохранение всего объекта» (Preserve Whole Object) позволяет взять группу данных, полученных от объекта, и заменить их самим объектом. Если есть несколько элементов данных без логического объекта, выберите «Введение граничного объекта» (Introduce Parameter Object).

Есть важное исключение, когда такие изменения не нужны. Оно касается ситуации, когда мы определенно не хотим создавать зависимость между вызываемым и более крупным объектами. В таких случаях разумно распаковать данные и передать их как параметры, но необходимо учесть, каких трудов это стоит. Если список параметров оказывается слишком длинным или модификации слишком частыми, следует пересмотреть структуру зависимостей.

Расходящиеся модификации

Мы структурируем программы, чтобы облегчить их модификацию; в конце концов, программы тем и отличаются от «железа», что их можно менять. Мы хотим, чтобы при модификации можно было найти в системе одно определенное место и внести изменения именно туда. Если этого сделать не удается, то тут пахнет двумя тесно связанными проблемами.

Расходящиеся (divergent) модификации имеют место тогда, когда один класс часто модифицируется различными способами по разным причинам. Если, глядя на класс, вы отмечаете для себя, что эти три метода придется модифицировать для каждой новой базы данных, а эти четыре метода придется модифицировать при каждом появлении нового финансового инструмента, это может означать, что вместо одного класса лучше иметь два. Благодаря этому каждый класс будет иметь свою четкую зону ответственности и изменяться в соответствии с изменениями в этой зоне. Не исключено, что это обнаружится лишь после добавления нескольких баз данных или финансовых инструментов. При каждой модификации, вызванной новыми условиями, должен изменяться один класс, и вся типизация в новом классе должна выражать эти условия. Для того чтобы все это привести в порядок, определяется все, что изменяется по данной причине, а затем применяется «Выделение класса» (Extract Class), чтобы объединить это все вместе.

«Стрельба дробью»

«Стрельба дробью» похожа на расходящуюся модификацию, но является ее противоположностью. Учуять ее можно, когда при выполнении любых модификаций приходится вносить множество мелких изменений в большое число классов. Когда изменения разбросаны повсюду, их трудно находить и можно пропустить важное изменение.

В такой ситуации следует использовать «Перемещение метода» (Move Method) и «Перемещение поля» (Move Field), чтобы свести все изменения в один класс. Если среди имеющихся классов подходящего кандидата нет, создайте новый класс. Часто можно воспользоваться «Встраиванием класса» (Inline Class), чтобы поместить целую связку методов в один класс. Возникнет какое-то число расходящихся модификаций, но с этим можно справиться.

Расходящаяся модификация имеет место, когда есть один класс, в котором производится много типов изменений, а «стрельба дробью» - это одно изменение, затрагивающее много классов. В обоих случаях желательно сделать так, чтобы в идеале между частыми изменениями и классами было взаимно однозначное отношение.

Завистливые функции

Весь смысл объектов в том, что они позволяют хранить данные вместе с процедурами их обработки. Классический пример дурного запаха - метод, который больше интересуется не тем классом, в котором он находится, а каким-то другим. Чаще всего предметом зависти являются данные. Не счесть случаев, когда мы сталкивались с методом, вызывающим полдюжины методов доступа к данным другого объекта, чтобы вычислить некоторое значение. К счастью, лечение здесь очевидно: метод явно напрашивается на перевод в другое место, что и достигается «Перемещением метода» (Move Method). Иногда завистью страдает только часть метода; в таком случае к завистливому фрагменту применяется «Выделение метода» (Extract Method), дающее ему то, о чем он мечтает.

Конечно, встречаются нестандартные ситуации. Иногда метод использует функции нескольких классов, так в который из них его лучше поместить? На практике мы определяем, в каком классе находится больше всего данных, и помещаем метод вместе с этими данными. Иногда легче с помощью «Выделения метода» (Extract Method) разбить метод на несколько частей и поместить их в разные места.

Разумеется, есть несколько сложных схем, нарушающих это правило. На ум сразу приходят паттерны «Стратегия» (Strategy pattern, Gang of Four) и «Посетитель» (Visitor pattern, Gang of Four) «банды четырех» [Gang of Four]. Самоделегирование Кента Бека [Beck] дает другой пример. С его помощью можно бороться с душком расходящихся модификаций. Фундаментальное практическое правило гласит: то, что изменяется одновременно, надо хранить в одном месте. Данные и функции, использующие эти данные, обычно изменяются вместе, но бывают исключения. Наталкиваясь на такие исключения, мы перемещаем функции, чтобы изменения осуществлялись в одном месте. Паттерны «Стратегия» и «Посетитель» позволяют легко изменять поведение, потому что они изолируют небольшой объем функций, которые должны быть заменены, ценой увеличения косвенности.

Группы данных

Элементы данных - как дети: они любят собираться в группы. Часто можно видеть, как одни и те же три-четыре элемента данных попадаются в множестве мест: поля в паре классов, параметры в нескольких сигнатурах методов. Связки данных, встречающихся совместно, надо превращать в самостоятельный класс. Сначала следует найти, где эти группы данных встречаются в качестве полей. Применяя к полям «Выделение метода» (Extract Method), преобразуйте группы данных в класс. Затем обратите внимание на сигнатуры методов и примените «Введение граничного объекта» (Introduce Parameter Object) или «Сохранение всего объекта» (Preserve Whole Object), чтобы сократить их объем. В результате сразу удается укоротить многие списки параметров и упростить вызов методов. Пусть вас не беспокоит, что некоторые группы данных используют лишь часть полей нового объекта. Заменив два или более полей новым объектом, вы оказываетесь в выигрыше.

Хорошая проверка: удалить одно из значений данных и посмотреть, сохранят ли при этом смысл остальные. Если нет, это верный признак того, что данные напрашиваются на объединение их в объект.

Сокращение списков полей и параметров, несомненно, удаляет некоторые дурные запахи, но после создания классов можно добиться и приятных ароматов. Можно поискать завистливые функции и обнаружить методы, которые желательно переместить в образованные классы. Эти классы не заставят долго ждать своего превращения в полезных членов общества.

Одержимость элементарными типами

В большинстве программных сред есть два типа данных. Тип «запись» позволяет структурировать данные в значимые группы. Элементарные типы данных служат стандартными конструктивными элементами. С записями всегда связаны некоторые накладные расходы. Они могут представлять таблицы в базах данных, но их создание может оказаться неудобным, если они нужны лишь в одном-двух случаях.

Один из ценных аспектов использования 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, являющиеся примитивами во многих других средах, суть классы.

Те, кто занимается ООП недавно, обычно неохотно используют маленькие объекты для маленьких задач, например денежные классы, соединяющие численное значение и валюту, диапазоны с верхней и нижней границами и специальные строки типа телефонных номеров и почтовых индексов. Выйти из пещерного мира в мир объектов помогает рефакторинг «Замена значения данных объектом» (Replace Data Value with Object) для отдельных значений данных. Когда значение данного является кодом типа, обратитесь к рефакторингу «Замена кода типа классом» (Replace Type Code with Class), если значение не воздействует на поведение. Если есть условные операторы, зависящие от кода типа, может подойти «Замена кода типа подклассами» (Replace Type Code with Subclasses) или «Замена кода типа состоянием/ стратегией» (Replace Type Code with State/Strategy).

При наличии группы полей, которые должны находиться вместе, применяйте «Выделение класса» (Extract Class). Увидев примитивы в списках параметров, воспользуйтесь разумной дозой «Введения граничного объекта» (Introduce Parameter Object). Если обнаружится разборка на части массива, попробуйте «Замену массива объектом» (Replace Array with Object).

Операторы типа switch

Одним из очевидных признаков объектно-ориентированного кода служит сравнительная немногочисленность операторов типа switch (или case). Проблема, обусловленная применением switch, по существу, связана с дублированием. Часто один и тот же блок switch оказывается разбросанным по разным местам программы. При добавлении в переключатель нового варианта приходится искать все эти блоки switch и модифицировать их. Понятие полиморфизма в ООП предоставляет элегантный способ справиться с этой проблемой.

Как правило, заметив блок switch, следует подумать о полиморфизме. Задача состоит в том, чтобы определить, где должен происходить полиморфизм. Часто переключатель работает в зависимости от кода типа. Необходим метод или класс, хранящий значение кода типа. Поэтому воспользуйтесь «Выделением метода» (Extract Method) для выделения переключателя, а затем «Перемещением метода» (Move Method) для вставки его в тот класс, где требуется полиморфизм. В этот момент следует решить, чем воспользоваться - «Заменой кода типа подклассами» (Replace Type Code with Subclasses) или «Заменой кода типа состоянием/стратегией» (Replace Type Code with State/ Strategy). Определив структуру наследования, можно применить «Замену условного оператора полиморфизмом» (Replace Conditional with Polymorphism).

Если есть лишь несколько вариантов переключателя, управляющих одним методом, и не предполагается их изменение, то применение полиморфизма оказывается чрезмерным. В данном случае хорошим выбором будет «Замена параметра явными методами» (Replace Parameter with Explicit Method). Если одним из вариантов является null, попробуйте прибегнуть к «Введению объекта Null» (Introduce Null Object).

Параллельные иерархии наследования

Параллельные иерархии наследования в действительности являются особым случаем «стрельбы дробью». В данном случае всякий раз при порождении подкласса одного из классов приходится создавать подкласс другого класса. Признаком этого служит совпадение префиксов имен классов в двух иерархиях классов.

Общая стратегия устранения дублирования состоит в том, чтобы заставить экземпляры одной иерархии ссылаться на экземпляры другой. С помощью «Перемещения метода» (Move Method) и «Перемещения поля» (Move Field) иерархия в ссылающемся классе исчезает.

Ленивый класс

Чтобы сопровождать каждый создаваемый класс и разобраться в нем, требуются определенные затраты. Класс, существование которого не окупается выполняемыми им функциями, должен быть ликвидирован. Часто это класс, создание которого было оправданно в свое время, но уменьшившийся в результате рефакторинга. Либо это класс, добавленный для планировавшейся модификации, которая не была осуществлена. В любом случае следует дать классу возможность с честью умереть. При наличии подклассов с недостаточными функциями попробуйте «Свертывание иерархии» (Collapse Hierarchy). Почти бесполезные компоненты должны быть подвергнуты «Встраиванию класса» (Inline Class).

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 такие вещи, и хотят обеспечить набор механизмов для работы с вещами, которые не нужны. То, что получается в результате, труднее понимать и сопровождать. Если бы все эти механизмы использовались, их наличие было бы оправданно, в противном случае они только мешают, поэтому избавляйтесь от них.

Если есть абстрактные классы, не приносящие большой пользы, избавляйтесь от них путем «Сворачивания иерархии» (Collapse Hierarhy). Ненужное делегирование можно устранить с помощью «Встраивания класса» (Inline Class). Методы с неиспользуемыми параметрами должны быть подвергнуты «Удалению параметров» (Remove Parameter). Методы со странными абстрактными именами необходимо вернуть на землю путем «Переименования метода» (Rename Method).

Теоретическая общность может быть обнаружена, когда единственными пользователями метода или класса являются контрольные примеры. Найдя такой метод или класс, удалите его и контрольный пример, его проверяющий. Если есть вспомогательный метод или класс для контрольного примера, осуществляющий разумные функции, его, конечно, надо оставить.

Временное поле

Иногда обнаруживается, что в некотором объекте атрибут устанавливается только при определенных обстоятельствах. Такой код труден для понимания, поскольку естественно ожидать, что объекту нужны все его переменные. Можно сломать голову, пытаясь понять, для чего существует некоторая переменная, когда не удается найти, где она используется.

С помощью «Выделения класса» (Extract Class) создайте приют для бедных осиротевших переменных. Поместите туда весь код, работающий с этими переменными. Возможно, удастся удалить условно выполняемый код с помощью «Введения объекта Null» (Introduce Null Object) для создания альтернативного компонента в случае недопустимости переменных.

Часто временные поля возникают, когда сложному алгоритму требуются несколько переменных. Тот, кто реализовывал алгоритм, не хотел пересылать большой список параметров (да и кто бы захотел?), поэтому он разместил их в полях. Но поля действенны только во время работы алгоритма, а в другом контексте лишь вводят в заблуждение. В таком случае можно применить «Выделение класса» (Extract Class) к переменным и методам, в которых они требуются. Новый объект является объектом метода [Beck].

Цепочки сообщений

Цепочки сообщений появляются, когда клиент запрашивает у одного объекта другой, у которого клиент запрашивает еще один объект, у которого клиент запрашивает еще один объект и т. д. Это может выглядеть как длинный ряд методов getThis или последовательность временных переменных. Такие последовательности вызовов означают, что клиент связан с навигацией по структуре классов. Любые изменения промежуточных связей означают необходимость модификации клиента.

Здесь применяется прием «Сокрытие делегирования» (Hide Delegate). Это может быть сделано в различных местах цепочки. В принципе, можно делать это с каждым объектом цепочки, что часто превращает каждый промежуточный объект в посредника. Обычно лучше посмотреть, для чего используется конечный объект. Попробуйте с помощью «Выделения метода» (Extract Method) взять использующий его фрагмент кода и путем «Перемещения метода» (Move Method) передвинуть его вниз по цепочке. Если несколько клиентов одного из объектов цепочки желают пройти остальную часть пути, добавьте метод, позволяющий это сделать.

Некоторых ужасает любая цепочка вызовов методов. Авторы известны своей спокойной, взвешенной умеренностью. По крайней мере, в данном случае это справедливо.

Посредник

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

Однако это может завести слишком далеко. Мы смотрим на интерфейс класса и обнаруживаем, что половина методов делегирует обработку другому классу. Тут надо воспользоваться «Удалением

посредника» (Remove Middle Man) и общаться с объектом, который действительно знает, что происходит. При наличии нескольких методов, не выполняющих большой работы, с помощью «Встраивания метода» (Inline Class) поместите их в вызывающий метод. Если есть дополнительное поведение, то с помощью «Замены делегирования наследованием» (Replace Delegation with Inheritance) можно преобразовать посредника в подкласс реального класса. Это позволит расширить поведение, не гонясь за всем этим делегированием.

Неуместная близость

Иногда классы оказываются в слишком близких отношениях и чаще, чем следовало бы, погружены в закрытые части друг друга. Мы не ханжи, когда это касается людей, но считаем, что классы должны следовать строгим пуританским правилам.

Чрезмерно интимничающие классы нужно разводить так же, как в прежние времена это делали с влюбленными. С помощью «Перемещения метода» (Move Method) и «Перемещения поля» (Move Field) необходимо разделить части и уменьшить близость. Посмотрите, нельзя ли прибегнуть к «Замене двунаправленной связи однонаправленной» (Change Bidirectional Association to Unidirectional). Если у классов есть общие интересы, воспользуйтесь «Выделением класса» (Extract Class), чтобы поместить общую часть в надежное место и превратить их в добропорядочные классы. Либо воспользуйтесь «Сокрытием делегирования» (Hide Delegate), позволив выступить в качестве связующего звена другому классу.

К чрезмерной близости может приводить наследование. Подклассы всегда знают о своих родителях больше, чем последним хотелось бы. Если пришло время расстаться с домом, примените «Замену наследования делегированием» (Replace Inheritance with Delegation).

Альтернативные классы с разными интерфейсами

Применяйте «Переименование метода» (Rename Method) ко всем методам, выполняющим одинаковые действия, но различающимся сигнатурами. Часто этого оказывается недостаточно. В таких случаях классы еще недостаточно деятельны. Продолжайте применять «Перемещение метода» (Move Method) для передачи поведения в классы, пока протоколы не станут одинаковыми. Если для этого приходится осуществить избыточное перемещение кода, можно попробовать компенсировать это «Выделением родительского класса» (Extract Superclass).

Неполнота библиотечного класса

Повторное использование кода часто рекламируется как цель применения объектов. Мы полагаем, что значение этого аспекта переоценивается (нам достаточно простого использования). Не будем отрицать, однако, что программирование во многом основывается на применении библиотечных классов, благодаря которым неизвестно, забыли мы, как действуют алгоритмы сортировки, или нет.

Разработчики библиотечных классов не всеведущи, и мы не осуждаем их за это; в конце концов, нередко проект становится понятен нам лишь тогда, когда он почти готов, поэтому задача у разработчиков библиотек действительно нелегкая. Проблема в том, что часто считается дурным тоном и обычно оказывается невозможным модифицировать библиотечный класс, чтобы он выполнял какие-то желательные действия. Это означает, что испытанная тактика вроде «Перемещения метода» (Move Method) оказывается бесполезной.

Для этой работы у нас есть пара специализированных инструментов. Если в библиотечный класс надо включить всего лишь один-два новых метода, можно выбрать «Введение внешнего метода» (Introduce Foreign Method). Если дополнительных функций достаточно много, необходимо применить «Введение локального расширения» (Introduce Local Extension).

Классы данных

Такие классы содержат поля, методы для получения и установки значений этих полей и ничего больше. Такие классы - бессловесные хранилища данных, которыми другие классы наверняка манипулируют излишне обстоятельно. На ранних этапах в этих классах могут быть открытые поля, и тогда необходимо немедленно, пока никто их не обнаружил, применить «Инкапсуляцию поля» (Incapsulate Field). При наличии полей коллекций проверьте, инкапсулированы ли они должным образом, и если нет, примените «Инкапсуляцию коллекции» (Incapsulate Collection). Примените «Удаление метода установки значения» (Remove Setting Method) ко всем полям, значение которых не должно изменяться.

Посмотрите, как эти методы доступа к полям используются другими классами. Попробуйте с помощью «Перемещения метода» (Move Method) переместить методы доступа в класс данных. Если метод не удается переместить целиком, обратитесь к «Выделению метода» (Extract Method), чтобы создать такой метод, который

можно переместить. Через некоторое время можно начать применять «Сокрытие метода» (Hide Method) к методам получения и установки значений полей.

Классы данных (и в этом они похожи на элементы данных) - как дети. В качестве отправной точки они годятся, но чтобы участвовать в работе в качестве взрослых объектов, они должны принять на себя некоторую ответственность.

Отказ от наследства

Подклассам полагается наследовать методы и данные своих родителей. Но как быть, если наследство им не нравится или попросту не требуется? Получив все эти дары, они пользуются лишь малой их частью.

Обычная история при этом - неправильно задуманная иерархия. Необходимо создать новый класс на одном уровне с потомком и с помощью «Спуска метода» (Push Down Method) и «Спуска поля» (Push Down Field) вытолкнуть в него все бездействующие методы. Благодаря этому в родительском классе будет содержаться только то, что используется совместно. Часто встречается совет делать все родительские классы абстрактными.

Мы этого советовать не станем, по крайней мере, это годится не на все случаи жизни. Мы постоянно обращаемся к созданию подклассов для повторного использования части функций и считаем это совершенно нормальным стилем работы. Кое-какой запашок обычно остается, но не очень сильный. Поэтому мы говорим, что если с не принятым наследством связаны какие-то проблемы, следуйте обычному совету. Однако не следует думать, что это надо делать всегда. В девяти случаях из десяти запах слишком слабый, чтобы избавиться от него было необходимо.

Запах отвергнутого наследства становится значительно крепче, когда подкласс повторно использует функции родительского класса, но не желает поддерживать его интерфейс. Мы не возражаем против отказа от реализаций, но при отказе от интерфейса очень возмущаемся. В этом случае не возитесь с иерархией; ее надо разрушить с помощью «Замены наследования делегированием» (Replace Inheritance with Delegation).

Комментарии

Не волнуйтесь, мы не хотим сказать, что писать комментарии не нужно. В нашей обонятельной аналогии комментарии издают не дурной, а даже приятный запах. Мы упомянули здесь комментарии потому, что часто они играют роль дезодоранта. Просто удивительно, как часто встречается код с обильными комментариями, которые появились в нем лишь потому, что код плохой.

Комментарии приводят нас к плохому коду, издающему все гнилые запахи, о которых мы писали в этой главе. Первым действием должно быть удаление этих запахов при помощи рефакторинга. После этого комментарии часто оказываются ненужными.

Если для объяснения действий блока требуется комментарий, попробуйте применить «Выделение метода» (Extract Method). Если метод уже выделен, но по-прежнему нужен комментарий для объяснения его действия, воспользуйтесь «Переименованием метода» (Rename Method). А если требуется изложить некоторые правила, касающиеся необходимого состояния системы, примените «Введение утверждения» (Introduce Assertion).

Примечание

Почувствовав потребность написать комментарий, попробуйте сначала изменить структуру кода так, чтобы любые комментарии стали излишними.

Комментарии полезны, когда вы не знаете, как поступить. Помимо описания происходящего, комментарии могут отмечать те места, в которых вы не уверены. Правильным будет поместить в комментарии обоснование своих действий. Это пригодится тем, кто будет модифицировать код в будущем, особенно если они страдают забывчивостью.

4 РАЗРАБОТКА ТЕСТОВ

При проведении рефакторинга важным предварительным условием является наличие надежных тестов. Даже если вам посчастливилось и у вас есть средство, автоматизирующее рефакторинг, без тестов все равно не обойтись. Время, когда все возможные методы рефакторинга будут автоматически выполняться с помощью специального инструмента, наступит не скоро.

Я не считаю это серьезной помехой. Мой опыт показывает, что, создав хорошие тесты, можно значительно увеличить скорость программирования, даже если это не связано с рефакторингом. Это оказалось сюрпризом для меня и противоречит интуиции многих программистов, поэтому стоит разобраться в причинах такого результата.

Ценность самотестирующегося кода

Если посмотреть, на что уходит время у большинства программистов, то окажется, что на написание кода в действительности тратится весьма небольшая его часть. Некоторое время уходит на уяснение задачи, еще какая-то его часть - на проектирование, а львиную долю времени занимает отладка. Уверен, что каждый читатель может припомнить долгие часы отладки, часто до позднего вечера. Любой программист может рассказать историю о том, как на поиски какой-то ошибки ушел целый день (а то и более). Исправить ошибку обычно удается довольно быстро, но поиск ее превращается в кошмар. А при исправлении ошибки всегда существует возможность внести новую, которая проявится гораздо позднее, и пройдет вечность, прежде чем вы ее обнаружите.

Событием, подтолкнувшим меня на путь создания самотестирующегося кода, стал разговор на OOPSLA в 92-м году. Кто-то (кажется, это был Дэйв Томас (Dave Thomas)) небрежно заметил: «Классы должны содержать тесты для самих себя». Мне пришло в голову, что это хороший способ организации тестов. Я истолковал эти слова так, что в каждом классе должен быть свой метод (пусть он называется test), с помощью которого он может себя протестировать.

В то время я также занимался пошаговой разработкой (incremental development) и поэтому попытался по завершении каждого шага добавлять в классы тестирующие методы. Проект тот был совсем небольшим, поэтому каждый шаг занимал у нас примерно неделю. Выполнять тесты стало достаточно просто, но хотя запускать их было легко, само тестирование все же оставалось делом весьма утомительным, потому что каждый тест выводил результаты на консоль и приходилось их проверять. Должен заметить, что я человек достаточно ленивый и готов немало потрудиться, чтобы избавиться от работы. Мне стало ясно, что можно не смотреть на экран в поисках некоторой информации модели, которая должна быть выведена, а заставить компьютер заниматься этим. Все, что требовалось, - это поместить ожидаемые данные в код теста и провести в нем сравнение. Теперь можно было выполнять тестирующий метод каждого класса, и если все было нормально, он просто выводил на экран сообщение «ОК». Классы стали самотестирующимися.

Примечание

Делайте все тесты полностью автоматическими, так чтобы они проверяли собственные результаты.

Теперь выполнять тесты стало не труднее, чем компилировать. Я начал выполнять их при каждой компиляции. Вскоре обнаружилось, что производительность моего труда резко возросла. Я понял, что перестал тратить на отладку много времени. Если я допускал ошибку, перехватываемую предыдущим тестом, это обнаруживалось, как только я запускал этот тест. Поскольку раньше этот тест работал, мне становилось известно, что ошибка была в том, что я сделал после предыдущего тестирования. Тесты я запускал часто, каждые несколько минут, и потому знал, что ошибка была в том коде, который я только что написал. Этот код был еще свеж в памяти и невелик по размеру, поэтому отыскать ошибку труда не составляло. Теперь ошибки, которые раньше можно было искать часами, обнаруживались за считанные минуты. Мощное средство обнаружения ошибок появилось у меня не только благодаря тому, что я строил классы с самотестированием, ной потому, что я часто запускал их тесты.

Заметив все это, я стал проводить тестирование более агрессивно. Не дожидаясь завершения шага разработки, я стал добавлять тесты сразу после написания небольших фрагментов функций. Обычно за день добавлялась пара новых функций и тесты для их проверки. Я стал тратить на отладку не более нескольких минут в день.

Примечание

Комплект тестов служит мощным детектором ошибок, резко сокращающим время их поиска.

Конечно, убедить других последовать тем же путем нелегко. Для тестов необходим большой объем кода. Самотестирование кажется бессмысленным, пока не убедишься сам, насколько оно ускоряет программирование. К тому же, многие не умеют писать тесты и даже никогда о них не думали. Проводить тестирование вручную неимоверно скучно, но если делать это автоматически, то написание тестов может даже доставлять удовольствие.

В действительности очень полезно писать тесты до начала программирования. Когда требуется ввести новую функцию, начните с создания теста. Это не так глупо, как может показаться. Когда вы пишете тест, то спрашиваете себя, что нужно сделать для добавления этой функции. При написании теста вы также сосредоточиваетесь на интерфейсе, а не на реализации, что всегда полезно. Это также значит, что появляется четкий критерий завершения кодирования - в итоге тест должен работать.

Идея частого тестирования составляет важную часть экстремального программирования [Beck, XP]. Название вызывает представление о программистах из числа скорых и небрежных хакеров. Однако экстремальные программисты весьма привержены тестированию. Они стремятся разрабатывать программы как можно быстрее и знают, что тесты позволяют двигаться вперед с исключительной скоростью.

Но хватит полемизировать. Хотя я уверен, что написание самотестирующегося кода выгодно всем, эта книга посвящена другому, а именно рефакторингу. Рефакторинг требует тестов. Тому, кто собирается заниматься рефакторингом, необходимо писать тесты. В этой главе дается первое представление о том, как это делать для Java. Эта книга не о тестировании, поэтому в особые детали я вникать не стану. Но что касается тестирования, то я убедился, что весьма малый его объем может принести весьма существенную пользу.

Как и все остальное в этой книге, подход к тестированию описывается на основе примеров. Разрабатывая код, я пишу тесты по ходу дела. Но часто, когда в рефакторинге также принимают участие другие люди, приходится работать с большим объемом кода, в котором отсутствует самотестирование. Поэтому прежде чем применить рефакторинг, нужно сделать код самотестирующимся.

Стандартной идиомой Java для тестирования является тестирующая функция main. Идея состоит в том, чтобы в каждом классе была тестирующая функция main, проверяющая класс. Это разумное соглашение (хотя оно не очень в чести), но оно может оказаться неудобным. При таком соглашении сложно запускать много тестов. Другой подход заключается в создании отдельных тестирующих классов, работа которых в среде должна облегчить тестирование.

Среда тестирования JUnit

Я пользуюсь средой тестирования JUnit - системой с открытым исходным кодом, которую разработали Эрих Гамма (Erich Gamma) и Кент Бек [JUnit]. Она очень проста, но, тем не менее, позволяет делать все главные вещи, необходимые для тестирования. В данной главе с помощью этой среды я разрабатываю тесты для некоторых классов ввода/вывода.

Сначала будет создан класс FileReaderTester для тестирования класса, читающего файлы. Все классы, содержащие тесты, должны создаваться как подклассы класса контрольного примера из среды тестирования. В среде используется составной паттерн (composite pattern) [Gang of Four], позволяющий объединять тесты в группы (рисунок 4.1).

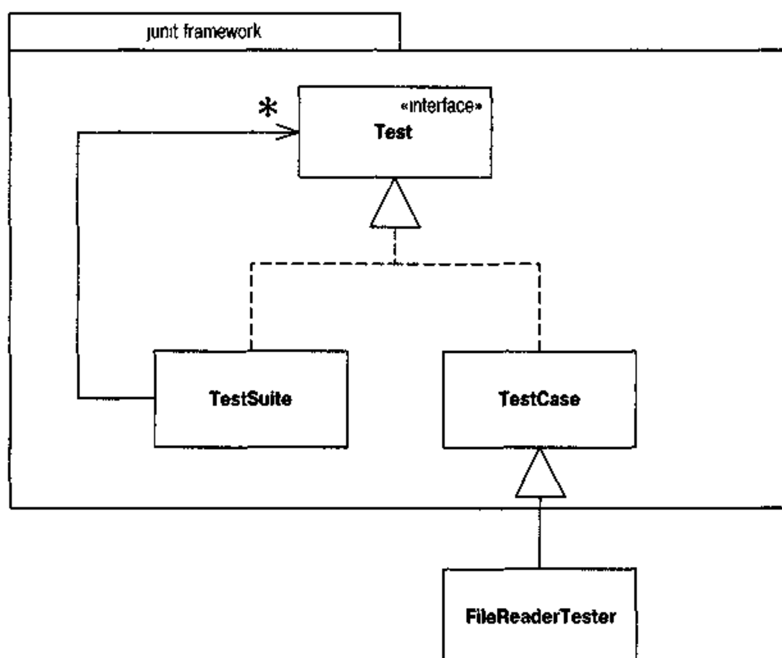


Рисунок 4.1 - Составная структура тестов

В этих наборах могут содержаться простые контрольные примеры или другие наборы контрольных примеров. Последние благодаря этому легко объединяются, и тесты выполняются автоматически.

```
class FileReaderTester extends TestCase {
```

```
    public FileReaderTester (String name) {
        super(name);
    }
}
```

В новом классе должен быть конструктор. После этого можно начать добавление кода теста. Первая задача - подготовить тестовые данные (test fixture). По существу, это объекты, выступающие в качестве образцов для тестирования. Поскольку будет происходить чтение файла, надо подготовить проверочный 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

Для того чтобы использовать файл в дальнейшем, я готовлю тестовые данные. В классе контрольного примера есть два метода работы с тестовыми данными: setUp создает объекты, а tearDown удаляет их. Оба реализованы как нулевые методы контрольного примера. Обычно удалять объекты нет необходимости (это делает сборщик мусора), но здесь будет правильным использовать для закрытия файла удаляющий метод:

```
class FileReaderTester {
    protected void setUp() {
        try {
            _input = new FileReader("data.txt");
        }
        catch (FileNotFoundException e) {
            throw new RuntimeException ("невозможно открыть тестовый файл");
        }
    }
    protected void tearDown() {
        try {
            _input.close();
        }
        catch (IOException e) {
            throw new RuntimeException ("ошибка при закрытии тестового файла");
        }
    }
}
```

Теперь, когда тестовые данные готовы, можно начать писать тесты. Сначала будем проверять метод чтения. Для этого я читаю несколько символов, а затем проверяю, верен ли очередной прочитанный символ:

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

Автоматическое тестирование выполняет метод assert. Если выражение внутри assert истинно, все в порядке. В противном случае генерируется сообщение об ошибке. Позднее я покажу, как это делает среда, но сначала опишу, как выполнить тест.

Первый шаг- создание комплекта тестов. Для этого создается метод с именем suite:

```
class FileReaderTester {
    public static Test suite() {
```

```
        TestSuite suite= new TestSuite();
        suite.addTest(new FileReaderTester("testRead"));
        return suite;
     }
    }
```

Этот комплект тестов содержит только один объект контрольного примера, экземпляр FileReaderTester. При создании контрольного примера я передаю конструктору аргумент в виде строки с именем метода, который собираюсь тестировать. В результате создается один объект, тестирующий один этот метод. Тест работает с тестируемым объектом при помощи Java reflection API. Чтобы понять, как это происходит, можно посмотреть исходный код. Я просто отношусь к этому как к магии.

Для запуска тестов я пользуюсь отдельным классом TestRunner. Есть две разновидности TestRunner: одна с развитым GUI, а другая - с простым символьным интерфейсом. Вариант с символьным интерфейсом можно вызывать в main:

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

Этот код создает класс, прогоняющий тесты, и требует, чтобы тот протестировал класс FileReaderTester. После прогона выводится следующее:

```
    Time 0.110
    0K (1 tests)
```

JUnit выводит точку для каждого выполняемого теста, чтобы можно было следить за продвижением вперед. Среда также сообщает о времени выполнения тестов. После этого, если не возникло неприятностей, выводится сообщение «ОК» и количество выполненных тестов. Можно выполнить тысячу тестов, и если все в порядке, будет выведено это сообщение. Такая простая обратная связь существенна в самотестирующемся коде. Если ее не будет, то тесты никогда не будут запускаться достаточно часто. Зато при ее наличии можно запустить кучу тестов, отправиться на обед (или на совещание), а вернувшись, посмотреть на результат.

Примечание

Чаще запускайте тесты. Запускайте тесты при каждой компиляции - каждый тест хотя бы раз в день.

При проведении рефакторинга выполняется лишь несколько тестов для проверки кода, над которым вы работаете. Можно выполнять лишь часть тестов, т. к. тестирование должно происходить быстро, в противном случае оно будет замедлять работу, и может возникнуть соблазн отказаться от него. Не поддавайтесь этому соблазну - возмездие неминуемо.

Что будет, если что-то пойдет не так, как надо? Введем ошибку умышленно:

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

Результат выглядит так:

```
    Time 0.220
    !!!FAILURES!!!
    Test Results
    Run 1 Failures 1 Errors 0
    There was 1 failure
    1) FileReaderTester testRead
```

```
test framework AssertionFailedError
```

Среда тестирования предупреждает об ошибке и сообщает, который из тестов оказался неудачным. Это сообщение об ошибке не слишком информативно. Можно улучшить его с помощью другого формата assert.

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

В большинстве утверждений assert проверяется равенство двух значений, для которого среда тестирования содержит assertEquals. Это удобно: для сравнения объектов в нем применяется equals(), а для сравнения значений - знак ==, о чем я часто забываю. Кроме того, в этом формате выводится более содержательное сообщение об ошибке:

```
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 заметить, что при создании тестов я часто начинаю с того, что проверяю их на отказ. Уже существующий код модифицируется так, чтобы тест завершался неудачей (если код доступен), либо в assert помещается неверное ожидаемое значение. Делается это для того, чтобы проверить, действительно ли выполняется тест и действительно ли он проверяет то, что требуется (вот почему я предпочитаю изменить тестируемый код, если это возможно). Можно считать это паранойей, но есть риск действительно запутаться, если тесты проверяют не то, что должны по вашим предположениям.

Помимо перехвата неудачного выполнения тестов (утверждений, оказывающихся ложными) среда тестирования перехватывает ошибки (неожиданные исключительные ситуации). Если закрыть поток и попытаться после этого произвести из него чтение, должна возникнуть исключительная ситуация. 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();  //генерирует исключительную ситуацию
 }
 assertEquals( 'm', ch);
}
```

При выполнении этого кода получается следующий результат:

```
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 не прошедшие тесты и ошибки, потому что они по-разному обнаруживаются и процесс отладки для них разный. В JUnit также удачно сделан GUI (рисунок 4.2). Индикатор выполнения имеет зеленый цвет, если все тесты прошли, и красный цвет, если возникли какие-либо отказы. Можно постоянно держать GUI

открытым, а среда тестирования автоматически подключает любые изменения, сделанные в коде. Это дает очень удобный способ выполнения тестов.

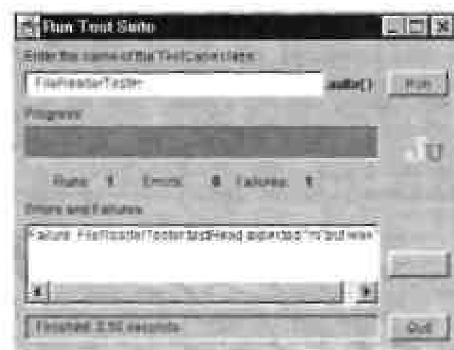


Рисунок 4.2 - Графический интерфейс JUnit

Тесты модулей и функциональные тесты

Данная среда используется для тестирования модулей, поэтому я хочу остановиться на различии между тестированием модулей и тестированием функциональности. Тесты, о которых идет речь, - это тесты модулей (unit tests). Я пишу их для повышения своей продуктивности как программиста. Побочный эффект при этом - удовлетворение отдела контроля качества. Тесты модулей тесно связаны с определенным местом. Каждый класс теста действует внутри одного пакета. Он проверяет интерфейсы с другими пакетами, но исходит из того, что за их пределами все работает.

Функциональные тесты (functional tests) - совсем другое дело. Их пишут для проверки работоспособности системы в целом. Они гарантируют качество продукта потребителю и нисколько не заботятся о производительности программиста. Разрабатывать их должна отдельная команда программистов из числа таких, для которых обнаружить ошибку - удовольствие. При этом они используют тяжеловесные инструменты и технологии.

Обычно в функциональных тестах стремятся в возможно большей мере рассматривать систему в целом как черный ящик. В системе, основанной на GUI, действуют через этот GUI. Для программы, модифицирующей файлы или базы данных, эти тесты только проверяют, какие получаются результаты при вводе определенных данных.

Когда при функциональном тестировании или работе пользователя обнаруживается ошибка в программном обеспечении, для ее исправления требуются, по меньшей мере, две вещи. Конечно, надо модифицировать код программы и устранить ошибку. Но необходимо также добавить тест модуля, который обнаружит эту ошибку. Действительно, если в моем коде найдена ошибка, я начинаю с того, что пишу тест модуля, который показывает эту ошибку. Если требуется точнее определить область действия ошибки или с ней могут быть связаны другие отказы, я пишу несколько тестов. С помощью тестов модулей я нахожу точное местоположение ошибки и гарантирую, что такого рода ошибка больше не проскочит через мои тесты.

Примечание

Получив сообщение об ошибке, начните с создания теста модуля, показывающего эту ошибку.

Среда JUnit разработана для создания тестов модулей. Функциональные тесты часто выполняются другими средствами. Хорошим примером служат средства тестирования на основе GUI. Однако часто имеет смысл написать собственные средства тестирования для конкретного приложения, с помощью которых легче организовать контрольные примеры, чем при использовании одних только сценариев GUI. С помощью JUnit можно осуществлять функциональное тестирование, но обычно это не самый эффективный способ. При решении задач рефакторинга я полагаюсь на тесты модулей - испытанного друга программистов.

Добавление новых тестов

Продолжим добавление тестов. Я придерживаюсь того стиля, при котором все действия, выполняемые классом, рассматриваются и проверяются во всех условиях, которые могут вызвать отказ класса. Это не то же самое, что тестирование всех открытых методов, пропагандируемое некоторыми программистами. Тестирование должно быть направлено на ситуации, связанные с риском. Помните, что необходимо найти ошибки, которые могут возникнуть сейчас или в будущем. Поэтому я не проверяю методы доступа, осуществляющие лишь чтение или запись поля: они настолько просты, что едва ли в них обнаружится ошибка.

Это важно, поскольку стремление написать слишком много тестов обычно приводит к тому, что их оказывается недостаточно. Я прочел не одну книгу о тестировании, вызывавшую желание уклониться от той

необъятной работы, которую в них предлагалось проделать. Такое стремление приводит к обратным результатам, поскольку заставляет предположить, что тестирование требует чрезмерных затрат труда.

Тестирование приносит ощутимую пользу, даже если осуществляется в небольшом объеме. Ключ к проблеме в том, чтобы тестировать те области, возможность ошибок в которых вызывает наибольшее беспокойство. При таком подходе затраты на тестирование приносят максимальную выгоду.

Примечание

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

В данный момент я рассматриваю метод чтения. Что еще он должен делать? В частности, видно, что он возвращает -1 в конце файла (на мой взгляд, не очень удачный протокол, но, пожалуй, программистам на С он должен 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(-1, ch);
}
```

Чтобы тест выполнялся, надо добавить его в комплект:

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

Выполняясь, этот комплект дает команду на прогон всех тестов, из которых состоит (двух контрольных примеров). Каждый контрольный пример выполняет setUp, код тестирующего метода и в заключение tearDown. Важно каждый раз выполнять setUp и tearDown, чтобы тесты были изолированы друг от друга. Это означает, что можно выполнять их в любом порядке.

Помнить о необходимости добавления тестов в метод suite обременительно. К счастью, Эрих Гамма и Кент Бек так же ленивы, как и я, поэтому они предоставили возможность избежать этого. Специальный конструктор комплекта тестов принимает класс в качестве параметра. Этот конструктор формирует комплект 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 тест, который я напишу, будет включен в комплект.

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

```
public void testReadBoundanes()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());
}
```

Обратите внимание, что в утверждение можно поместить сообщение, выводимое в случае неудачи теста.

Примечание

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

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());
}
```

Здесь я создаю дополнительные тестовые данные для этого теста. Если в дальнейшем мне понадобится пустой файл, можно поместить его в обычный набор тестов, перенеся 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 продолжить чтение за концом файла? Снова надо возвратить -1, и я изменяю еще один тест, чтобы проверить это:

```
public void testReadBoundanes()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 на то, как я играю роль противника кода. Я активно стараюсь вывести его из строя. Такое умонастроение кажется мне одновременно продуктивным и забавным. Оно доставляет удовольствие темной стороне моей натуры.

Проверяя тесты, не забывайте убедиться, что ожидаемые ошибки происходят в надлежащем порядке. При попытке чтения потока после его закрытия должно генерироваться исключение IOException. Это тоже надо проверять:

```
public void testReadAfterClose() throws IOException {
 _input.close();
 try {
    _input.read();
    fail("не возбуждена исключительная ситуация при чтении за концом файла");
 }
 catch (IOException e) {}
}
```

Все другие исключительные ситуации, кроме IOException, генерируют ошибку обычным образом.

Примечание

Не забывайте проверять, чтобы в случае возникновения проблем генерировались исключительные ситуации.

В этих строках продолжается облечение тестов плотью. Для того чтобы протестировать интерфейс некоторых классов, придется потрудиться, но в процессе обретается действительное понимание интерфейса класса. В частности, благодаря этому 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 комплекты, помещая в создаваемые для этого 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(FileWnterTester.class));
    // и т д
    return result;
 }
}
```

Когда следует остановиться? Наверняка вы неоднократно слышали, что нельзя доказать отсутствие в программе ошибок путем тестирования. Это верное замечание, но оно не умаляет способности тестирования повысить скорость программирования. Предложено немало правил, призванных гарантировать, что протестированы все мыслимые комбинации. Ознакомиться с ними полезно, но не стоит принимать их слишком близко к сердцу. Они могут уменьшить выгоду, приносимую тестированием, и существует опасность, что попытка написать слишком много тестов окажется настолько обескураживающей, что в итоге вы вообще перестанете писать тесты. Необходимо сконцентрироваться на 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 во время рефакторинга можно лучше понять программу и благодаря этому найти больше ошибок. Я всегда начинаю рефакторинг с создания комплекта тестов, но по мере продвижения вперед неизменно пополняю его.

Примечание

Опасение по поводу того, что тестирование не выявит все ошибки, не должно помешать написанию тестов, которые выявят большинство ошибок.

Одна из сложностей, связанных с объектами, состоит в том, что наследование и полиморфизм затрудняют тестирование, поскольку приходится проверить очень много комбинаций. Если у вас есть три абстрактных класса, работающих друг с другом, у каждого из которых три подкласса, то получается девять разных вариантов использования интерфейсов (имеется в виду, что каждый конкретный класс может представляться, используя интерфейс абстрактного родительского класса). С другой стороны, существует двадцать семь возможных комбинаций обмена интерфейсами между классами. Я не стараюсь всегда проверять все возможные комбинации обмена интерфейсами, но пытаюсь проверить все варианты использования интерфейсов. Если варианты использования интерфейсов достаточно независимы один от другого, я обычно не пытаюсь перепробовать все комбинации. Всегда есть риск что-то пропустить, но лучше потратить разумное время, чтобы выявить большинство ошибок, чем потратить вечность, пытаясь найти их все.

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

5 НА ПУТИ К КАТАЛОГУ МЕТОДОВ РЕФАКТОРИНГА

Главы с 5 по 12 составляют начальный каталог методов рефакторинга. Их источником служит мой опыт рефакторинга нескольких последних лет. Этот каталог не следует считать исчерпывающим или бесспорным, но он должен обеспечить надежную отправную точку для проведения пользователем рефакторинга.

Формат методов рефакторинга

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

- Сначала идет **название (name)**. Название важно для создания словаря методов рефакторинга и используется в книге повсюду.
- За названием следует **краткая сводка (summary)** ситуаций, в которых требуется данный метод, и краткие сведения о том, что этот метод делает. Это позволяет быстрее находить необходимый метод.
- **Мотивировка (motivation)** описывает, почему следует пользоваться этим методом рефакторинга и в каких случаях применять его не следует.
- **Техника (mechanics)** подробно шаг за шагом описывает, как применять этот метод рефакторинга.
- **Примеры (examples)** показывают очень простое применение метода, чтобы проиллюстрировать его действие.

Краткая сводка содержит формулировку проблемы, решению которой способствует данный метод, сжатое описание производимых действий и набросок простого примера, показывающего положение до и после проведения рефакторинга. Иногда я использую для наброска код, а иногда унифицированный язык моделирования (UML) в зависимости от того, что лучше передает сущность данного метода. (Все диаграммы UML в этой книге изображены с точки зрения реализации [Fowler, UML].) Если вы уже видели этот рефакторинг раньше, набросок должен дать хорошее представление о том, к чему относится данный рефакторинг. Если нет, то, вероятно, следует проработать пример, чтобы лучше понять смысл.

Техника основывается на заметках, сделанных мной, чтобы вспомнить, как выполнять данный рефакторинг после того, как я некоторое время им не пользовался. Поэтому данный раздел, как правило, довольно лаконичен и не поясняет, почему выполняются те или иные шаги. Более пространные пояснения даются в примере. Таким образом, техника представляет собой краткие заметки, к которым можно обратиться, когда метод рефакторинга вам знаком, но необходимо вспомнить, какие шаги должны быть выполнены (так, по крайней мере, использую их я). При проведении рефакторинга впервые, вероятно, потребуется прочесть пример.

Я составил технику таким образом, чтобы каждый шаг рефакторинга был как можно мельче. Особое внимание обращается на соблюдение осторожности при проведении рефакторинга, для чего необходимо осуществлять его маленькими шажками, проводя тестирование после каждого из них. На практике я обычно двигаюсь более крупными шагами, чем описываемые здесь, а столкнувшись с ошибкой, делаю шаг назад и двигаюсь дальше маленькими шагами. Описание шагов содержит ряд ссылок на особые случаи. Таким образом, описание шагов выступает также в качестве контрольного списка; я часто сам забываю об этих вещах.

Примеры просты до смешного. В них я ставлю задачу помочь разъяснению базового рефакторинга, стараясь как можно меньше отвлекаться, поэтому надеюсь, что их упрощенность простительна. (Они, несомненно, не могут служить примерами добротного проектирования бизнес-объектов.) Я уверен, что вы сможете применить их к своим более сложным условиям. Для некоторых очень простых методов рефакторинга примеры отсутствуют, поскольку вряд ли от них могло быть много пользы.

В частности, помните, что эти примеры включены только для иллюстрации одного обсуждаемого метода рефакторинга. В большинстве случаев в получаемом коде сохраняются проблемы, для решения которых требуется применение других методов рефакторинга. В нескольких случаях, когда методы рефакторинга часто сочетаются, я переношу примеры из одного рефакторинга в другой. Чаще всего я оставляю код таким, каким он получается после одного рефакторинга. Это делается для того, чтобы каждый рефакторинг был автономным, поскольку каталог должен служить главным образом как справочник.

Не следует принимать эти примеры в качестве предложений относительно конструкции объектов служащего (employee) или заказа (order). Примеры приведены только для иллюстрации методов рефакторинга и ни для чего более. В частности, вы обратите внимание, что в примерах для представления денежных величин используется тип double. Это сделано лишь для упрощения примеров, поскольку представление не важно при проведении рефакторинга. Я настоятельно рекомендую избегать применения double для денежных величин в коммерческих программах. Для представления денежных величин я пользуюсь паттерном «Количество» (Quantity) [Fowler, AP].

Во время написания этой книги в коммерческих проектах чаще всего применялась версия Java 1.1, поэтому на нее и ориентировано большинство примеров; это особенно заметно для коллекций. Когда я завершал эту книгу, более широко доступной стала версия Java 2. Я не вижу необходимости переделывать все примеры, потому что коллекции с точки зрения пояснения рефакторинга имеют второстепенное значение. Однако некоторые методы рефакторинга, такие как «Инкапсуляция коллекции» (Encapsulate Collection, 214), в Java 2 осуществляются иначе. В таких случаях я излагаю обе версии - для Java 2 и для Java 1.1.

Я использую полужирный моноширинный шрифт для выделения изменений в коде, если они окружены неизмененным кодом и могут быть замечены с трудом. Полужирным шрифтом выделяется не весь модифицированный код, поскольку его неумеренное применение 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 уменьшается риск пропустить ссылку, и поиск обычно выполняется значительно быстрее, чем при обычном просмотре кода.

В большинстве языков компьютерные программы рассматриваются как текстовые файлы, поэтому лучше всего прибегнуть к подходящему виду текстового поиска. Во многих средах программирования можно осуществлять поиск в одном файле или группе файлов.

Не выполняйте поиск и замену автоматически. Для каждой ссылки посмотрите, действительно ли она указывает на то, что вы хотите заменить. Можно искусно назначить шаблон поиска, но я всегда мысленно проверяю правильность выполняемой замены. Если в разных классах есть одноименные методы или в одном классе есть методы с разными сигнатурами, весьма велика вероятность ошибки.

В сильно типизированных языках поиску может помочь компилятор. Часто можно удалить прежнюю функцию и позволить компилятору искать висячие ссылки. Это хорошо тем, что компилятор найдет их все. Однако с таким подходом связаны некоторые проблемы.

Во-первых, компилятор запутается, если функция определена в иерархии наследования несколько раз. Особенно это касается поиска методов, перегружаемых по несколько раз. Работая с иерархией, используйте текстовый поиск, чтобы узнать, не определен ли метод, которым вы занимаетесь, в других классах.

Вторая проблема заключается в том, что компилятор может оказаться слишком медлительным для такой работы. В таком случае обратитесь сначала к текстовому поиску; по крайней мере, компилятор сможет перепроверить ваши результаты. Данный способ действует, только если надо удалить метод, но иногда приходится посмотреть на то, как он используется, и решить, что делать дальше. В таких случаях применяйте вариант текстового поиска.

Третья проблема связана с тем, что компилятор не может перехватить использование Java reflection API. Это одна из причин, по которым применять Java reflection API следует с осторожностью. Если в системе используется Java reflection API, то необходимы текстовый поиск и дополнительное усиление тестирования. В ряде мест я предлагаю компилировать без тестирования в тех ситуациях, когда компилятор обычно перехватывает ошибки. В случае Java reflection API такие варианты не проходят, и многочисленные компиляции должны тестироваться.

Некоторые среды Java, в особенности VisualAge IBM, предоставляют возможности, аналогичные броузеру Smalltalk. Вместо текстового поиска для нахождения ссылок можно воспользоваться пунктами меню. В таких средах код хранится не в текстовых файлах, а в базе данных в оперативной памяти. Привыкните работать с этими пунктами меню; часто они превосходят по своим возможностям отсутствующий текстовый поиск.

Насколько зрелыми являются предлагаемые методы рефакторинга?

Любому пишущему на технические темы приходится выбирать момент для публикации своих идей. Чем раньше, тем скорее идеями могут воспользоваться другие. Однако человек всегда учится. Если опубликовать полусырые идеи, они могут оказаться недостаточными и даже послужить источником проблем для тех, кто попытается ими воспользоваться.

Базовая технология рефакторинга, т. е. внесение небольших модификаций и частое тестирование, проверена на протяжении многих лет, особенно в сообществе Smalltalk. Поэтому я уверен, что основная идея рефакторинга прочно установилась.

Методы рефакторинга, описываемые в данной книге, представляют собой мои заметки по поводу типов рефакторинга, которые применялись мной на практике. Однако есть разница между применением рефакторинга и сведением его в последовательность механических шагов, здесь описываемую. В частности, иногда встречаются проблемы, возникающие лишь при весьма специфических обстоятельствах. Не могу сказать, что те, кто выполнял описанные шаги, обнаружили много проблем такого рода. Применяя рефакторинг, думайте о

том, что вы делаете. Помните, что, используя рецепт, необходимо адаптировать рефакторинг к своим условиям. Если вы столкнетесь с интересной проблемой, напишите мне по электронной почте, и я постараюсь поставить других в известность об этих конкретных условиях.

Другой аспект этих методов рефакторинга, о котором необходимо помнить, - они описываются в предположении, что программное обеспечение выполняется в одном процессе. Со временем, надеюсь, будут описаны методы рефакторинга, применяемые при параллельном и распределенном программировании. Эти методы будут иными. Например, в программах, выполняемых в одном процессе, не надо беспокоиться о частоте вызова метода: вызов метода обходится дешево. Однако в распределенных программах переходы туда и обратно должны выполняться как можно реже. Для таких видов программирования есть отдельные типы рефакторинга, которые в данной книге не рассматриваются .

Многие методы рефакторинга, например «Замена кода типа состоянием/стратегией» (Replace Type Code with State/Strategy) или «Формирование шаблона метода» (Form Template Method), вводят в систему паттерны. В основной книге «банды четырех» сказано, что «паттерны проектирования предоставляют объекты для проведения рефакторинга». Существует естественная связь между паттернами и рефакторингом. Паттерны представляют собой цели; рефакторинг дает методы их достижения. В этой книге нет методов рефакторинга для всех известных паттернов, даже для всех паттернов «банды четырех» [Gang of Four]. Это еще одно отношение, в котором данный каталог не полон. Надеюсь, что когда-нибудь этот пробел будет исправлен.

Применяя эти методы рефакторинга, помните, что они представляют собой лишь начало пути. Вы наверняка обнаружите их недостаточность. Я публикую их сейчас потому, что считаю полезными несмотря на несовершенство. Полагаю, что они послужат для вас отправной точкой и повысят ваши возможности эффективного рефакторинга. Это то, чем они являются для меня.

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

6 СОСТАВЛЕНИЕ МЕТОДОВ

Значительная часть моих рефакторингов заключается в составлении методов, правильным образом оформляющих код. Почти всегда проблемы возникают из-за слишком длинных методов, часто содержащих массу информации, погребенной под сложной логикой, которую они обычно в себе заключают. Основным типом рефакторинга здесь служит «Выделение метода» (Extract Method), в результате которого фрагмент кода превращается в отдельный метод. «Встраивание метода» (Inline Method), по существу, является противоположной процедурой: вызов метода заменяется при этом кодом, содержащимся в теле. «Встраивание метода» (Inline Method) требуется мне после проведения нескольких выделений, когда я вижу, что какие-то из полученных методов больше не выполняют свою долю работы или требуется реорганизовать способ разделения кода на методы.

Самая большая проблема «Выделения метода» (Extract Method) связана с обработкой локальных переменных и, прежде всего, с наличием временных переменных. Работая над методом, я люблю с помощью «Замены временной переменной вызовом метода» (Replace Temp with Query) избавиться от возможно большего количества временных переменных. Если временная переменная используется для разных целей, я сначала применяю «Расщепление временной переменной» (Split Temporary Variable), чтобы облегчить последующую ее замену.

Однако иногда временные переменные оказываются слишком сложными для замены. Тогда мне требуется «Замена метода объектом метода» (Replace Method with Method Object). Это позволяет разложить даже самый запутанный метод, но ценой введения нового класса для выполнения задачи.

С параметрами меньше проблем, чем с временными переменными, при условии, что им не присваиваются значения. В противном случае требуется выполнить «Удаление присваиваний параметрам» (Remove Assignments to Parameters).

Когда метод разложен, мне значительно легче понять, как он работает. Иногда обнаруживается возможность улучшения алгоритма, позволяющая сделать его более понятным. Тогда я применяю «Замещение алгоритма» (Substitute Algorithm).

Выделение метода (Extract Method)

Есть фрагмент кода, который можно сгруппировать.

Преобразуйте фрагмент кода в метод, название которого объясняет 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);
}
```

Мотивировка

«Выделение метода» (Extract Method) - один из наиболее часто проводимых мной типов рефакторинга. Я нахожу метод, кажущийся слишком длинным, или код, требующий комментариев, объясняющих его назначение. Тогда я преобразую этот фрагмент кода в отдельный метод.

По ряду причин я предпочитаю короткие методы с осмысленными именами. Во-первых, если выделен мелкий метод, повышается вероятность его использования другими методами. Во-вторых, методы более

высокого уровня начинают выглядеть как ряд комментариев. Замена методов тоже упрощается, когда они мелко структурированы.

Понадобится некоторое время, чтобы привыкнуть к этому, если раньше вы пользовались большими методами. Маленькие методы действительно полезны, если выбирать для них хорошие имена. Иногда меня спрашивают, какой длины должно быть имя метода. Думаю, важна не длина, а семантическое расстояние между именем метода и телом метода. Если выделение метода делает код более понятным, выполните его, даже если имя метода окажется длиннее, чем выделенный код.

Техника

Создайте новый метод и назовите его соответственно назначению метода (тому, что, а не как он делает).

Когда предполагаемый к выделению код очень простой, например, если он выводит отдельное сообщение или вызывает одну функцию, следует выделять его, если имя нового метода лучше раскрывает назначение кода. Если вы не можете придумать более содержа тельное имя, не выделяйте код.

Скопируйте код, подлежащий выделению, из исходного метода в создаваемый.

Найдите в извлеченном коде все обращения к переменным, имеющим локальную область видимости для исходного метода. 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 в создаваемом методе.

Посмотрите, модифицирует ли выделенный код какие-либо из этих переменных с локальной областью видимости. Если модифицируется одна переменная, попробуйте превратить выделенный код в вызов другого метода, результат которого присваивается этой переменной. Если это затруднительно или таких переменных несколько, в существующем виде выделить метод нельзя. Попробуйте сначала выполнить «Расщепление временной переменной» (Split Temporary Variable), а затем снова выделить метод. Временные переменные можно ликвидировать с помощью «Замены временных переменных вызовом методов» (Replace Temp with Query) (см. обсуждение в примерах).

Передайте в создаваемый метод в качестве параметров переменные с локальной областью видимости, чтение которых осуществляется в выделенном коде.

Справившись со всеми локальными переменными, выполните компиляцию.

Замените в исходном методе выделенный код вызовом созданного метода.

Если какие-либо временные переменные перемещены в созданный метод, найдите их объявления вне выделенного кода. Если таковые имеются, можно их удалить.

Выполните компиляцию и тестирование.

Пример: при отсутствии локальных переменных

В простейшем случае «Выделение метода» (Extract Method) выполняется тривиально. Возьмем такой 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 деталей
 System.out.println ("имя  " + _name);
 System.out.println ("сумма" + outstanding);
```

```
    }
```

Код, выводящий баннер, легко выделить с помощью копирования и вставки:

```
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 деталей
 System.out.println ("имя " + _name);
 System.out.println ("сумма " + outstanding);
}
void printBanner() { // вывод баннера
 System.out.println ("****************************");
 System.out.println ("*** Задолженность клиента ***");
 System.out.println ("****************************");
}
```

Пример: с использованием локальных переменных

В чем наша проблема? В локальных переменных - параметрах, передаваемых в исходный метод, и временных переменных, объявленных в исходном методе. Локальные переменные действуют только в исходном методе, поэтому при «Выделении метода» (Extract Method) с ними связана дополнительная работа. В некоторых случаях они даже вообще не позволят выполнить рефакторинг.

Проще всего, когда локальные переменные читаются, но не изменяются. В этом случае можно просто передавать их в качестве параметров. Пусть, например, есть такой 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 деталей
 System.out.println ("имя " + _name);
 System.out.println ("сумма " + outstanding);
}
```

Вывод деталей 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 ("сумма " + outstanding);
   }
```

Такой способ может быть использован с любым числом локальных переменных.

То же применимо, если локальная переменная является объектом и вызывается метод, модифицирующий переменную. В этом случае также можно передать объект в качестве параметра. Другие меры потребуются, только если действительно выполняется присваивание локальной переменной.

Пример: присваивание нового значения локальной переменной

Сложности возникают, когда локальным переменным присваиваются новые значения. В данном случае мы говорим только о временных переменных. Увидев присваивание параметру, нужно сразу применить «Удаление присваиваний параметрам» (Remove Assignments to Parametrs).

Есть два случая присваивания временным переменным. В простейшем случае временная переменная используется лишь внутри выделенного кода. Если это так, временную переменную можно переместить в выделенный код. Другой случай - использование переменной вне этого кода. В этом случае необходимо обеспечить возврат выделенным кодом модифицированного значения переменной. Можно проиллюстрировать это 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 выделяю 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 выделяемом коде, поэтому можно целиком перенести ее в новый метод. Переменная outstanding используется в обоих местах, поэтому выделенный метод должен ее возвратить. После компиляции и тестирования, выполненных вслед за выделением, возвращаемое значение переименовывается в соответствии с обычными соглашениями:

```
  double getOutstanding() {
    Enumeration e = _orders.elements();
    double result = 0.0;
    while (e.hasMoreElements()) {
       Order each = (Order) e.nextElement();
       result += each.getAmount();
    }
    return result;
  }
```

В данном случае переменная outstanding инициализируется только очевидным начальным значением, поэтому достаточно инициализировать ее в выделяемом методе. Если с переменной выполняются какие-либо сложные действия, нужно передать ее последнее значение в качестве параметра. Исходный код для этого случая может выглядеть так:

```
  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 );
  }
```

В данном случае выделение будет выглядеть так:

```
  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 выполнения компиляции и тестирования я делаю более понятным способ инициализации переменной outstanding:

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

Может возникнуть вопрос, что произойдет, если надо возвратить не одну, а несколько переменных.

Здесь есть ряд вариантов. Обычно лучше всего выбрать для выделения другой код. Я предпочитаю, чтобы метод возвращал одно значение, поэтому я бы попытался организовать возврат разных значений разными методами. (Если язык, с которым вы работаете, допускает выходные параметры, можно этим воспользоваться. И по возможности стараюсь работать с одиночными возвращаемыми значениями.)

Часто временных переменных так много, что выделять методы стано вится очень трудно. В таких случаях я пытаюсь сократить число временных переменных с помощью «Замены временной переменной вызовом метода» (Replace Temp with Query). Если и это не помога ет, я прибегаю к «Замене метода объектом методов» (Replace Method with Method Object). Для последнего рефакторинга безразлично, сколько имеется временных переменных и какие действия с ними вы полняются.

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;
}
```

Мотивировка

Длинам книга учит тому, как использовать короткие методы с названиями, отражающими их назначение, что приводит к получению более понятного и легкого для чтения кода. Но иногда встречаются методы, тела которых столь же прозрачны, как названия, либо изначально, либо становятся таковыми в результате рефакторинга. В этом случае необходимо избавиться от метода. Косвенность может быть полезной, но излишняя косвенность раздражает.

Еще один случай для применения «Встраивания метода» (Inline Method) возникает, если есть группа методов, структура которых представляется неудачной. Можно встроить их все в один большой метод, а затем выделить методы иным способом. Кент Бек считает полезным провести такую операцию перед «Заменой метода объектом методов» (Replace Method with Method Object). Надо встроить различные вызовы, выполняемые методом, функции которых желательно поместить в объект методов. Проще переместить один метод, чем перемещать метод вместе с вызываемыми им методами.

Я часто обращаюсь к «Встраиванию метода» (Inline Method), когда в коде слишком много косвенности и оказывается, что каждый метод просто выполняет делегирование другому методу, и во всем этом делегировании можно просто заблудиться. В таких случаях косвенность в какой-то мере оправданна, но не целиком. Путем встраивания удается собрать полезное и удалить все остальное.

Техника

Убедитесь, что метод не является полиморфным.

Избегайте встраивания, если есть подклассы, перегружающие метод; они не смогут перегрузить отсутствующий метод.

Найдите все вызовы метода.

Замените каждый вызов телом метода.

Выполните компиляцию и тестирование.

Удалитеобъявлениеметода.

Так, как оно здесь описано, «Встраивание метода» (Inline Method) выполняется просто, но в целом 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 по той причине, что, встретив такого рода сложности, не следует проводить данный рефакторинг.

Встраивание временной переменной (Inline Temp)

Имеется временная переменная, которой один раз присваивается простое выражение, и эта переменная мешает проведению других рефакторингов. Замените этим выражением все ссылки на данную переменную.

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

```
return (anOrder.basePriced > 1000);
```

Мотивировка

Чаще всего встраивание переменной производится в ходе «Замены временной переменной вызовом метода» (Replace Temp with Query), поэтому подлинную мотивировку следует искать там. Встраивание временной переменной выполняется самостоятельно только тогда, когда обнаруживается временная переменная, которой присваивается значение, возвращаемое вызовом метода. Часто эта переменная безвредна, и можно оставить ее в покое. Но если она мешает другим рефакторингам, например, «Выделению метода» (Extract Method), ее надо встроить.

Техника

Объявите временную переменную с ключевым словом final, если это еще не сделано, и скомпилируйте код.

Так вы убедитесь, что значение этой переменной присваивается действительно один раз.

Найдите все ссылки на временную переменную и замените их правой частью присваивания.

Выполняйте компиляцию и тестирование после каждой модификации.

Удалите объявление и присваивание для данной переменной.

Выполните компиляцию и тестирование.

Замена временной переменной вызовом метода (Replace Temp with Query)

Временная переменная используется для хранения значения выражения.

Преобразуйте выражение в метод. Замените все ссылки на временную переменную вызовом метода. Новый метод может быть использован 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;
}
```

Мотивировка

Проблема с этими переменными в том, что они временные и локальные. Поскольку они видны лишь в контексте метода, в котором используются, временные переменные ведут к увеличению размеров методов, потому что только так можно до них добраться. После замены временной переменной методом запроса получить содержащиеся в ней данные может любой метод класса. Это существенно содействует получению качественного кода для класса.

«Замена временной переменной вызовом метода» (Replace Temp with Query) часто представляет собой необходимый шаг перед «Выделением метода» (Extract Method). Локальные переменные затрудняют выделение, поэтому замените как можно больше переменных вызовами методов.

Простыми случаями данного рефакторинга являются такие, в которых присваивание временным переменным осуществляется однократно, и те, в которых выражение, участвующее в присваивании, свободно от побочных эффектов. Остальные ситуации сложнее, но разрешимы. Облегчить положение могут предварительное «Расщепление временной переменной» (Split Temporary Variable) и «Разделение запроса и модификатора» (Separate Query from Modifier). Если временная переменная служит для накопления результата (например, при суммировании в цикле), соответствующая логика должна быть воспроизведена в методе запроса.

Техника

Рассмотрим простой случай:

Найти простую переменную, для которой присваивание выполняется один раз.

Если установка временной переменной производится несколько раз, попробуйте воспользоваться «Расщеплением временной переменной» (Split Temporary Variable).

Объявите временную переменную с ключевым словом final.

Скомпилируйте код.

Это гарантирует, что присваивание временной переменной выполняется только один раз.

Выделите правую часть присваивания в метод.

Сначала пометьте метод как закрытый (private). Позднее для него может быть найдено дополнительное применение, и тогда защиту будет легко ослабить.

Выделенный метод должен быть свободен от побочных эффектов, т. е. не должен модифицировать какие-либо объекты. Если это не так, воспользуйтесь «Разделением запроса и модификатора» (Separate Query from Modifier).

Выполните компиляцию и тестирование.

Выполните для этой переменной «Замену временной переменной вызовом метода» (Replace Temp with Query).

Временные переменные часто используются для суммирования данных в циклах. Цикл может быть полностью выделен в метод, что позволит избавиться от нескольких строк отвлекающего внимание кода. Иногда в цикле складываются несколько величин. В таком случае повторите цикл отдельно для каждой временной переменной, чтобы иметь возможность заменить ее вызовом метода. Цикл должен быть очень простым, поэтому дублирование кода не опасно.

В данном случае могут возникнуть опасения по поводу снижения производительности. Оставим их пока в стороне, как и другие связанные с ней проблемы. В девяти случаях из десяти они не существенны. Если производительность важна, то она может быть улучшена на этапе оптимизации. Когда код имеет четкую структуру, часто находятся более мощные оптимизирующие решения, которые без рефакторинга остались бы незамеченными. Если дела пойдут совсем плохо, можно легко вернуться к 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 по очереди заменить обе временные переменные.

Хотя в данном случае все достаточно ясно, можно проверить, действительно ли значения присваиваются им 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;
}
```

После этого компилятор сообщит о возможных проблемах. Это первое действие, потому что при возникновении проблем от проведения данного рефакторинга следует воздержаться. Временные переменные заменяются поочередно. Сначала выделяю 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 «Замену временной переменной вызовом метода» (Replace Temp with Query). Сначала заменяется первая ссылка 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;
}
```

Выполните компиляцию и тестирование, потом следующую замену. Поскольку она будет и последней, удаляется объявление временной переменной:

```
double getPrice() {
 final double discountFactor;
 if (basePrice() > 1000) discountFactor = 0.95;
 else discountFactor = 0.98;
 return basePrice() * discountFactor;
}
```

После этого можно аналогичным образом выделить 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, как трудно было бы выделить discountFactor без предварительной замены basePrice вызовом метода. В итоге приходим к следующему виду метода getPrice:

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

Введение поясняющей переменной (Introduce Explaining Variable)

Имеется сложное выражение.

Поместите результат выражения или его части во временную переменную, имя которой поясняете его назначение.

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

```
final boolean isMacOS = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;
final boolean isResized = resize > 0;
if(isMacOS && isIEBrowser && wasInitialized() && isResized) {
 // do something
```

```
}
```

Мотивировка

Выражения могут становиться очень сложными и трудными для чтения. В таких ситуациях полезно с помощью временных переменных превратить выражение в нечто, лучше поддающееся управлению. Особую ценность «Введение поясняющей переменной» (Introduce Explaining Variable) имеет в условной логике, когда удобно для каждого пункта условия объяснить, что он означает, с помощью временной переменной с хорошо подобранным именем. Другим примером служит длинный алгоритм, в котором каждый шаг можно раскрыть с помощью временной переменной.

«Введение поясняющей переменной» (Introduce Explaining Variable) - очень распространенный вид рефакторинга, но должен признаться, что сам я применяю его не очень часто. Почти всегда я предпочитаю «Выделение метода» (Extract Method), если это возможно. Временная переменная полезна только в контексте одного метода Метод же можно использовать всюду в объекте и в других объектах. Однако бывают ситуации, когда локальные переменные затрудняют применение «Выделения метода» (Extract Method). В таких случаях обращаюсь к «Введению поясняющей переменной» (Introduce Explaining Variable).

Техника

Объявите локальную переменную с ключевым словом final и установите ее значением результат части сложного выражения.

Замените часть выражения значением временной переменной.

Если эта часть повторяется, каждое повторение можно заменять поочередно

Выполните компиляцию и тестирование.

Повторите эти действия для других частей выражения.

Пример

Начну с простого вычисления:

```
double price() {
 // price есть базисная цена - скидка по количеству + поставка
 return _quantity * _itemPrice -
    Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
    Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

Простой код, но можно сделать его еще понятнее. Для начала я обо значу количество, умноженное на цену предмета, как базисную цену Эту часть расчета можно превратить во временную переменную:

```
double price() {
 // price есть базисная цена - скидка по количеству + поставка
 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 одного предмета, используется также далее, поэтому и там можно подставить временную переменную:

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

Теперь возьмем скидку в зависимости от количества:

```
double price() {
 // price есть базисная цена - скидка по количеству + поставка
 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 поставкой. После этого можно убрать комментарий, потому что теперь в нем нет ничего, о чем бы не говорил 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 «Выделением метода»

Для подобного случая я не стал бы создавать поясняющие переменные, а предпочел бы «Выделение метода» (Extract Method). Снова начинаю с

```
double price() {
 // price есть базисная цена - скидка по количеству + поставка
 return _quantity * _itemPrice -
    Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
    Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

На этот раз я выделяю метод для базисной цены:

```
double price() {
 // price есть базисная цена - скидка по количеству + поставка
 return basePrice() -
    Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
    Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
private double basePrice() {
 return _quantity * _itemPrice;
}
```

Продолжаю по шагам. В итоге получается:

```
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 «Выделение метода» (Extract Method), поскольку в результате методы становятся доступными в любом другом месте объекта, где они могут понадобиться. Первоначально я делаю их закрытыми, но всегда могу ослабить ограничения, если методы понадобятся другому объекту. По моему, опыту обычно для «Выделения метода» (Extract Method) требуется не больше усилий, чем для «Введения поясняющей переменной» (Introduce Explaining Variable).

Когда же я применяю «Введение поясняющей переменной» (Introduce Explaining Variable)? Тогда, когда «Выделение метода» (Extract Method) действительно требует больше труда. Если идет работа с алгоритмом, использующим множество локальных переменных, «Выделение метода» (Extract Method) может оказаться делом непростым. В такой ситуации я выбираю «Введение поясняющей переменной» (Introduce Explaining Variable), что позволяет мне лучше понять действие алгоритма. Когда логика станет более доступной для понимания, я смогу применить «Замену временной переменной вызовом метода» (Replace Temp with Query). Временная переменная окажется также полезной, если в итоге мне придется прибегнуть к «Замене метода объектом методов» (Replace Method with Method Object).

Расщепление временной переменной (Split Temporary Variable)

Имеется временная переменная, которой неоднократно присваивается значение, но это не переменная цикла и не временная переменная для нак опле н и я результата.

Создайте для каждого присваивания отдельную временную переменную.

```
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);
```

Мотивировка

Временные переменные создаются с различными целями. Иногда эти цели естественным образом приводят к тому, что временной переменной несколько раз присваивается значение. Переменные управления циклом [Beck] изменяются при каждом проходе цикла (например, *i* в *for (int i=0; i<10; i++)*. Накопительные временные переменные [Beck] аккумулируют некоторое значение, получаемое при выполнении метода.

Другие временные переменные часто используются для хранения результата пространного фрагмента кода, чтобы облегчить последующие ссылки на него. Переменным такого рода значение должно присваиваться только один раз. То, что значение присваивается им неоднократно, свидетельствует о выполнении ими в методе нескольких задач Все переменные, выполняющие несколько функций, должны быть менены отдельной переменной для каждой из этих функций. Использование одной и той же переменной для решения разных задач очень затрудняет чтение кода.

Техника

Измените имя временной переменной в ее объявлении и первом присваивании ей значения.

Если последующие присваивания имеют вид *i=i+некоторое_выражение*, то это накопительная временная переменная, и ее расщеплять не надо. С накопительными временными переменными обычно производятся такие действия, как сложение, конкатенация 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 шаги переименования в месте объявления и изменения ссылок вплоть до очередного присваивания.

Пример

В этом примере вычисляется расстояние, на которое перемещается хаггис. В стартовой позиции на хаггис воздействует первоначальная сила. После некоторой задержки вступает в действие вторая сила, придающая дополнительное ускорение. С помощью обычных законов движения можно вычислить 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 * ace * secondaryTime * secondaryTime;
 }
 return result;
}
```

Маленькая и довольно неуклюжая функция. Для нашего примера представляет интерес то, что переменная *acc* устанавливается в ней дважды. Она выполняет две задачи: содержит первоначальное ускорение, вызванное первой силой, и позднее ускорение, вызванное обеими силами. Я намерен ее расщепить.

Начнем с изменения имени переменной и объявления нового имени как final. После этого будут изменены все ссылки на эту временную переменную, начиная с этого места и до следующего присваивания. В месте следующего присваивания переменная будет объявлена:

```
double getDistanceTravelled (int tine) {
 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, представляющее лишь первое применение временной переменной. Я объявил его как final, чтобы гарантировать однократное присваивание ему значения. После этого можно объявить первоначальную переменную в месте второго присваивания ей некоторого значения. Теперь можно выполнить компиляцию и тестирование, и все должно работать.

Последующие действия начинаются со второго присваивания временной переменной. Они окончательно удаляют первоначальное имя переменной, заменяя его новой временной переменной, используемой во второй задаче.

```
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 много других способов рефакторинга. Пожалуйста. (Думаю, это лучше, чем есть хаггис - вы знаете, что в него кладут?)

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;
}
```

Мотивировка

Прежде всего, договоримся о смысле выражения «присваивание параметру». Под этим подразумевается, что если в качестве значения параметра передается объект с именем foo, то присваивание параметру означает, что foo изменится и станет указывать на другой объект. Нет проблем при выполнении каких-то операций над переданным объектом - я делаю это постоянно. Я лишь возражаю против изменения foo, превращающего его в ссылку на совершенно другой объект:

```
void aMethod(Object foo) {
 foo.modifyInSomeWay();// это нормально
 foo = anotherObject;  // последуют всевозможные неприятности
}
```

Причина, по которой мне это не нравится, связана с отсутствием ясности и смешением передачи по значению и передачи по ссылке. В Java используется исключительно передача по значению (см. ниже), и данное изложение основывается именно на этом способе. При передаче по значению изменения параметра не отражаются в вызвавшей программе. Это может смутить тех, кто привык к передаче по ссылке.

Другая область, которая может вызвать замешательство, это само тело кода. Код оказывается значительно понятнее, если параметр используется только для представления того, что передано, поскольку это строгое употребление.

Не присваивайте значений параметрам в Java и, увидев код, который это делает, примените «Удаление присваиваний параметрам» (Remove Assignments to Parametrs).

Конечно, данное правило не является обязательным для других языков, в которых применяются выходные параметры, но даже в таких языках я предпочитаю как можно реже пользоваться выходными параметрами.

Техника

Создайте для параметра временную переменную.

Замените все обращения к параметру, осуществляемые после присваивания, временной переменной.

Измените присваивание так, чтобы оно производилось для временной переменной.

Выполните компиляцию и тестирование.

Если передача параметра осуществляется по ссылке, проверьте, используется ли параметр в вызывающем методе снова. Посмотрите также, сколько в этом методе параметров, передаваемых по ссылке, которым присваивается значение и которые в дальнейшем используются. Попробуйте сделать так, чтобы метод возвращал одно 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 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;
}
```

Принудить к выполнению данного соглашения можно с помощью ключевого слова final:

```
int discount (final int inputVal, final int quantity, final int yearToDate) {
 if (inputVal > 50) result -= 2;
 if (quantity > 100) result -= 1;
 if (yearToDate > 10000) result -= 4;
 return result;
}
```

Признаться, я редко пользуюсь final, поскольку, по моему мнению, для улучшения понятности коротких методов это не много дает. Я прибегаю к этому объявлению в длинных методах, что позволяет мне обнаружить, не изменяется ли где-нибудь параметр.

Передача по значению в Java

Передача по значению, применяемая в Java, часто приводит к путанице. Java строго придерживается передачи по значению, поэтому следующая программа:

```
class Param {
 public static void mam(String[] args) {
    int x = 5;
    triple(x);
    System.out.println(“x after triple” + x);
 }
 private static void triple(int arg) {
```

```
        arg = arg * 3;
        System.out.println (“arg in triple” + arg);
    }
   }
```

выводит такие результаты:

```
arg in triple 15
x after triple 5
```

Неразбериха возникает с объектами. Допустим, создается и затем модифицируется дата в такой программе:

```
class Param {
 public static void main(String[] args) {
    Date d1 = new Date (“1 Apr 98”);
    nextDateUpdate(dl);
    System.out.println (“d1 after nextDay ” + d1);
    Date d2 = new Date (“1 Apr 98”);
    nextDateReplace(d2);
    System.out.println (“d2 after nextDay ” + d2);
 }
 private static void nextDateUpdate (Date arg) {;
    arg.setDate(arg.getDate() + 1);
    System.out.println (“arg in nextDay ” + arg);
 }
 private static void nextDateReplace (Date arg) {
    arg = new Date (arg.getYear(),arg.getMonth(), arg.getDay() + 1);
    System.out.println (“arg in nextDay ” + arg);
 }
}
```

Эта программа выводит следующие результаты:

```
arg in nextDay Thu Apr 02 00 00 00 EST 1998
d1 after nextDay Thu Apr 02 00 00 00 EST 1998
arg in nextDay Thu Apr 02 00 00 00 EST 1998
d2 after nextDay Wed Apr 01 00 00 00 EST 1998
```

Важно, что ссылка на объект передается по значению. Это позволяет модифицировать объект, но повторное присваивание параметру при этом не принимается во внимание.

В Java 1.1 и выше можно пометить параметр как final, что препятствует присваиванию параметру. Однако при этом сохраняется возможность модификации объекта, на который ссылается переменная. Я всегда работаю с параметрами как с константами, но редко помечаю их в таком качестве в списке параметров.

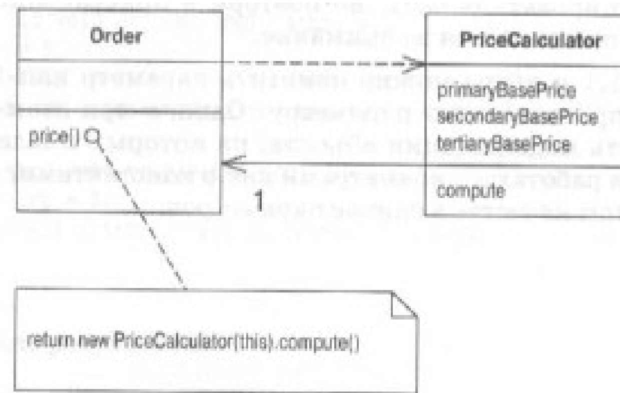
Замена метода объектом методов (Replace Method with Method Object)

Есть длинный метод, в котором локальные переменные используются таким образом, что это не дает применить «Выделение метода».

Преобразуйте метод в отдельный объект так, чтобы локальные переменные стали полями этого объекта. После этого можно разложить данный метод на несколько методов того же объекта.

```
class Order {
 double price() {
    double primaryBasePrice;
    double secondaryBasePrice;
```

```
        double tertiaryBasePnce;
        // длинные ычичиспвния;
     }
   }
```



Мотивировка

В данной книге подчеркивается прелесть небольших методов. Путем выделения в отдельные методы частей большого метода можно сделать код значительно более понятным.

Декомпозицию метода затрудняет наличие локальных переменных. Когда они присутствуют в изобилии, декомпозиция может оказаться сложной задачей. «Замена временной переменной вызовом метода» (Replace Temp with Query) может облегчить ее, но иногда необходимое расщепление метода все же оказывается невозможным. В этом случае нужно поглубже запустить руку в ящик с инструментами и извлечь оттуда объект методов (method object) [Beck].

«Замена метода объектом методов» (Replace Method with Method Object) превращает локальные переменные в поля объекта методов. Затем к новому объекту применяется «Выделение метода» (Extract Method), создающее новые методы, на которые распадается первоначальный метод.

Техника

Бессовестно заимствована у Бека [Beck].

Создайте новый класс и назовите его так же, как метод.

Создайте в новом классе поле с модификатором final для объекта-владельца исходного метода (исходного объекта) 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 таким, который создает новый объект и вызывает compute.

Теперь начинается нечто интересное. Поскольку все локальные переменные стали полями, можно беспрепятственно разложить метод, не нуждаясь при этом в передаче каких-либо параметров.

Пример

Для хорошего примера потребовалась бы целая глава, поэтому я продемонстрирую этот рефакторинг на методе, которому он не нужен. (Не задавайте вопросов 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 т д
        return importantValue3 - 2 * importantValue1;
    }
}
```

Чтобы превратить это в объект метода, я сначала объявляю новый класс. В нем создается константное поле для исходного объекта и поля для всех параметров и временных переменных, присутствующих 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 придерживаюсь соглашения о пометке полей префиксом в виде символа подчеркивания, но, чтобы не забегать вперед, я сейчас оставлю их такими, какие они есть.

Добавим конструктор:

```
Gamma (Account source, int inputValArg, int quantityArg, int yearToDateArg) {
 inputVal = inputValArg;
 quantity = quantityArg;
 yearToDate = yearToDateArg;
}
```

Теперь можно переместить исходный метод. При этом следует модифицировать любые вызовы функций Account так, чтобы они выполнялись через поле _account :

```
int compute () {
 importantValue1 = (inputVal * quantity) + _account.delta();
 importantValue2 = (inputVal * yearToDate) + 100;
 if ((yearToDate - importantValuel) > 100) importantValue2 -= 20;
 int importantValue3 = importantValue2 * 7;
 // и так далее
 return importantValue3 - 2 * importantValue1;
}
```

После этого старый метод модифицируется так, чтобы делегировать выполнение объекту методов:

```
int gamma (int inputVal, int quantity, int yearToDate) {
 return new Gamma(this, inputVal, quantity, yearToDate).compute();
}
```

Вот, в сущности, в чем состоит этот рефакторинг. Преимущество его в том, что теперь можно легко применить «Выделение метода» (Extract Method) к методу compute, не беспокоясь о передаче аргументов:

```
int compute () {
 importantValue1 = (inputVal * quantity) + _account.delta();
 importantValue2 = (inputVal * yearToDate) + 100;
 importantThing();
 int importantValue3 = importantValue2 * 7;
 // and so on.
 return importantValue3 - 2 * importantValue1;
```

```
  }
  void importantThing() {
   if ((yearToDate - importantValue1) > 100) importantValue2 -= 20;
  }
```

Замещение 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].eguals("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 "";
  }
```

Мотивировка

Я никогда не пробовал спать на потолке. Говорят, есть несколько способов. Наверняка одни из них легче, чем другие. С алгоритмами то же самое. Если обнаруживается более понятный способ сделать что-либо, следует заменить сложный способ простым. Рефакторинг позволяет разлагать сложные вещи на более простые части, но иногда наступает такой момент, когда надо взять алгоритм целиком и заменить его чем-либо более простым. Это происходит, когда вы ближе знакомитесь с задачей и обнаруживаете, что можно решить ее более простым способом. Иногда с этим сталкиваешься, начав использовать библиотеку, в которой есть функции, дублирующие имеющийся код.

Иногда, когда желательно изменить алгоритм, чтобы он решал несколько иную задачу, легче сначала заменить его таким кодом, в котором потом проще произвести необходимые изменения.

Перед выполнением этого приема убедитесь, что дальнейшая декомпозиция метода уже невозможна. Замену большого и сложного алгоритма выполнить очень трудно; только после его упрощения замена становится осуществимой.

Техника

Подготовьте свой вариант алгоритма. Добейтесь, чтобы он компилировался.

Прогоните новый алгоритм через свои тесты. Если результаты такие же, как и для старого, на этом следует остановиться.

Если результаты отличаются, используйте старый алгоритм для сравнения при тестировании и отладке.

Выполните каждый контрольный пример со старым и новым алгоритмами и следите за результатами. Это поможет увидеть, с какими контрольными примерами возникают проблемы и какого рода эти проблемы.

7 ПЕРЕМЕЩЕНИЕ ФУНКЦИЙ МЕЖДУ ОБЪЕКТАМИ

Решение о том, где разместить выполняемые функции, является одним из наиболее фундаментальных решений, принимаемых при проектировании объектов. Я работаю с объектами свыше десяти лет, но мне до сих пор не удается сделать правильный выбор с первого раза. Раньше это меня беспокоило, но сейчас я знаю, что рефакторинг позволяет изменить решение.

Часто такие проблемы решаются просто с помощью «Перемещения метода» (Move Method) или «Перемещения поля» (Move Field). Если надо выполнить обе операции, то предпочтительнее начать с «Перемещения поля» (Move Field).

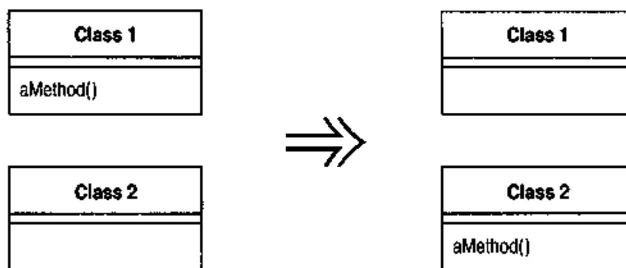
Часто классы перегружены функциями. Тогда я применяю «Выделение класса» (Extract Class), чтобы разделить эти функции на части. Если некоторый класс имеет слишком мало обязанностей, с помощью «Встраивания класса» (Inline Class) я присоединяю его к другому классу. Если функции класса на самом деле выполняются другим классом, часто удобно скрыть этот факт с помощью «Сокрытия делегирования» (Hide Delegate). Иногда сокрытие класса, которому делегируются функции, приводит к постоянным изменениям в интерфейсе класса, и тогда следует воспользоваться «Удалением посредника» (Remove Middle Man).

Два последних рефакторинга, описываемые в этой главе, «Введение внешнего метода» (Introduce Foreign Method) и «Введение локального расширения» (Introduce Local Extension), представляют собой особые случаи. Я выбираю их только тогда, когда мне недоступен исходный код класса, но переместить функции в класс, который я не могу модифицировать, тем не менее, надо. Если таких методов всего один-два, я применяю «Введение внешнего метода» (Introduce Foreign Method), если же методов больше, то пользуюсь «Введением локального расширения» (Introduce Local Extension).

Перемещение метода (Move Method)

Метод чаще использует функции другого класса (или используется ими), а не того, в котором он определен - в данное время или, возможно, в будущем.

Создайте новый метод с аналогичным телом в том классе, который чаще всего им используется. Замените тело прежнего метода простым делегированием или удалите его вообще.



Мотивировка

Перемещение методов - это насущный хлеб рефакторинга. Я перемещаю методы, если в классах сосредоточено слишком много функций или когда классы слишком плотно взаимодействуют друг с другом и слишком тесно связаны. Перемещая методы, можно сделать классы проще и добиться более четкой реализации функций.

Обычно я проглядываю методы класса, пытаясь обнаружить такой, который чаще обращается к другому объекту, чем к тому, в котором сам располагается. Это полезно делать после перемещения каких-либо полей. Найдя подходящий для перемещения метод, я рассматриваю, какие методы вызывают его, какими методами вызывается он сам, и ищу переопределяющие его методы в иерархии классов. Я стараюсь определить, стоит ли продолжить работу, взяв за основу объект, с которым данный метод взаимодействует теснее всего.

Принять решение не всегда просто. Если нет уверенности в необходимости перемещения данного метода, я перехожу к рассмотрению других методов. Часто принять решение об их перемещении проще. Фактически особой разницы нет. Если принять решение трудно, то, вероятно, оно не столь уж важно. Я принимаю то решение, которое подсказывает инстинкт; в конце концов, его всегда можно будет изменить.

Техника

Изучите все функции, используемые исходным методом, которые определены в исходном классе, и определите, не следует ли их так же переместить.

Если некоторая функция используется только тем методом, который вы. собираетесь переместить, ее тоже вполне можно переместить. Если эта функция используется другими методами, посмотрите, нельзя ли и их переместить. Иногда проще переместить сразу группу методов, чем перемещать их по одному.

Проверьте, нет ли в подклассах и родительских классах исходного класса других объявлений метода.

Если есть другие объявления, перемещение может оказаться невозможным, пока полиморфизм также не будет отражен в целевом классе.

Объявите метод в целевом классе.

Можете выбрать для него другое имя, более оправданное для целевого класса.

Скопируйте код из исходного метода в целевой. Приспособьте метод для работы в новом окружении.

Если методу нужен его исходный объект, необходимо определить способ ссылки на него из целевого метода. Если в целевом классе нет соответствующего механизма, передайте новому методу ссылку на исходный объект в качестве параметра.

Если метод содержит обработчики исключительных ситуаций, определите, какому из классов логичнее обрабатывать исключительные ситуации. Если эту функцию следует выполнять исходному классу, оставьте обработчики в нем.

Выполните компиляцию целевого класса.

Определите способ ссылки на нужный целевой объект из исходного.

Поле или метод, представляющие целевой объект, могут уже существовать. Если нет, посмотрите, трудно ли создать для этого метод. При неудаче надо 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 этот объект.

Сделайте из исходного метода делегирующий метод.

Выполните компиляцию и тестирование.

Определите, следует ли удалить исходный метод или сохранить его как делегирующий свои функции.

Проще оставить исходный метод как делегирующий, если есть много ссылок на него.

Если исходный метод удаляется, замените все обращения к нему обращениями к созданному методу.

Выполнять компиляцию и тестирование можно после каждой ссылки, хотя обычно проще заменить все ссылки сразу путем поиска и замены.

Выполните компиляцию и тестирование.

Пример

Данный рефакторинг иллюстрирует класс банковского счета:

```
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 введено несколько новых типов счетов со своими правилами начисления платы за овердрафт (превышение кредита). Я хочу переместить метод начисления этой оплаты в соответствующий тип счета.

Прежде всего, посмотрим, какие функции использует метод overdraftCharge, и решим, следует ли перенести один метод или сразу всю группу. В данном случае надо, чтобы поле _daysOverdrawn осталось в исходном классе, потому что оно будет разным для отдельных счетов.

После этого я копирую тело метода в класс типа счета и подгоняю его по новому месту.

```
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 подгонка означает удаление _type из вызовов функций account и некоторые действия с теми функциями account, которые все же нужны. Если мне требуется некоторая функция исходного класса, можно выбрать один из четырех вариантов: (1) переместить эту функцию в целевой класс, (2) создать ссылку из целевого класса в исходный или воспользоваться уже имеющейся, (3) передать исходный объект в качестве параметра метода, (4) если необходимая функция представляет собой переменную, передать ее в виде параметра.

В данном случае я передал переменную как параметр.

Если метод подогнан и компилируется в целевом классе, можно заменить тело исходного метода простым делегированием:

```
class Account double overdraftCharge() {
 return _type.overdraftCharge(_daysOverdrawn);
}
```

Теперь можно выполнить компиляцию и тестирование.

Можно оставить все в таком виде, а можно удалить метод из исходного класса. Для удаления метода надо найти все места его вызова и выполнить в них переадресацию к методу в классе типа счета:

```
class Account double bankCharge() {
 double result = 4.5;
 if (_daysOverdrawn > 0) {
    result += _type.overdraftCharge(_daysOverdrawn);}
 }
 return result;
}
```

После замены во всех точках вызова можно удалить объявление метода в Account. Можно выполнять компилирование и тестирование после каждого удаления либо сделать это за один прием. Если метод не является закрытым, необходимо посмотреть, не пользуются ли им другие классы. В строго типизированном языке при компиляции после удаления объявления в исходном классе обнаруживается все, что могло быть пропущено.

В данном случае метод обращался к единственному полю, поэтому я смог передать его как переменную. Если бы метод вызывал другой метод класса Account, я не смог бы этого сделать. В таких ситуациях требуется передавать исходный объект:

```
class AccountType {
 double overdraftCharge(Account account) {
    if (isPremium()) {
```

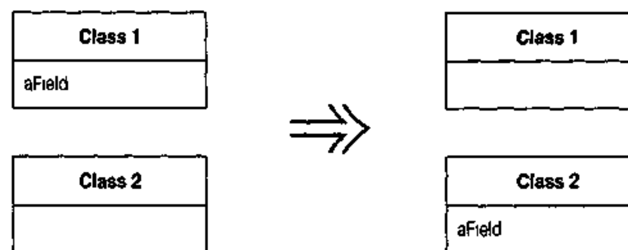
```
        double result = 10;
        if (account.getDaysOverdrawn() > 7) {
          result += (account.getDaysOverdrawn() - 7) * 0.85;
        }
        return result;
      }
      else {
        return account.getDaysOverdrawn() * 1.75;
      }
   }
  }
```

Я также передаю исходный объект, когда мне требуется несколько функций класса, хотя если их оказывается слишком много, требуется дополнительный рефакторинг. Обычно приходится выполнить декомпозицию и вернуть некоторые части обратно.

Перемещение поля (Move Field)

Поле используется или будет использоваться другим классом чаще, чем классом, в котором оно определено.

Создайте в целевом классе новое поле и отредактируйте всех его пользователей.



Мотивировка

Перемещение состояний и поведения между классами составляет самую суть рефакторинга. По мере разработки системы выясняется необходимость в новых классах и перемещении функций между ними. Разумное и правильное проектное решение через неделю может оказаться неправильным. Но проблема не в этом, а в том, чтобы не оставить это без внимания.

Я рассматриваю возможность перемещения поля, если вижу, что его использует больше методов в другом классе, чем в своем собственном. Использование может быть косвенным, через методы доступа. Можно принять решение о перемещении методов, что зависит от интерфейса. Но если представляется разумным оставить методы на своем месте, я перемещаю поле.

Другим основанием для перемещения поля может быть осуществление «Выделения класса» (Extract Class). В этом случае сначала перемещаются поля, а затем методы.

Техника

Если поле открытое, выполните «Инкапсуляцию поля» (Encapsulate Field).

Если вы собираетесь переместить методы, часто обращающиеся к полю, или есть много методов, обращающихся к полю, может оказаться полезным воспользоваться «Самоинкапсуляцией поля» (Self Encapsulate Field).

Выполните компиляцию и тестирование.

Создайте в целевом классе поле с методами для чтения и установки значений.

Скомпилируйте целевой класс.

Определите способ ссылки на целевой объект из исходного.

Целевой класс может быть получен через уже имеющиеся поля или методы. Если нет, посмотрите, трудно ли создать для этого метод. При неудаче 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 этот объект.

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 ссылку 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 которых является interestForAmount_days.

Далее создается 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 для использования целевого класса и удаляю поле процентной ставки из исходного класса. Это поле надо удалить для уверенности в том, что переадресация действительно происходит. Таким образом, 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 есть много методов, которые используют поле процентной ставки, можно начать с применения «Самоинкапсуляции поля» (Self Encapsulate Field):

```
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;
    }
}
```

В результате требуется выполнить переадресацию только для методов доступа:

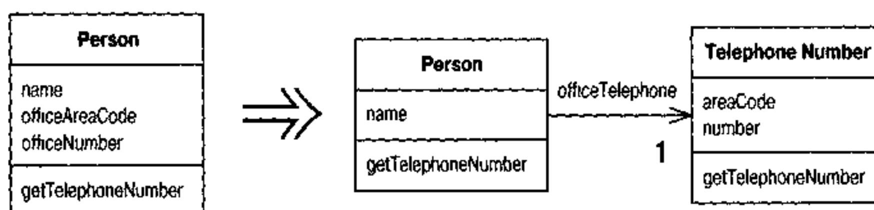
```
double interestForAmountAndDays (double amount, int days) {
 return getInterestRate() * amount * days / 365;
}
private void setInterestRate (double arg) {
 _type.setInterestRate(arg);
}
private double getInterestRate () {
 return _type.getInterestRate();
}
```

Позднее при желании можно выполнить переадресацию для клиентов методов доступа, чтобы они использовали новый объект. Применение самоинкапсуляции позволяет выполнять рефакторинг более мелкими шагами. Это удобно, когда класс подвергается значительной переделке. В частности, упрощается «Перемещение метода» (Move Method) для передислокации методов в другой класс. Если они обращаются к методам доступа, то такие ссылки не требуется изменять.

Выделение класса (Extract Class)

Некоторый класс выполняет работу, которую следует поделить между двумя классами.

Создайте новый класс и переместите соответствующие поля и методы из старого класса в новый.



Мотивировка

Вероятно, вам приходилось слышать или читать, что класс должен представлять собой ясно очерченную абстракцию, выполнять несколько отчетливых обязанностей. На практике классы подвержены разрастанию. То здесь добавится несколько операций, то там появятся новые данные. Добавляя новые функции в класс, вы чувствуете, что для них не стоит заводить отдельный класс, но по мере того, как функции растут и плодятся, класс становится слишком сложным.

Получается класс с множеством методов и кучей данных, который слишком велик для понимания. Вы рассматриваете возможность разделить его на части и делите его. Хорошим признаком является сочетание подмножества данных с подмножеством методов. Другой хороший признак - наличие подмножеств данных, которые обычно совместно изменяются или находятся в особой зависимости друг от друга. Полезно задать себе вопрос о том, что произойдет, если удалить часть данных или метод. Какие другие данные или методы станут бессмысленны?

Одним из признаков, часто проявляющихся в дальнейшем во время разработки, служит характер создания подтипов класса. Может оказаться, что выделение подтипов оказывает воздействие лишь на некоторые функции или что для некоторых функций выделение подтипов производится иначе, чем для других.

Техника

Определите, как будут разделены обязанности класса.

Создайте новый класс, выражающий отделяемые обязанности.

Если обязанности прежнего класса перестают соответствовать его названию, переименуйте его.

Организуйте ссылку из старого класса в новый.

Может потребоваться двусторонняя ссылка, но не создавайте обратную ссылку, пока это не станет необходимо.

Примените «Перемещение поля» (Move Field) ко всем полям, которые желательно переместить.

После каждого перемещения выполните компиляцию и тестирование.

Примените «Перемещение метода» (Move Method) ко всем методам, перемещаемым из старого класса в новый. Начните с методов более низкого уровня (вызываемых, а не вызывающих) и наращивайте их до более высокого уровня.

После каждого перемещения выполняйте компиляцию и тестирование.

Пересмотрите интерфейсы каждого класса и сократите их.

Создав двустороннюю ссылку, посмотрите, нельзя ли превратить ее в одностороннюю.

Определите, должен ли новый класс быть выставлен наружу. Если да, то решите, как это должно быть сделано - в виде объекта ссылки или объекта с неизменяемым значением.

Пример

Начну с простого класса, описывающего личность:

```
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;
}
```

В данном примере можно выделить в отдельный класс функции, относящиеся к телефонным номерам. Начну с определения класса телефонного номера:

```
class TelephoneNumber {
}
```

Это было просто! Затем создается ссылка из класса Person в класс телефонного номера:

```
class Person {
 private TelephoneNumber _officeTelephone = new TelephoneNumber();
}
```

Теперь применяю «Перемещение поля» (Move Field) 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 можно перенести другое поле и применить «Перемещение метода» (Move Method) 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;
   }
```

После этого необходимо решить, в какой мере сделать новый класс доступным для клиентов. Можно полностью скрыть класс, создав для интерфейса делегирующие методы, а можно сделать класс открытым. Можно открыть класс лишь для некоторых клиентов (например, находящихся в моем пакете).

Решив сделать класс общедоступным, следует учесть опасности, связанные со ссылками. Как отнестись к тому, что при открытии телефонного номера клиент может изменить код зоны? Такое изменение может произвести косвенный клиент - клиент клиента клиента.

Возможны следующие варианты:

1. Допускается изменение любым объектом любой части телефонного номера. В таком случае телефонный номер становится объектом ссылки, и следует рассмотреть «Замену значения ссылкой» (Change Value to Reference). Доступ к телефонному номеру осуществляется через экземпляр класса Person.

2 Я не желаю, чтобы кто-либо мог изменить телефонный номер, кроме как посредством методов экземпляра класса Person. Можно сделать неизменяемым телефонный номер или обеспечить неизменяемый интерфейс к телефонному номеру.

3 Существует также возможность клонировать телефонный номер перед тем, как предоставить его, но это может привести к недоразумениям, потому что люди будут думать, что могут изменить его значение. При этом могут также возникнуть проблемы со ссылками у клиентов, если телефонный номер часто передается.

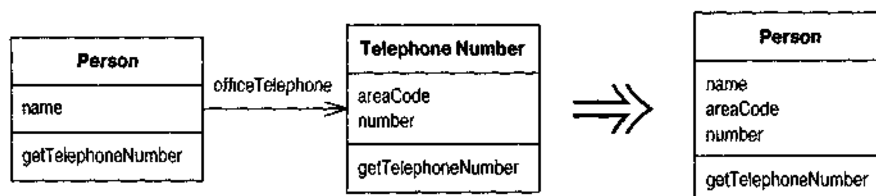
«Выделение класса» (Extract Class) часто используется для повышения живучести параллельной программы, поскольку позволяет устанавливать раздельные блокировки для двух получаемых классов. Если не требуется блокировать оба объекта, то и необязательно делать это. Подробнее об этом сказано в разделе 3.3 у Ли [Lea].

Однако здесь возникает опасность. Если необходимо обеспечить совместную блокировку обоих объектов, вы попадаете в сферу транзакций и другого рода совместных блокировок. Как описывается у Ли [Lea] в разделе 8.1, это сложная область, требующая более мощного аппарата, применение которого обычно мало оправданно. Применять транзакции очень полезно, но большинству программистов не следовало бы браться за написание менеджеров транзакций.

Встраивание класса (Inline Class)

Класс выполняет слишком мало функций.

Переместите все функции в другой класс и удалите исходный.



Мотивировка

«Встраивание класса» (Inline Class) противоположно «Выделению класса» (Extract Class). Я прибегаю к этой операции, если от класса становится мало пользы и его надо убрать. Часто это происходит в результате рефакторинга, оставившего в классе мало функций.

В этом случае следует вставить данный класс в другой, выбрав для этого такой класс, который чаще всего его использует.

Техника

Объявите открытый протокол исходного класса в классе, который его поглотит. Делегируйте все эти методы исходному классу.

Если для методов исходного класса имеет смысл отдельный интерфейс, выполните перед встраиванием «Выделение интерфейса» (Extract Interface)

Перенесите все ссылки из исходного класса в поглощающий класс.

Объявите исходный класс закрытым, чтобы удалить ссылки из запределов пакета Поменяйте также имя исходного класса, чтобы компилятор перехватил повисшие ссылки на исходный класс.

Выполните компиляцию и тестирование.

С помощью «Перемещения метода» (Move Method) и «Перемещения поля» (Move Field) перемещайте функции одну за другой из исходного класса, пока в нем ничего не останется.

Отслужите короткую и скромную поминальную службу.

Пример

Создав класс из телефонного номера, я теперь хочу встроить 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(Stnng arg) {
    officeTelephone.setNumber(arg);
  }
}
```

Теперь найду клиентов телефонного номера и переведу их на использование интерфейса класса Person. Поэтому

```
Person martin = new Person();
Martin.getOfficeTelephone().setAreaCode ("781");
```

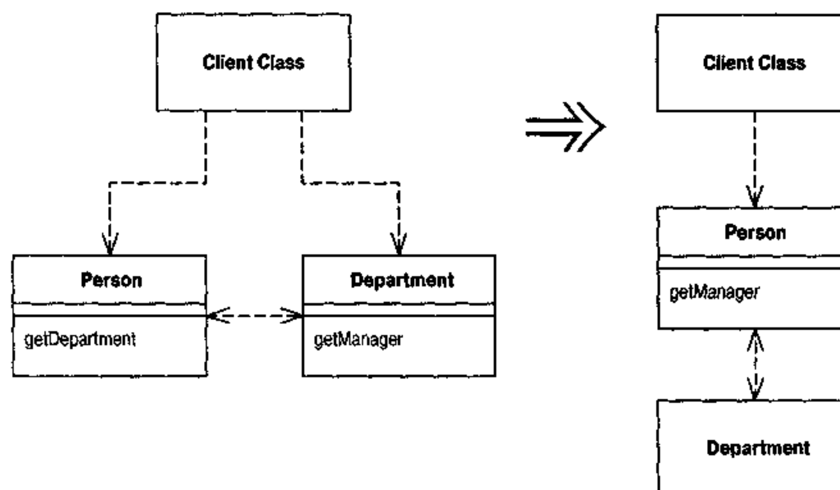
превращается в

```
Person martin = new Person();
Martin.setAreaCode ("781");
```

Теперь можно повторять «Перемещение метода» (Move Method) и «Перемещение поля» (Move Field), пока от класса телефонного номера ничего не останется.

Сокрытие делегирования (Hide Delegate)

Клиент обращается к делегируемому классу объекта. Создайте на сервере методы, скрывающие делегирование.



Мотивировка

Одним из ключевых свойств объектов является инкапсуляция. Инкапсуляция означает, что объектам приходится меньше знать о других частях системы. В результате при модификации других частей об этом требуется сообщить меньшему числу объектов, что упрощает внесение изменений.

Всякий, кто занимался объектами, знает, что поля следует скрывать, несмотря на то что Java позволяет делать поля открытыми. По мере роста искушенности в объектах появляется понимание того, что инкапсулировать можно более широкий круг вещей.

Если клиент вызывает метод, определенный над одним из полей объекта-сервера, ему должен быть известен соответствующий делегированный объект. Если изменяется делегированный объект, может потребоваться модификация клиента. От этой зависимости можно избавиться, поместив в сервер простой делегирующий метод, который скрывает делегирование (рисунок 7.1). Тогда изменения ограничиваются сервером и не распространяются на клиента.

Может оказаться полезным применить «Выделение класса» (Extract Class) к некоторым или всем клиентам сервера. Если скрыть делегирование от всех клиентов, можно убрать всякое упоминание о нем из интерфейса сервера.

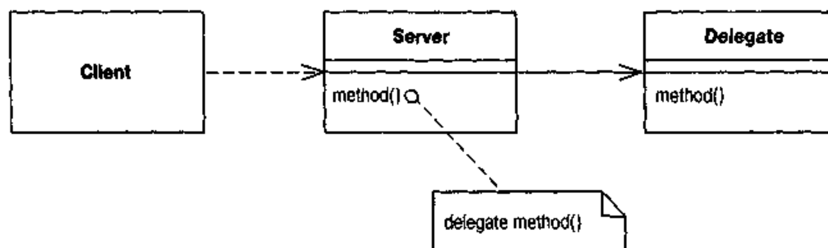


Рисунок 7.1 – Простое делегирование

Техника

Для каждого метода класса-делегата создайте простой делегирующий метод сервера.

Модифицируйте 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 доступ к делегату больше не нужен никаким клиентам, уберите из сервера метод доступа 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 менеджером некоторого лица, он должен сначала узнать, в каком отделе это лицо работает:

```
manager = john.getDepartment().getManager();
```

Так клиенту открывается характер работы класса Department и то, что в нем хранятся данные о менеджере. Эту связь можно сократить, скрыв от клиента класс Department. Это осуществляется путем создания простого делегирующего метода в Person:

```
public Person getManager() {
 return _department.getManager();
}
```

Теперь необходимо модифицировать всех клиентов Person, чтобы они использовали новый метод:

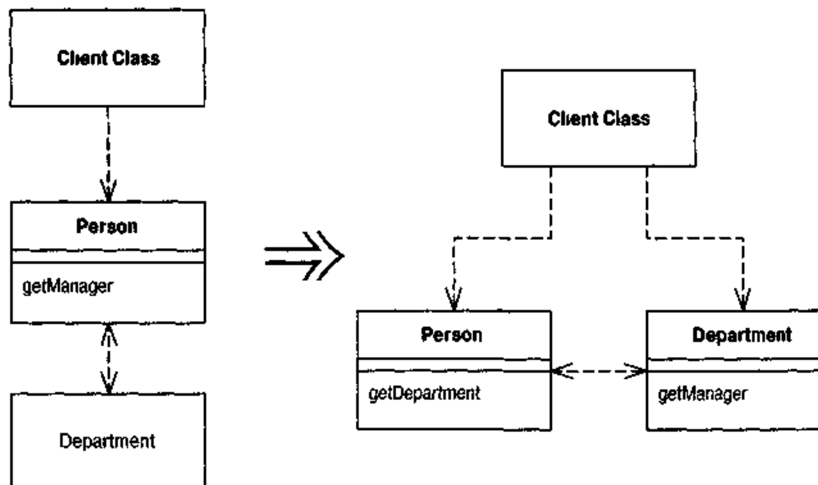
```
manager = john.getManager();
```

Проведя изменения для всех методов Department и всех клиентов Person, можно удалить метод доступа getDepartment из Person.

Удаление посредника (Remove Middle Man)

Класс слишком занят простым делегированием.

Заставьте клиента обращаться к делегату непосредственно.



Мотивировка

В мотивировке «Сокрытия делегирования» (Hide Delegate) я отмечал преимущества инкапсуляции применения делегируемого объекта. Однако есть и неудобства, связанные с тем, что при желании клиента использовать новую функцию делегата необходимо добавить в сервер простой делегирующий метод. Добавление достаточно большого количества функций оказывается утомительным. Класс сервера становится просто посредником, и может настать момент, когда клиенту лучше непосредственно обращаться к делегату.

Сказать точно, какой должна быть мера сокрытия делегирования, трудно. К счастью, это не столь важно благодаря наличию «Сокрытия делегирования» (Hide Delegate) и «Удаления посредника» (Remove Middle Man). Можно осуществлять настройку системы по мере надобности. По мере развития системы меняется и отношение к тому, что должно быть скрыто. Инкапсуляция, удовлетворявшая полгода назад, может оказаться неудобной в настоящий момент. Смысл рефакторинга в том, что надо не раскаиваться, а просто внести необходимые исправления.

Техника

Создайте метод доступа к делегату.

Для каждого случая использования клиентом метода делегата удалите этот метод с сервера и замените его вызов в клиенте вызовом метода делегата.

После обработки каждого метода выполняйте компиляцию и тестирование.

Пример

Для примера я воспользуюсь классами Person и Department, повернув все в обратную сторону. Начнем с Person, скрывающего делегирование Department:

```
class Person {
 Department _department;
 public Person getManager() {
    return _department.getManager();
 }
}
class Department {
 private Person _manager;
 public Department (Person manager) {
    _manager = manager;
 }
}
```

Чтобы узнать, кто является менеджером некоторого лица, клиент делает запрос:

```
manager = john.getManager();
```

Это простая конструкция, инкапсулирующая отдел (Department). Однако при наличии множества методов, устроенных таким образом, в классе Person оказывается слишком много простых делегирований. Тогда посредника лучше удалить. Сначала создается метод доступа к делегату:

```
class Person.{
 public Department getDepartment() {
    return _department;
 }
}
```

После этого я по очереди рассматриваю каждый метод. Я нахожу клиентов, использующих этот метод Person, и изменяю их так, чтобы они сначала получали класс-делегат:

```
manager = john.getDepartment().getManager();
```

После этого можно убрать getManager из Person. Компиляция покажет, не было ли чего-либо упущено.

Может оказаться удобным сохранить некоторые из делегирований. Возможно, следует скрыть делегирование от одних клиентов, но показать другим. В этом случае также надо оставить некоторые простые делегирования,

Введение внешнего метода (Introduce Foreign Method)

Необходимо ввести в сервер дополнительный метод, но отсутствует возможность модификации класса.

Создайте в классе клиента метод, которому в качестве первого аргумента передается класс сервера.

```
Date newStart = new Date (previousEnd.getYear(),
    previousEnd.getMonth(), previousEnd.getDate()+ 1;
```

```
Date newStart = nextDay(previousEnd);
static Date nextDay(Date arg) {
 return new Date (arg.getYear(), arg.getMonth(), arg.getDate() + 1);
}
```

Мотивировка

Ситуация достаточно распространенная. Есть прекрасный класс с отличными сервисами. Затем оказывается, что нужен еще один сервис, но класс его не предоставляет. Мы клянем класс со словами: «Ну почему же ты этого не делаешь?» Если есть возможность модифицировать исходный код, вводится новый метод. Если такой возможности нет, приходится обходными путями программировать отсутствующий метод в клиенте.

Если клиентский класс использует этот метод единственный раз, то дополнительное кодирование не представляет больших проблем и, возможно, даже не требовалось для исходного класса. Однако если метод используется многократно, приходится повторять кодирование снова и снова. Повторение кода - корень всех зол в программах, поэтому повторяющийся код необходимо выделить в отдельный метод. При проведении этого рефакторинга можно явно известить о том, что этот метод в действительности должен находиться в исходном классе, сделав его внешним методом.

Если обнаруживается, что для класса сервера создается много внешних методов или что многим классам требуется один и тот же внешний метод, то следует применить другой рефакторинг - «Введение локального расширения» (Introduce Local Extension).

Не забывайте о том, что внешние методы являются искусственным приемом. По возможности старайтесь перемещать методы туда, где им надлежит находиться. Если проблема связана с правами владельца кода, отправьте внешний метод владельцу класса сервера и попросите его реализовать метод для вас.

Техника

Создайте в классе клиента метод, выполняющий нужные вам действия.

Создаваемый метод не должен обращаться к каким-либо характеристикам клиентского класса. 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 должен располагаться на сервере.

Благодаря этому вы сможете позднее, если появится возможность переместить метод, найти внешние методы, с помощью текстового поиска.

Пример

Есть некий код, в котором нужно открыть новый период выставления счетов. Первоначально код выглядит так:

```
Date newStart = new Date (previousEnd.getYear(),
    previousEnd.getMonth(), previousEnd.getDate()+ 1;
```

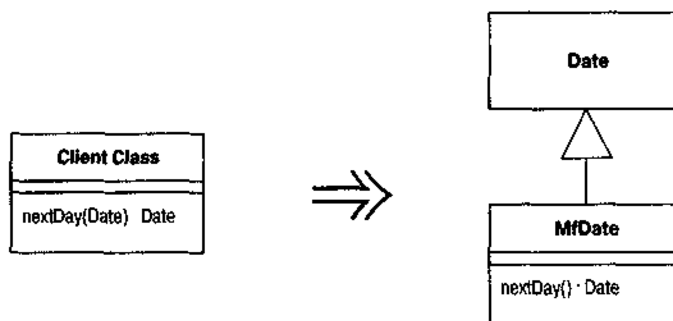
Код в правой части присваивания можно выделить в метод, который будет внешним для Date:

```
Date newStart = nextDay(previousEnd);
static Date nextDay(Date arg) { // внешний метод, должен быть в date
 return new Date (arg.getYear(), arg.getMonth(), arg.getDate() + 1);
}
```

Введение локального расширения (Introduce Local Extension)

Используемый класс сервера требуется дополнить несколькими методами, но класс недоступен для модификации.

Создайте новый класс с необходимыми дополнительными методами. Сделайте его подклассом или оболочкой для исходного класса.



Мотивировка

К сожалению, создатели классов не всеведущи и не могут предоставить все необходимые методы. Если есть возможность модифицировать исходный код, то часто лучше всего добавить в него новые методы. Однако иногда исходный код нельзя модифицировать. Если нужны лишь один-два новых метода, можно применить «Введение внешнего метода» (Introduce Foreign Method). Однако если их больше, они выходят из-под контроля, поэтому необходимо объединить их выбрав для этого подходящее место. Очевидным способом является стандартная объектно-ориентированная технология создания подклассов и оболочек. В таких ситуациях я называю подкласс или оболочку локальным расширением.

Локальное расширение представляет собой отдельный класс, но выделенный в подтип класса, расширением которого он является. Это означает, что он умеет делать то же самое, что и исходный класс, но при этом имеет дополнительные функции. Вместо работы с исходным классом следует создать экземпляр локального расширения и пользоваться им.

Применяя локальное расширение, вы поддерживаете принцип упаковки методов и данных в виде правильно сформированных блоков. Если же продолжить размещение в других классах кода, который должен располагаться в расширении, это приведет к усложнению других классов и затруднению повторного использования этих методов.

При выборе подкласса или оболочки я обычно склоняюсь к применению подкласса, поскольку это связано с меньшим объемом работы. Самое большое препятствие на пути использования подклассов заключается в том, что они должны применяться на этапе создания объектов. Если есть возможность управлять процессом создания, проблем не возникает. Они появляются, если локальное расширение необходимо применять позднее. При работе с подклассами приходится создавать новый объект данного подкласса. Если есть другие объекты, ссылающиеся на старый объект, то появляются два объекта, содержащие данные оригинала. Если оригинал неизменяемый, то проблем не возникает, т. к. можно благополучно воспользоваться копией. Однако если оригинал может изменяться, то возникает проблема, поскольку изменения одного объекта не отражаются в другом. В этом случае надо применить оболочку, тогда изменения, осуществляемые через локальное расширение, воздействуют на исходный объект, и наоборот.

Техника

Создайте класс расширения в виде подкласса или оболочки оригинала.

Добавьте к расширению конвертирующие конструкторы.

Конструктор принимает в качестве аргумента оригинал. В варианте с подклассом вызывается соответствующий конструктор родительского класса; в варианте с оболочкой аргумент присваивается полю для делегирования.

Поместите в расширение новые функции.

В нужных местах замените оригинал расширением.

Если есть внешние методы, определенные для этого класса, переместите их в расширение.

Примеры

Мне часто приходилось заниматься такими вещами при работе с Java 1.0.1 и классом даты. С классом календаря в Java 1.1 появилось многое из того, что мне требовалось, но пока его не было, предоставлялась масса возможностей использовать расширения. Этот опыт нам пригодится здесь в качестве примера.

Сначала надо решить, что выбрать- подкласс или оболочку. Более очевиден путь применения подклассов:

```
Class mfDate extends Date {
 public nextDay();
 public dayOfYear();
```

```
}
```

В оболочке используется делегирование:

```
class mfDate {
 private Date _onginal;
}
```

Пример: использование подкласса

Сначала создается новая дата как подкласс оригинала:

```
class MfDateSub extends Date
```

Затем осуществляется замена дат расширением. Надо повторить конструкторы оригинала путем простого делегирования:

```
public MfDateSub (String dateString) {
 super (dateString);
};
```

Теперь добавляется конвертирующий конструктор, принимающий оригинал в качестве аргумента:

```
public MfDateSub (Date arg) {
 super (arg.getTime());
}
```

Затем можно добавить в расширение новые функции и с помощью «Перемещения метода» (Move Method) перенести в расширение имеющиеся внешние методы:

```
client class {
 private static Date nextDay(Date arg) {
    // внешний метод должен быть в 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

Начну с объявления класса-оболочки:

```
class mfDate {
 private Date _onginal;
}
```

При подходе с созданием оболочки конструкторы организуются иначе. Исходные конструкторы реализуются путем простого делегирования:

```
public MfDateWrap (String dateString) {
 _original = new Date(dateString);
}
```

Конвертирующий конструктор теперь просто устанавливает переменную экземпляра:

```
public MfDateWrap (Date arg) {
 _original = arg;
}
```

Затем выполняется скучная работа делегирования 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()));
}
```

Покончив с этим, можно с помощью «Перемещения метода» (Move Method) ввести в новый класс специальные функции для дат:

```
client class {
 private static Date nextDay(Date arg) {
    // внешний метод, должен быть в 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 вызывают методы, принимающие оригинал в качестве аргумента, например:

```
public boolean after (Date arg)
```

Поскольку оригинал изменить нельзя, для after есть только одно направление:

```
aWrapper.after(aDate); // можно заставить работать
aWrapper.after(anotherWrapper); // можно заставить работать
aDate.after(aWrapper); // не будет работать
```

Цель такой замены - скрыть от пользователя класса тот факт, что применяется оболочка. Это правильная политика, потому что пользователю оболочки должно быть безразлично, что это оболочка, и у него должна быть возможность работать с обоими классами одинаково. Однако полностью скрыть эту информацию я не могу. Проблема вызывается некоторыми системными методами, например equals. Казалось бы, в идеале можно заменить equals в MfDateWrap:

```
public boolean equals (Date arg); // возможны проблемы
```

Это опасно, поскольку хотя это и можно адаптировать к своим задачам, но в других частях Java-системы предполагается, что равенство симметрично, т.е. если a.equals(b), то b.equals(a). При нарушении данного правила возникает целый ряд странных ошибок. Единствен ный способ избежать их - модифицировать Date, но если бы можно бы ло это сделать, не потребовалось бы и проведение данного рефакторинга. Поэтому в таких ситуациях приходится просто обнажить тот факт, что применяется оболочка. Для проверки равенства приходится выбрать метод с другим именем:

```
public boolean equalsDate (Date arg);
```

Можно избежать проверки типа неизвестного объекта, предоставив версии данного метода как для Date, так и для MfDateWrap:

```
public boolean equalsDate (MfDateWrap arg);
```

Такая проблема не возникает при работе с подклассами, если не замещать операцию. Если же делать замену, возникает полная путаница с поиском метода. Обычно я не замещаю методы в расширениях, а только добавляю их.

8 ОРГАНИЗАЦИЯ ДАННЫХ

В данной главе обсуждаются некоторые методы рефакторинга, облегчающие работу с данными. Многие считают «Самоинкапсуляцию поля» (Self Encapsulate Field) излишней. Уже давно ведутся спокойные дебаты о том, должен ли объект осуществлять доступ к собственным данным непосредственно или через методы доступа (accessors). Иногда методы доступа нужны, и тогда можно получить их с помощью «Самоинкапсуляции поля» (Self Encapsulate Field). Обычно я выбираю непосредственный доступ, потому что если мне понадобится такой рефакторинг, его будет легко выполнить.

Одно из удобств объектных языков состоит в том, что они разрешают определять новые типы, которые позволяют делать больше, чем простые типы данных обычных языков. Однако необходимо некоторое время, чтобы научиться с этим работать. Часто мы сначала используем простое значение данных и лишь потом понимаем, что объект был бы удобнее. «Замена значения данных объектом» (Replace Data Value with Object) позволяет превратить бессловесные данные в членораздельно говорящие объекты. Поняв, что эти объекты представляют собой экземпляры, которые понадобятся во многих местах программы, можно превратить их в объекты ссылки посредством «Замены значения ссылкой» (Change Value to Reference).

Если видно, что объект выступает как структура данных, можно сделать эту структуру данных более понятной с помощью «Замены массива объектом» (Replace Array with Object). Во всех этих случаях объект представляет собой лишь первый шаг. Реальные преимущества достигаются при «Перемещении метода» (Move Method), позволяющем добавить поведение в новые объекты.

Магические числа - числа с особым значением - давно представляют собой проблему. Когда я еще только начинал программировать, меня предупреждали, чтобы я ими не пользовался. Однако они продолжают появляться, и я применяю «Замену магического числа символической константой» (Replace Magic Number with Symbolic Constant), чтобы избавляться от них, как только становится понятно, что они делают.

Ссылки между объектами могут быть односторонними или двусторонними. Односторонние ссылки проще, но иногда для поддержки новой функции нужна «Замена однонаправленной связи двунаправленной» (Change Unidirectional Association to Bidirectional). «Замена двунаправленной связи однонаправленной» (Change Bidirectional Associa tion to Unidirectional) устраняет излишнюю сложность, когда обнаруживается, что двунаправленная ссылка больше не нужна.

Я часто сталкивался с ситуациями, когда классы GUI выполняют бизнес-логику, которая не должна на них возлагаться. Для перемещения поведения в классы надлежащей предметной области необходимо хранить данные в классе предметной области и поддерживать GUI с помощью «Дублирования видимых данных» (Duplicate Observed Data). Обычно я против дублирования данных, но этот случай составляет исключение.

Одним из главных принципов объектно-ориентированного программирования является инкапсуляция. Если какие-либо открытые данные повсюду демонстрируют себя, то с помощью «Инкапсуляции поля» (Encapsulate Field) можно обеспечить им достойное прикрытие. Если данные представляют собой коллекцию, следует воспользоваться «Инкапсуляцией коллекции» (Encapsulate Collection), поскольку у нее особый протокол. Если обнажена целая запись, выберите «Замену записи классом данных» (Replace Record with Data Class).

Особого обращения требует такой вид данных, как код типа - специальное значение, обозначающее какую-либо особенность типа экземпляра. Часто эти данные представляются перечислением, иногда реализуемым как статические неизменяемые целые числа. Если код предназначен для информации и не меняет поведения класса, можно прибегнуть к «Замене кода типа классом» (Replace Type Code with Class), что ведет к лучшей проверке типа и создает основу для перемещения поведения в дальнейшем. Если код типа оказывает влияние на поведение класса, можно попытаться применить «Замену кода типа подклассами» (Replace Type Code with Subclasses). Если это не удается, возможно, подойдет более сложная (но более гибкая) «Замена кода типа состоянием/стратегией» (Replace Type Code with State/ Strategy).

Самоинкапсуляция поля (Self Encapsulate Field)

Обращение к полю осуществляется непосредственно, но взаимодействие с полем становится затруднительным.

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;}
```

Мотивировка

Что касается обращения к полям, то здесь существуют две школы. Первая из них учит, что в том классе, где определена переменная, обращаться к ней следует свободно (непосредственный доступ к переменной). Другая школа утверждает, что даже внутри этого класса следует всегда применять методы доступа (косвенный доступ к переменной). Между двумя школами возникают горячие дискуссии. (См. также обсуждение в работах Ауэра [Auer] и Бека [Beck].)

В сущности, преимущество косвенного доступа к переменной состоит в том, что он позволяет переопределить в подклассе метод получения информации и обеспечивает большую гибкость в управлении данными, например отложенную инициализацию (lazy initialization), при которой переменная инициализируется лишь при необходимости ее использования.

Преимущество прямого доступа к переменной заключается в легкости чтения кода. Не надо останавливаться с мыслью: «да это просто метод получения значения переменной».

У меня двойственный взгляд на эту проблему выбора. Обычно я готов действовать так, как считают нужным остальные участники команды. Однако, действуя в одиночку, я предпочитаю сначала применять непосредственный доступ к переменной, пока это не становится препятствием. При возникновении неудобств я перехожу на косвенный доступ к переменным. Рефакторинг предоставляет свободу изменить свое решение.

Важнейший повод применить «Самоинкапсуляцию поля» (Self Encapsulate Field) возникает, когда при доступе к полю родительского класса необходимо заменить обращение к переменной вычислением значения в подклассе. Первым шагом для этого является самоинкапсуляция поля. После этого можно заместить методы получения и установки значения теми, которые необходимы.

Техника

Создайте для поля методы получения и установки значения.

Найдите все ссылки на поле 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.

Измените модификатор видимости поля на private.

Проверьте, все ли ссылки найдены и заменены.

Выполните компиляцию и тестирование.

Пример

Возможно, следующий код слишком прост для примера, но по крайней мере не потребовалось много времени для его 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 их 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 осуществлении самоинкапсуляции необходимо проявлять осторожность, используя метод установки значения в конструкторе. Часто предполагается, что метод установки применяется уже после создания объекта, поэтому его поведение может быть иным, чем во время инициализации. В таких случаях я предпочитаю в конструкторе непосредственный доступ или отдельный метод инициализации:

```
IntRange (int low, int high) {
 initialize (low, high);
}
private void initialize (int low, int high) {
 _low = low;
 _high = high;
}
```

Значение этих действий проявляется, когда создается подкласс, например:

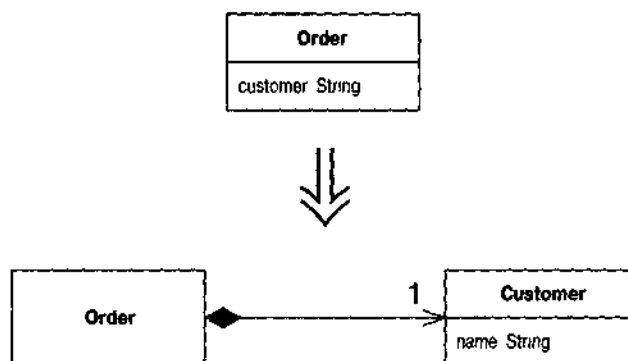
```
class CappedRange extends IntRange {
 CappedRange (int low, int high, int cap) {
    super (low, high);
    _cap = cap;
 }
 private int _cap;
 int getCap() {
    return _cap;
 }
 int getHigh() {
    return Math.min(super.getHigh(), getCap());
 }
}
```

Можно целиком заместить поведение IntRange так, чтобы оно учитывало cap, не модифицируя при этом поведение исходного класса.

Замена значения данных объектом (Replace Data Value with Object)

Есть некоторый элемент данных, для которого требуются дополнительные данные или поведение.

Преобразуйте элемент данных в объект.



Мотивировка

Часто на ранних стадиях разработки принимается решение о представлении простых фактов в виде простых элементов данных. В процессе разработки обнаруживается, что эти простые элементы в действительности сложнее. Телефонный номер может быть вначале представлен в виде строки, но позднее выясняется, что у него должно быть особое поведение в виде форматирования, извлечения кода зоны и т. п. Если таких элементов один-два, можно поместить соответствующие методы в объект, который ими владеет, однако код быстро начинает попахивать дублированием и завистью к функциям. Если такой душок появился, преобразуйте данные в объект.

Техника

Создайте класс для данных. Предусмотрите в нем неизменяемое поле того же типа, что и поле данных 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 в новом классе.

Если поле участвует 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 выполнить «Замену значения ссылкой» (Change Value to Reference).

Пример

Начну с примера класса заказа (order), в котором клиент (customer) хранится в виде строки, а затем преобразую клиента в объект, чтобы хранить такие данные, как адрес или оценку кредитоспособности, и полезные методы 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 некоторый клиентский код, использующий 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 создам новый класс клиента. В нем будет иметься константное поле для строкового атрибута, поскольку это то, что используется в настоящий момент в заказе. Я назову его name, ведь эта строка, очевидно, содержит имя клиента. Я также создам метод получения значения 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 customer и методы, которые ссылаются на него, чтобы они использовали правильные ссылки на класс Customer, Метод получения значения и конструктор очевидны. Метод установки значения создает новый Customer:

```
class Order {
 public Order (String customer) {
   _customer = new Customer(customer);
 }
 public String getCustomer() {
   return _customer.getName();
}
 public void setCustomer(String arg) {
   _customer = new Customer(arg);
 }
 private Customer _customer;
}
```

Метод установки данных создает новый Customer потому, что прежний строковый атрибут был объектом данных, и в настоящее время Customer тоже является объектом данных. Это означает, что в каждом заказе собственный объект Customer. Как правило, объекты данных должны быть неизменяемыми, благодаря чему удается избежать некоторых неприятных ошибок, связанных со ссылками. Позднее мне потребуется, чтобы Customer стал объектной ссылкой, но это произойдет в результате другого рефакторинга. В данный момент можно выполнить компиляцию и тестирование.

Теперь я рассмотрю методы Order, работающие с Customer, и произведу некоторые изменения, призванные прояснить новое положение вещей. К методу получения значения я применю «Переименование метода» (Rename Method), чтобы стало ясно, что он возвращает имя, а не объект:

```
public String getCustomerName() {
 return _customer.getName();
}
```

Для конструктора и метода установки значения не требуется изменять сигнатуру, но имена аргументов должны быть изменены:

```
public Order (String customerName) {
 _customer = new Customer(customerName);
}
public void setCustomer(String customerName) {
 _customer = new Customer(customerName);
}
```

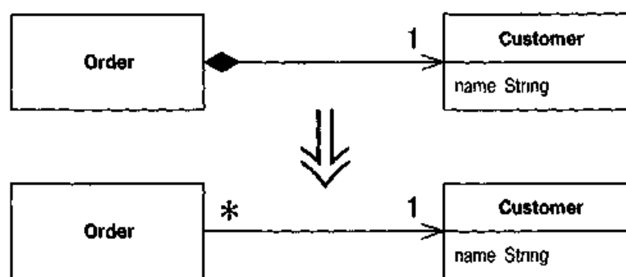
При проведении следующего рефакторинга может потребоваться добавление нового конструктора и метода установки, которому передается существующий Customer.

На этом данный рефакторинг завершается, но здесь, как и во многих других случаях, надо сделать еще одну вещь. Если требуется добавить к клиенту оценку кредитоспособности, адрес и т. п., то в настоящий момент сделать это нельзя, потому что Customer действует как объект данных. В каждом заказе собственный объект Customer. Чтобы создать в Customer требуемые атрибуты, необходимо применить к Customer «Замену значения ссылкой» (Change Value to Reference), чтобы все заказы для одного и того же клиента использовали один и тот же объект Customer. Данный пример будет продолжен в описании этого рефакторинга.

Замена значения ссылкой (Change Value to Reference)

Есть много одинаковых экземпляров одного класса, которые требуется заменить одним объектом.

Превратите объект в объект ссылки.



Мотивировка

Во многих системах можно провести полезную классификацию объектов как объектов ссылки и объектов значения. Объекты ссылки (reference objects) - это такие объекты, как клиент или счет. Каждый объект обозначает один объект реального мира, и равными считаются объекты, которые совпадают. Объекты значений (value objects) - это, например, дата или денежная сумма. Они полностью определены своими значениями данных. Против существования дубликатов нет возражений, и в системе могут обращаться сотни объектов со значением «1/1/2000». Равенство объектов должно проверяться, для чего перегружается метод equals (а также метод hashCode).

Выбор между ссылкой и значением не всегда очевиден. Иногда вначале есть простое значение с небольшим объемом неизменяемых данных. Затем возникает необходимость добавить изменяемые данные и обеспечить передачу этих изменений при всех обращениях к объекту. Тогда необходимо превратить его в объект ссылки.

Техника

Выполните «Замену конструктора фабричным методом» (Replace Constructor with Factory Method).

Выполните компиляцию и тестирование.

Определите объект, отвечающий за предоставление доступа к объектам.

Им может быть статический словарь или регистрирующий объект.

Можно использовать несколько объектов в качестве точки доступа для нового объекта.

Определите, будут ли объекты создаваться заранее или динамически по мере надобности.

Если объекты создаются предварительно и извлекаются из памяти, необходимо обеспечить их загрузку перед использованием.

Измените фабричный метод так, чтобы он возвращал объект ссылки.

Если объекты создаются заранее, необходимо решить, как обраба тывать ошибки при запросе несуществующего объекта.

Может понадобиться применить к фабрике «Переименование метода» (Rename Method) для информации о том, что она возвращает только существующие объекты.

Выполните компиляцию и тестирование.

Пример

Продолжу с того места, где я закончил пример для «Замены значения данных объектом» (Replace Data Value with Object). Имеется следующий класс клиента:

```
class Customer {
 public Customer (String name) {
    _name = name;
 }
 public String getName() {
    return _name;
 }
 private final String _name;
}
```

Его использует класс заказа:

```
class Order {
 public Order (String customerName) {
    _customer = new Customer(customerName);
 }
 public String getCustomerName() {
    return _customer.getName();
 }
 public void setCustomer(String customerName) {
    _customer = new Customer(customerName);
 }
 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 данный момент это значение. У каждого заказа собственный объект customer, даже если концептуально это один клиент. Я хочу внести такие изменения, чтобы при наличии нескольких заказов для одного и того же клиента в них использовался один и тот же экземпляр класса Customer. В данном случае это означает, что для каждого имени клиента должен существовать только один объект клиента.

Начну с «Замены конструктора фабричным методом» (Replace Constructor with Factory Method). Это позволит мне контролировать процесс создания, что окажется важным в дальнейшем. 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 делаю конструктор закрытым:

```
class Customer {
 private Customer (String name) {
    _name = name;
 }
}
```

Теперь надо решить, как выполнять доступ к клиентам. Я предпочитаю использовать другой объект. Это удобно при работе с чем-нибудь вроде пунктов в заказе. Заказ отвечает за предоставление доступа к пунктам. Однако в данной ситуации такого объекта не видно. В таких случаях я обычно создаю регистрирующий объект, который должен служить точкой доступа, но в данном примере я для простоты организую хранение с помощью статического поля в классе Customer, который превращается в точку доступа:

```
private static Dictionary _instances = new Hashtable();
```

Затем я решаю, как создавать клиентов - динамически по запросу или заранее. Воспользуюсь последним способом. При запуске своего приложения я загружаю тех клиентов, которые находятся в работе. Их можно взять из базы данных или из файла. Для простоты используется явный код. Впоследствии всегда можно изменить это с помощью «Замещения алгоритма» (SubstituteAlgorithm).

```
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, чтобы он возвращал заранее созданного клиента:

```
public static Customer create (String name) {
 return (Customer) _instances.get(name);
```

```
    }
```

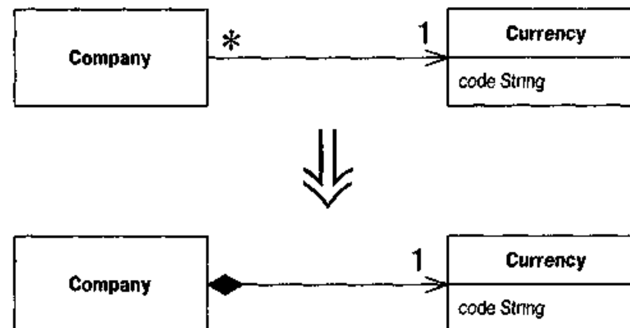
Поскольку метод create всегда возвращает уже существующего клиента, это должно быть пояснено с помощью «Переименования метода» (Rename Method).

```
class Customer {
 public static Customer getNamed (String name) {
    return (Customer) _instances.get(name);
 }
}
```

Замена ссылки значением (Change Reference to Value)

Имеется маленький, неизменяемый и неудобный в управлении объект ссылки.

Превратите его в объект значения.



Мотивировка

Как и для «Замены значения ссылкой» (Change Value to Reference), выбор между объектом ссылки и объектом значения не всегда очевиден. Такого рода решение часто приходится изменять.

Побудить к переходу от ссылки к значению могут возникшие при работе с объектом ссылки неудобства. Объектами ссылки необходимо некоторым образом управлять. Всегда приходится запрашивать у контроллера нужный объект. Ссылки в памяти тоже могут оказаться неудобными в работе. Объекты значений особенно полезны 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 одинаковый результат. Если это так, то не возникает проблем при наличии многих объектов, представляющих одну и ту же вещь. Если значение изменяемое, необходимо обеспечить, чтобы при изменении любого из объектов обновлялись все остальные объекты, которые представляют ту же самую сущность. Это настолько обременительно, что проще всего для этого создать объект ссылки.

Важно внести ясность в смысл слова неизменяемый (immutable). Если есть класс, представляющий денежную сумму и содержащий тип валюты и величину, 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 денежной суммы другим объектом денежной суммы, а не изменять значение в существующем объекте. Связь может измениться, но сам объект денежной суммы не изменяется.

Техника

Проверьте, чтобы преобразуемый объект был неизменяемым или мог стать неизменяемым.

Если объект в настоящее время является изменяемым, добейтесь с помощью «Удаления метода установки» (Remove Setting Method), чтобы он стал таковым.

Если преобразуемый объект нельзя сделать неизменяемым, данный рефакторинг следует прервать.

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;
 }
}
```

Все действия этого класса состоят в хранении и возврате кода валюты. Это объект ссылки, поэтому для получения экземпляра необходимо использовать

```
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")) // возвращает false
```

Чтобы преобразовать это в объект значения, важно удостовериться в неизменяемости объекта. Если объект изменяем, не пытайтесь выполнить это преобразование, поскольку изменяемое значение влечет бесконечные изматывающие ссылки.

В данном 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));
}
```

Если я определяю equals, то должен также определить hashCode. Простой способ сделать это - взять хеш-коды всех полей, используемых в методе equals, и выполнить над ними поразрядное «исключающее или» (^). Здесь это просто, потому что поле только одно:

```
public int hashCode() {
 return _code.hashCode();
}
```

Заменив оба метода, можно выполнить компиляцию и тестирование. Я должен выполнить то и другое, иначе любая коллекция, зависящая от хеширования, такая как Hashtable, HashSet или HashMap, может повести себя неверно.

Теперь я могу создать сколько угодно одинаковых объектов. Я могу избавиться от любого поведения контроллера в классе и фабричном методе и просто использовать конструктор, который теперь можно объявить открытым.

```
new Currency("USD").equals(new Currency("USD")); // теперь возвращает true
```

Замена массива объектом (Replace Array with Object)

Есть массив, некоторые элементы которого могут означать 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 Pefformance();
row.setName("Liverpool");
row.setWins("15");
```

Мотивировка

Массивы представляют собой структуру, часто используемую для организации данных. Однако их следует применять лишь для хранения коллекций аналогичных объектов в определенном порядке. Впрочем, иногда можно столкнуться с хранением в массивах ряда различных объектов. Соглашения типа «первый элемент массива содержит имя лица» трудно запоминаются. Используя объекты, можно передавать такую информацию с помощью названий полей или методов, чтобы не запоминать ее и не полагаться на точность комментариев. Можно также инкапсулировать такую информацию и добавить к ней поведение с помощью «Перемещения метода» (Move Method).

Техника

Создайте новый класс для представления данных, содержащихся в массиве. Организуйте в нем открытое поле для хранения массива. Модифицируйте пользователей массива так, чтобы они применяли новый класс.

Выполните компиляцию и тестирование.

Поочередно для каждого элемента массива добавьте методы получения и установки значений. Дайте методам доступа названия, соответствующие назначению элемента массива. Модифицируйте клиентов так, чтобы они использовали 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 в классе поле и измените методы доступа так, чтобы они использовали это поле.

После изменения каждого элемента выполняйте компиляцию и тестирование.

После замены всех элементов массива полями удалите массив.

Пример

Начну с массива, хранящего название спортивной команды, количество выигрышей и количество поражений. Он будет определен так:

```
String[] row = new Stnng[3];
```

Предполагается его использование с кодом типа:

```
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-данное. (Знаю, что это дурно и плохо, но в свое время все будет исправлено.)

```
public String[] _data = new String[3];
```

Теперь нахожу места, в которых создается массив и выполняется доступ к нему. Вместо создания массива пишем:

```
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]);
```

Поочередно добавляю методы получения и установки с более содержательными именами. Начинаю с названия:

```
class Performance {
 public String getName() {
    return _data[0];
 }
 public void setName(String arg) {
    _data[0] = arg;
 }
}
```

Вхождения этой строки редактирую так, чтобы в них применялись методы доступа:

```
row.setName("Liverpool");
row._data [1] = "15";
String name = row.getName();
int wins = Integer.parseInt(row._data[1]);
```

То же самое можно проделать со вторым элементом. Для облегчения жизни можно инкапсулировать преобразование типа данных:

```
class Performance {
 public int getWins() {
    return Integer.parseInt(_data[1]);
 }
 public void setWins(String arg) {
    _data[1] = arg;
 }
}
client code
row.setName("Liverpool");
row.setWins("15");
String name = row.getName();
int wins = row.getWins();
```

Проделав это для всех элементов, можно объявить массив закрытым:

```
private String[] _data = new String[3];
```

Теперь главная часть этого рефакторинга, замена интерфейса, завершена. Однако полезно будет также заменить массив внутри класса. Это можно сделать, добавив поля для всех элементов массива и переориентировав методы доступа на их использование:

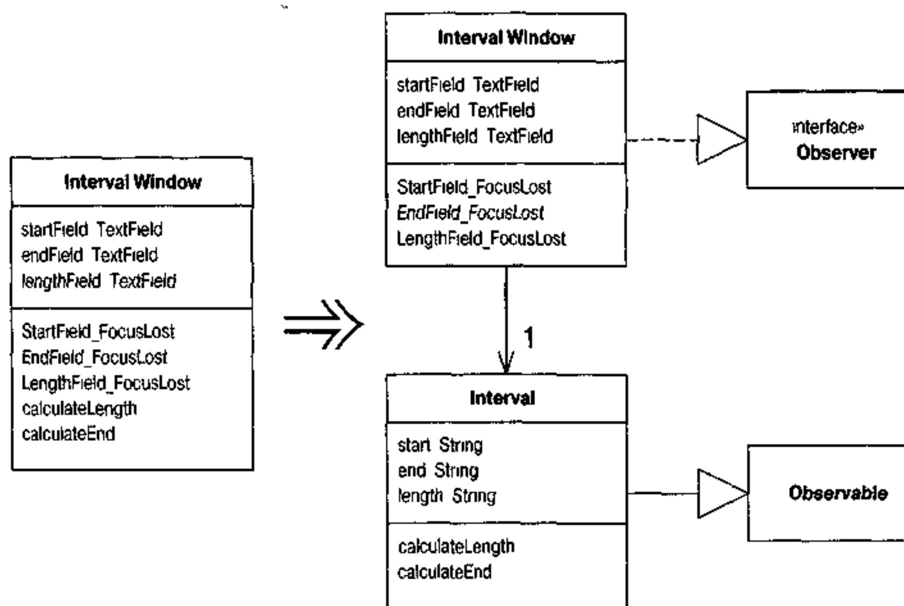
```
class Performance {
 public String getName() {
    return _name;
 }
 public void setName(String arg) {
    _name = arg;
    }
 private String _name;
}
```

Выполнив это для всех элементов массива, можно удалить массив.

Дублирование видимых данных (Duplicate Observed Data)

Есть данные предметной области приложения, присутствующие только в графическом элементе GUI, к которым нужен доступ методам предметной области приложения.

Скопируйте данные в объект предметной области приложения. Создайте объект-наблюдатель, который будет обеспечивать синхронность данных.



Мотивировка

В хорошо организованной многоуровневой системе код, обрабатывающий интерфейс пользователя, отделен от кода, обрабатывающего бизнес-логику. Это делается по нескольким причинам. Может потребоваться несколько различных интерфейсов пользователя для одинаковой бизнес-логики; интерфейс пользователя становится слишком сложным, если выполняет обе функции; сопровождать и развивать объекты предметной области проще, если они отделены от GUI; с разными частями приложения могут иметь дело разные разработчики.

Поведение можно легко разделить на части, чего часто нельзя сделать с данными. Данные должны встраиваться в графический элемент GUI и иметь тот же смысл, что и данные в модели предметной области. Шаблоны создания пользовательского интерфейса, начиная с MVC (модель-представление-контроллер), используют многозвенную систему, которая обеспечивает механизмы, предоставляющие эти данные и поддерживающие их синхронность.

Столкнувшись с кодом, разработанным на основе двухзвенного подхода, в котором бизнес-логика встроена в интерфейс пользователя, необходимо разделить поведение на части. В значительной мере это осуществляется с помощью декомпозиции и перемещения методов. Однако данные нельзя просто переместить, их надо скопировать и обеспечить механизм синхронизации.

Техника

Сделайте класс представления наблюдателем для класса предметной области [Gang of Four].

Если класса предметной области не существует, создайте его.

Если отсутствует ссылка из класса представления в класс предметной области, поместите класс предметной области в поле класса представления.

Примените «Самоинкапсуляцию поля» (Self Encapsulate Field) к данным предметной области в классе GUI.

Выполните компиляцию и тестирование.

Поместите в обработчик события вызов метода установки, чтобы обновлять компонент его текущим значением с помощью прямого доступа.

Поместите метод в обработчик события, который обновляет значение компонента исходя из его текущего значения. Конечно, в этом совершенно нет необходимости; вы просто устанавливаете значение равным его текущему значению, но благодаря использованию метода установки вы даете возможность выполниться любому имеющемуся в нем поведению.

При проведении этой модификации не пользуйтесь методом получения значения компонента, предпочтите непосредственный до ступ к нему. Позже метод получения станет извлекать значение из предметной области, которое изменяется только при выполне нии метода установки.

С помощью тестового кода убедитесь, что механизм обработки событий срабатывает.

Выполните компиляцию и тестирование.

Определите данные и методы доступа в классе предметной области.

Убедитесь, что метод установки в предметной области запускает механизм уведомления в схеме наблюдателя.

Используйте в предметной области тот же тип данных, что и в представлении: обычно это строка. Преобразуйте тип данных при проведении следующего рефакторинга.

Переадресуйте методы доступа на запись в поле предметной области.

Измените метод обновления в наблюдателе так, чтобы он копировал данные из поля объекта предметной области в элемент GUI.

Выполните компиляцию и тестирование.

Пример

Начну с окна, представленного на рисунке 8.1. Поведение очень простое. Как только изменяется значение в одном из текстовых полей, обновляются остальные поля. При изменении полей Start или End вычисляется length; при изменении поля length вычисляется End. Все методы находятся в простом классе IntervalWindow. Поля реагируют на потерю фокуса.

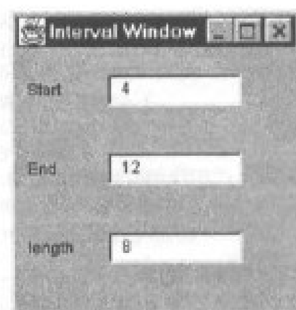


Рисунок 8.1 Простое, окно GUI

```
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);
    }
 }
}
```

Слушатель реагирует вызовом StartField_FocusLost при потере фокуса начальным полем и вызовом EndField_FocusLost или LengthField_FocusLost при потере фокуса другими полями. Эти методы обработки событий выглядят так:

```
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 сделал такое окно, скажу, что это простейший способ, предложенный моей средой разработки (Cafe).

При появлении нецифрового символа в любом поле вводится ноль и вызывается соответствующая вычислительная процедура:

```
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 ("Unexpected Number Format Error");
 }
}
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 ("Unexpected Number Format Error");
 }
}
```

Моей задачей, если я решу этим заниматься, станет отделение внутренней логики от GUI. По существу, это означает перемещение calculateLength и calculateEnd в отдельный класс предметной области. Для этого мне необходимо обращаться к данным начала, конца и длины, минуя оконный класс. Это можно сделать, только дублируя эти данные в классе предметной области и синхронизируя их с GUI. Эта задача описывается в «Дублировании видимых данных» (Duplicate Observed Data).

В данный момент класса предметной области нет, поэтому надо его создать (пустой):

```
class Interval extends Observable {}
```

В окне должна быть ссылка на этот новый класс предметной области.

```
private Interval _subject;
```

Затем надо соответствующим образом инициализировать это поле и сделать окно интервала наблюдателем интервала. Это можно сделать, поместив в конструктор окна интервала 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 GUI инициализируется из класса предметной области. Для этого я должен объявить, что окно интервала реализует Observable:

```
public class IntervalWindow extends Frame implements Observer
```

Чтобы реализовать наблюдателя, я должен создать метод обновления update. Вначале он может иметь тривиальную реализацию:

```
public void update(Observable observed, Object arg) { }
```

В этот момент я могу выполнить компиляцию и тестирование. Никакой реальной модификации еще не проведено, но ошибки можно совершить в простейших местах.

Теперь можно заняться перемещением полей. Как обычно, я делаю изменения в полях поочередно. Демонстрируя владение английским языком, начну с конечного поля. Сначала следует применить самоинкапсуляцию поля. Текстовые поля обновляются с помощью методов getText и setText. Создаю методы доступа, которые их вызывают:

```
String getEnd() {
 return _endField.getText();
}
void setEnd (String arg) {
 _endField.setText(arg);
}
```

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 (“Unexpected Number Format Error”)
 }
}
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 (“Unexpected Number Format Error”);
```

```
    }
  }
  void EndField_FocusLost(java.awt.event.FocusEvent event) {
   if (isNotInteger(getEnd()))
      setEnd("0");
   calculateLength();
  }
```

Это обычный процесс для «Самоинкапсуляции поля» (Self Encapsulate Field). Однако при работе с GUI возникает сложность. Пользователь может изменить значение поля непосредственно, не вызывая setEnd. Поэтому я должен поместить вызов setEnd в обработчик события для GUI. Этот вызов заменяет старое значение поля End текущим. Конечно, в данный момент это бесполезно, но тем самым обеспечивается прохождение данных, вводимых пользователем, 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 пользуюсь getEnd, а обращаюсь к полю непосредственно. Я поступаю так потому, что после продолжения рефакторинга getEnd будет получать значение от объекта предметной области, а не из поля. Тогда вызов метода привел бы к тому, что при каждом изменении пользователем значения поля этот код возвращал бы его обратно, поэтому мне необходим здесь прямой доступ. Теперь я могу выполнить компиляцию и протестировать инкапсулированное поведение.

Затем добавляю в класс предметной области поле конца:

```
  class Interval.{
   private String _end = “0”;
  }
```

Я инициализирую его тем же значением, которым оно инициализируется в GUI. Теперь я добавлю методы получения и установки:

```
  class Interval {
   String getEnd() {
      return _end;
    }
   void setEnd (String arg) {
      _end = arg;
    }
   setChanged();
   notifyObservers();
  }
```

Поскольку используется схема с наблюдателем, следует поместить в метод установки код, посылающий уведомление. Я использую в качестве последнего строку, а не число (что было бы более логично). Но я хочу, чтобы модификаций было как можно меньше. Когда данные будут успешно продублированы, я смогу поменять внутренний тип данных на целочисленный.

Теперь можно перед дублированием еще раз выполнить компиляцию и тестирование. Благодаря проведенной подготовительной работе рискованность этого ненадежного шага снизилась.

Сначала модифицируем методы доступа в IntervalWindow, чтобы использовать в них Interval.

```
  class IntervalWindow {
   String getEnd() {
```

```
      return _subject.getEnd();
   }
   void setEnd (String arg) {
      _subject.setEnd(arg);
   }
  }
```

Необходимо также обновить update, чтобы обеспечить реакцию GUI на уведомление:

```
  class IntervalWindow {
   public void update(Observable observed, Object arg) {
      _endField.setText(_subject.getEnd());
   }
  }
```

Это еще одно место, где необходим прямой доступ. А вызов метода установки привел бы к бесконечной рекурсии.

Теперь я могу выполнить компиляцию и тестирование, и данные правильно дублируются.

Можно повторить эти действия для оставшихся двух полей, после чего с помощью «Перемещения метода» (Move Method) перенести calculateEnd и calculateLength в класс интервала. Я получу класс предметной области, содержащий все поведение и данные предметной области отдельно от кода GUI.

Сделав это, можно попробовать вообще избавиться от класса GUI. Если это старый класс AWT (Abstract Windows Toolkit), то можно улучшить внешний вид с помощью Swing, причем Swing лучше справляется с координированием действий. Построить Swing GUI можно поверх класса предметной области. Успешно сделав это, можно удалить старый класс GUI.

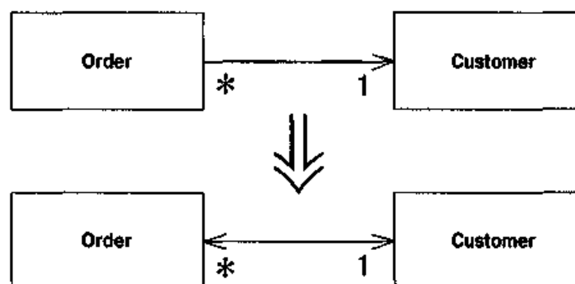
Применение слушателей событий

«Дублирование наблюдаемых данных» (Duplicate Observed Data) применимо и в том случае, когда вы используете слушателей событий вместо наблюдателя/наблюдаемого. В этом случае надо создать слушателя и событие в модели предметной области (либо использовать классы AWT, если это вас не смущает). Объект предметной области должен зарегистрировать слушателей таким же образом, как наблюдаемый класс, и посылать им событие при своем изменении, как в методе обновления. После этого окно интервала может реализовать интерфейс пользователя с помощью внутреннего класса и вызывать необходимые методы обновления.

Замена однонаправленной связи двунаправленной (Change Unidirectional Association to Bidirectional)

Есть два класса, каждый из которых должен использовать функции другого, но ссылка между ними есть только в одном направлении.

Добавьте обратные указатели и измените модификаторы, чтобы они обновляли оба набора.



Мотивировка

Может случиться так, что первоначально организовано два класса, один из которых ссылается на другой. Со временем обнаруживается, что клиенту класса, на который происходит ссылка, нужны объекты, которые на него ссылаются. Фактически это означает перемещение по указателям в обратном направлении. Однако это невозможно, потому что указатели представляют собой односторонние ссылки. Часто удается справиться с этой проблемой, найдя другой маршрут. Это может вызвать некоторое увеличение объема вычислений, но оправданно, и в классе, на который происходит ссылка, можно создать метод, использующий это поведение.

Однако иногда такой способ затруднителен и требуется организовать двустороннюю ссылку, которую часто называют обратным указателем (back pointer). Без навыков работы с обратными указателями в них легко запутаться. Однако если привыкнуть к этой идиоме, она перестает казаться сложной.

Идиома настолько неудобна, что следует проводить для нее тестирование, по крайней мере, в период освоения. Обычно я не утруждаю себя тестированием методов доступа (риск не очень велик), поэтому данный вид рефакторинга - один из немногих, для которых добавляется тест.

В данном рефакторинге обратные указатели применяются для реализации двунаправленности. Другие приемы, например объекты связи (link objects), требуют других методов рефакторинга.

Техника

Добавьте поле для обратного указателя.

Определите, который из классов будет управлять связью.

Создайте вспомогательный метод на подчиненном конце связи. Дайте ему имя, ясно указывающее на ограниченность применения.

Если имеющийся модификатор находится на управляющем конце, модифицируйте его так, чтобы он обновлял обратные указатели.

Если имеющийся модификатор находится на управляемом конце, создайте управляющий метод на управляющем конце и вызывайте его из имеющегося 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 начну рефакторинг с добавления поля в класс Customer. Поскольку у клиента может быть несколько заказов, это поле будет коллекцией. Я не хочу, чтобы один и тот же заказ клиента появлялся в коллекции дважды, поэтому типом коллекции выбирается множество:

```
class Customer {
 private Set _orders = new HashSet();
}
```

Теперь надо решить, который из классов будет отвечать за связь. Я предпочитаю возлагать ответственность на один класс, т. к. это позволяет хранить всю логику управления связью в одном месте. Процедура принятия решения выглядит так:

1 Если оба объекта представляют собой объекты ссылок, и связь имеет тип «один-ко-многим», то управляющим будет объект, содержащий одну ссылку. (То есть если у одного клиента несколько заказов, связью управляет заказ.)

2 Если один объект является компонентом другого (т. е. связь имеет тип «целое-часть»), управлять связью должен составной объект.

3 Если оба объекта представляют собой объекты ссылок, и связь имеет тип «многие-ко-многим», то в качестве управляющего можно произвольно выбрать класс заказа или класс клиента.

Поскольку отвечать за связь будет заказ, я должен добавить к клиенту вспомогательный метод, предоставляющий прямой доступ к коллекции заказов. С его помощью модификатор заказа будет синхронизировать два набора указателей. Я дам этому методу имя friendOrders, чтобы предупредить о необходимости применять его только в данном особом случае. Я также ограничу его область видимости

пакетом, если это возможно. Мне придется объявить его с модификатором видимости public, если второй класс находится в другом пакете:

```
class Customer.{
 Set friendOrders() {
    /* должен использоваться только в Order при модификации связи */
    return _orders;
 }
}
```

Теперь изменяю модификатор так, чтобы он обновлял обратные указатели:

```
class Order.{
 void setCustomer (Customer arg) {
    if (_customer != null) _customer.frendOrders().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 проверки его на null, но надо проверять на null аргумент. Однако базовая схема всегда одинакова: сначала новому объекту сообщается, чтобы он уничтожил указатель на исходный объект, затем мы устанавливаем указатель исходного объекта на новый и сообщаем этому последнему, чтобы он добавил ссылку на исходный.

Если вы хотите модифицировать ссылку через Customer, пусть он вызывает управляющий метод:

```
class Customer.{
 void addOrder(Order arg) {
    arg.setCustomer(this);
 }
}
```

Если у заказа может быть несколько клиентов, то это случай отношения «многие-ко-многим», и методы выглядят так:

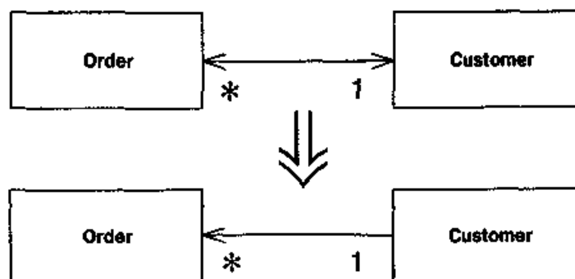
```
class Order {
 //управляющие методы
 void addCustomer (Customer arg) {
    arg.friendOrders().add(this);
    _customers.add(arg);
 }
 void removeCustomer (Customer arg) {
    arg.friendOrders().remove(this);
    _customers.remove(arg);
 }
}
class Customer.{
 void addOrder(Order arg) {
    arg.addCustomer(this);
 }
 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.

Опустить ненужный конец связи.



Мотивировка

Двунаправленные связи удобны, но не даются бесплатно. А ценой является дополнительная сложность поддержки двусторонних ссылок и обеспечения корректности создания и удаления объектов. Для многих программистов двунаправленные ссылки непривычны и потому часто служат источником ошибок.

Большое число двусторонних ссылок может служить источником ошибок, приводящих к появлению «зомби» - объектов, которые должны быть уничтожены, но все еще существуют, потому что не была удалена ссылка на них.

Двунаправленные ассоциации устанавливают взаимозависимость между двумя классами. Какое-либо изменение в одном классе может повлечь изменение в другом. Если классы находятся в разных пакетах, возникает взаимозависимость между пакетами. Наличие большого числа взаимозависимостей приводит к возникновению сильно связанных систем, в которых любые малозначительные изменения вызывают массу непредсказуемых последствий.

Двунаправленными связями следует пользоваться только тогда, когда это действительно необходимо. Как только вы замечаете, что двунаправленная связь больше не несет никакой нагрузки, лишний конец следует обрубить.

Техника

Исследуйте все места, откуда осуществляется чтение поля с указателем, которое вы хотите удалить, и убедитесь в осуществимости удаления.

Рассмотрите непосредственное чтение и дальнейшие методы, которые вызывают методы.

Выясните, можно ли определить другой объект без помощи указателя. Если да, то, применив «Замещение алгоритма» (Substitute Algorithm) к методу получения, можно разрешить клиентам пользоваться им даже при отсутствии указателя.

Рассмотрите возможность передачи объекта в качестве аргумента всех методов, использующих это поле.

Если клиентам необходим метод получения данных объекта, выполните «Самоинкапсуляцию поля» (Self Encapsulate Field); «Замещение алгоритма» (Substitute Algorithm) для метода получения, компиляцию и тестирование.

Если метод получения данных объекта клиентам не нужен, модифицируйте каждого клиента так, чтобы он получал объект в поле другим способом. Выполняйте компиляцию и тестирование после каждой модификации.

Когда кода, читающего поле, не останется, уберите все обновления поля и само поле.

Если присваивание полю происходит во многих местах, выполните «Самоинкапсуляцию поля» (Self Encapsulate Field), чтобы всюду применялся один метод установки. Выполните компиляцию и тестирование. Модифицируйте метод установки так, чтобы у него было пустое тело. Выполните компиляцию и тестирование. Если они пройдут удачно, удалите поле, метод его установки и все вызовы этого метода.

Выполните компиляцию и тестирование.

Пример

Начну с того места, где я завершил пример в «Замене однонаправленной связи двунаправленной» (Change Unidirectional Association to Bidirectional). 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 обнаружил, что в моем приложении заказы появляются, только если клиент уже есть, поэтому я решил разорвать ссылку от заказа к клиенту.

Самое трудное в данном рефакторинге - проверка возможности его осуществления. Необходимо убедиться в безопасности, а выполнить рефакторинг легко. Проблема в том, зависит ли код от наличия поля клиента. Чтобы удалить поле, надо предоставить другую альтернативу.

Первым делом изучаются все операции чтения поля и все методы, использующие эти операции. Есть ли другой способ предоставить объект клиента? Часто это означает, что надо передать клиента в качестве аргумента операции. Вот упрощенный пример:

```
class Order {
 double getDiscountedPrice() {
    return getGrossPrice() * (1 - _customer.getDiscount());
 }
}
```

заменяется на

```
class Order {
 double getDiscountedPrice(Customer customer) {
    return getGrossPrice() * (1 – customer.getDiscount());
 }
}
```

Особенно хорошо это действует, когда поведение вызывается клиентом, которому легко передать себя в качестве аргумента. Таким образом,

```
class Customer double getPriceFor(Order order) {
 Assert isTrue(_orders.contains(order));
 // см «Введение утверждения» (Introduce Assertion)
 return order getDiscountedPrice();
}
```

становится

```
class Customer {
 double getPriceFor(Order order) {
    Assert isTrue(_orders.contains(order));
    return order getDiscountedPrice(this);
 }
}
```

Другая рассматриваемая альтернатива - такое изменение метода получения данных, которое позволяет ему получать клиента, не используя поле. Тогда можно применить к телу Order getCustomer «Замещение алгоритма» (Substitute Algorithm) и сделать что-то вроде 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 выполнение может даже быть не таким медленным, если применить запрос базы данных. Если в классе заказа есть методы, работающие с полем клиента, то с помощью «Самоинкапсуляции поля» (Self Encapsulate Field) можно изменить их, чтобы они использовали getCustomer.

Если метод доступа сохранен, то связь остается двунаправленной по интерфейсу, но по реализации будет однонаправленной. Я удаляю обратный указатель, но сохраняю взаимозависимость между двумя классами.

При замене метода получения все остальное я оставляю на более позднее время. Или же поочередно изменяю вызывающие методы так, чтобы они получали клиента из другого источника. После каждого изменения я выполняю компиляцию и тестирование. На практике этот процесс обычно осуществляется довольно быстро. Будь это сложно, я отказался бы от этого рефакторинга.

Избавившись от чтения поля, можно заняться записью в него. Для этого надо лишь убрать все присваивания полю, а затем удалить поле. Поскольку никто его больше не читает, это не должно иметь значения.

Замена магического числа символической константой (Replace Magic Number with Symbolic Constant)

Есть числовой литерал, имеющий определенный смысл.

Создайте константу, дайте ей имя, соответствующее смыслу, 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;
```

Мотивировка

Магические числа - одна из старейших болезней в вычислительной науке. Это числа с особыми значениями, которые обычно не очевидны. Магические числа очень неприятны, когда нужно сослаться на логически одно и то же число в нескольких местах. Если числа меняются, их модификация становится кошмаром. Даже если не требуется модификация, выяснить, что происходит, можно лишь с большим трудом.

Многие языки позволяют объявлять константы. При этом не наносится ущерб производительности и значительно улучшается читаемость кода.

Прежде чем выполнять этот рефакторинг, всегда стоит поискать возможную альтернативу. Посмотрите, как используется магическое число. Часто можно найти лучший способ его применения. Если магиче-кое число представляет собой код типа, попробуйте выполнить «Замену кода типа классом» (Replace Type Code with Class). Если магическое число представляет длину массива, используйте вместо него при обходе массива anArray.length.

Техника

Объявите константу и проинициализируйте ее значением магическое число.

Найдите все вхождения магического числа.

Проверьте, согласуется ли магическое число с использованием константы; если да, замените магическое число константой.

Выполните компиляцию.

После замены всех магических чисел выполните компиляцию и тестирование. Все должно работать так, как если бы ничего не изменялось.

Хорошим тестом будет проверка легкости изменения константы. Для этого может потребоваться изменить некоторые ожидаемые результаты в соответствии с новым значением. Не всегда это оказывается возможным, но является хорошим приемом (когда срабатывает).

Инкапсуляция поля (Encapsulate Field)

Имеется открытое поле.

Сделайте его закрытым и обеспечьте методы доступа.

```
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 _lowl}
int getHigh{return _high;}
```

Мотивировка

Одним из главных принципов объектно-ориентированного программирования является инкапсуляция, или сокрытие данных. Она гласит, что данные никогда не должны быть открытыми. Если сделать данные открытыми, объекты смогут читать и изменять их значения без ведома объекта-владельца этих данных. Тем самым данные отделяются от поведения.

Это считается неправильным, потому что уменьшает модульность программы. Когда данные и использующее их поведение сгруппированы вместе, проще изменять код, поскольку изменяемый код расположен в одном месте, а не разбросан по всей программе.

Процедура «Инкапсуляции поля» (Encapsulate Field) начинается с сокрытия данных и создания методов доступа. Однако это лишь первый шаг. Класс, в котором есть только методы доступа, это немой класс, не реализующий действительные возможности объектов, а терять напрасно объект очень жалко. Выполнив «Инкапсуляцию поля» (Encapsulate Field), я отыскиваю методы, которые используют новые методы, мечтая упаковать свои вещи и поскорее переехать в новый объект с помощью «Перемещения метода» (Move Method).

Техника

Создайте методы получения и установки значений поля.

Найдите за пределами класса всех клиентов, ссылающихся на поле. Если клиент пользуется значением, замените его вызовом метода получения. Если клиент изменяет значение, замените его вызовом метода установки.

Если поле представляет собой объект и клиент вызывает метод, модифицирующий этот объект, это использование, а не изменение. Применяйте метод установки только для замены присваивания.

После каждого изменения выполните компиляцию и тестирование.

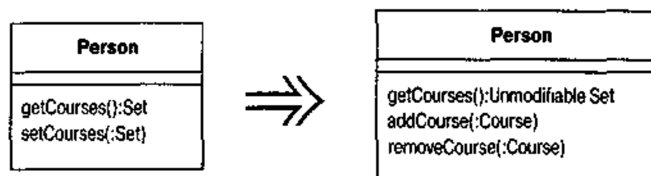
После модификации всех клиентов объявите поле закрытым.

Выполните компиляцию и тестирование.

Инкапсуляция коллекции (Encapsulate Collection)

Метод возвращает коллекцию.

Сделайте возвращаемое методом значение доступным только для чтения и создайте методы добавления/удаления элементов.



Мотивировка

Часто в классе содержится коллекция экземпляров. Эта коллекция может быть массивом, списком, множеством или вектором. В таких случаях часто есть обычные методы получения и установки значений для коллекции.

Однако коллекции должны использовать протокол, несколько отличный от того, который используется другими типами данных. Метод получения не должен возвращать сам объект коллекции, потому что это позволило бы клиентам изменять содержимое коллекции без ведома владеющего ею класса. Кроме того, это чрезмерно раскрывало бы клиентам строение внутренних структур данных объекта. Метод получения для многозначного атрибута должен возвращать такое значение, которое не позволяло бы изменять коллекцию и не раскрывало бы лишних данных о ее структуре. Способ реализации зависит от используемой версии Java.

Кроме того, не должно быть метода, присваивающего коллекции значение; вместо этого должны быть операции для добавления и удаления элементов. Благодаря этому объект-владелец получает контроль над добавлением и удалением элементов коллекции.

Такой протокол осуществляет правильную инкапсуляцию коллекции, что уменьшает связывание владеющего ею класса с клиентами.

Техника

Создайте методы для добавления и удаления элементов коллекции.

Присвойте полю пустую коллекцию в качестве начального значения.

Выполните компиляцию.

Найдите вызовы метода установки. Измените метод установки так, чтобы он использовал операции добавления и удаления элементов, или сделайте так, чтобы эти операции вызывали клиенты.

Методы установки используются в двух случаях: когда коллекция пуста и когда метод заменяет непустую коллекцию.

Может оказаться желательным «Переименование метода» (Rename Method). Измените имя с set на initialize или replace.

Выполните компиляцию и тестирование.

Найдите всех пользователей метода получения, модифицирующих коллекцию. Измените их так, чтобы они применяли методы добавления и удаления элементов. После каждого изменения выполняйте компиляцию и тестирование.

Когда все вызовы метода получения, модифицирующие коллекцию, заменены, модифицируйте метод получения, чтобы он возвращал представление коллекции, доступное только для чтения.

В Java 2 им будет соответствующее немодифицируемое представление коллекции.

В Java 1.1 должна возвращаться копия коллекции.

Выполните компиляцию и тестирование.

Найдите пользователей метода получения. Поищите код, которому следует находиться в объекте, содержащем метод. С помощью «Выделения метода» (Extract Method) и «Перемещения метода» (Move Method) перенесите этот код в объект.

Для Java 2 на этом работа завершена. В Java 1.1, однако, клиенты могут использовать перечисление. Чтобы создать перечисление:

Поменяйте имя текущего метода получения и создайте новый метод получения, возвращающий перечисление. Найдите пользователей прежнего метода получения и измените их так, чтобы они использовали один из новых методов.

Если такой скачок слишком велик, примените к прежнему методу получения «Переименование метода» (Rename Method), создайте новый метод, возвращающий перечисление, и измените места вызова, чтобы в них использовался новый метод.

Выполните компиляцию и тестирование.

Примеры

В Java 2 добавлена целая новая группа классов для работы с коллекциями. При этом не только добавлены новые классы, но и изменился стиль использования коллекций. В результате способ инкапсуляции коллекции различен в зависимости от того, используются ли коллекции Java 2 или коллекции Java 1.1. Сначала я опишу подход Java 2, т. к. полагаю, что в течение срока жизни этой книги более функциональные коллекции Java 2 вытеснят коллекции Java 1.1.

Пример: Java 2

Некое лицо слушает курс лекций. Наш курс довольно прост:

```
class Course.{
 public Course (String name, boolean isAdvanced) {...};
 public boolean isAdvanced() {...};
}
```

Я не буду обременять класс Course чем-либо еще. Нас интересует класс Person:

```
class Person.{
 public Set getCourses() {
    return _courses;
 }
 public void setCourses(Set arg) {
    _courses = arg;
 }
 private Set _courses;
}
```

При таком интерфейсе клиенты добавляют курсы с помощью следующего кода:

```
Person kent = new Person();
Set s = new HashSet();
s.add(new Course (“Smalltalk Programming”, false));
s.add(new Course (“Appreciating Single Halts”, 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 так:

```
Iterator iter = person.getCourses().iterator();
int count = 0;
while (iter.hasNext()) {
 Course each = (Course) iter.next();
 if (each.isAdvanced()) count ++;
}
```

Первым делом я хочу создать надлежащие методы модификации для этой коллекции и выполнить компиляцию, вот так:

```
class Person {
 public void addCourse (Course arg) {
    _courses.add(arg);
 }
 public void removeCourse (Course arg) {
    _courses.remove(arg);
 }
}
```

Облегчим также себе жизнь, проинициализировав поле:

```
private Set _courses = new HashSet();
```

Теперь посмотрим на пользователей метода установки. Если клиентов много и метод установки интенсивно используется, необходимо заменить тело метода установки, чтобы в нем использовались операции добавления и удаления. Сложность этой процедуры зависит от способа использования метода установки. Возможны два случая. В простейшем из них клиент инициализирует значения с помощью метода установки, т. е. до применения метода установки курсов не существует. В этом случае я изменяю тело метода установки так, чтобы в нем использовался 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 такой модификации разумно с помощью «Переименования метода» (Rename Method) 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, а затем добавлять новые. Однако это происходит редко (как и бывает с общими случаями).

Если я знаю, что другого поведения при добавлении элементов во время инициализации нет, можно убрать цикл и применить addAll.

```
public void imtializeCourses(Set arg) {
 Assert.isTrue(_courses.isEmpty());
    _courses.addAll(arg);
 }
}
```

Я не могу просто присвоить значение множеству, даже если предыдущее множество было пустым. Если клиент соберется модифицировать множество после того, как передаст его, это станет нарушением инкапсуляции. Я должен создать копию.

Если клиенты просто создают множество и пользуются методом установки, я могу заставить их пользоваться методами добавления и удаления непосредственно и полностью убрать метод установки. Например, следующий 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

```
Person kent = new Person();
kent.addCourse(new Course ("Smalltalk Programming", false));
kent.addCourse(new Course ("Appreciating Single Malts", true));
```

Теперь я смотрю, кто использует метод получения. В первую очередь меня интересуют случаи модификации коллекции с его помощью, например:

```
kent.getCourses().add(new Course ("Brutal Sarcasm", false));
```

Я должен заменить это вызовом нового метода модификации:

```
kent.addCourse(new Course ("Brutal Sarcasm", false));
```

Выполнив всюду такую замену, я могу проверить, не выполняется ли где-нибудь модификация через метод получения, изменив его тело так, чтобы оно возвращало немодифицируемое представление:

```
public Set getCourses() {
 return Collections.unmodifiableSet(_courses);
}
```

Теперь я инкапсулировал коллекцию. Никто не сможет изменить элементы коллекции, кроме как через методы Person.

Перемещение поведения в класс

Правильный интерфейс создан. Теперь я хочу посмотреть на пользователей метода получения и найти код, который должен располагаться в 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 поместить в Person, потому что он использует данные только класса Person. Сначала я применяю к коду «Выделение метода» (Extract Method):

```
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;
   }
```

После этого с помощью «Перемещения метода» (Move Method) я переношу его в 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 встречается такой случай:

```
   kent.getCourses().size();
```

Его можно заменить более легко читаемым:

```
   kent.numberOfCourses();
   class Person {
    public int numberOfCourses() {
       return _courses.size();
    }
   }
```

Несколько лет назад меня озаботило бы, что такого рода перемещение поведения в Person приведет к разбуханию этого класса. Практика показывает, что обычно с этим неприятностей не возникает.

Пример: Java 1.1

Во многих отношениях ситуация с Java 1.1 аналогична Java 2. Я воспользуюсь тем же самым примером, но с вектором:

```
   class Person {
    public Vector getCourses() {
       return _courses;
    }
    public void setCourses(Vector arg) {
       _courses = arg;
    }
    private Vector _courses;
   }
```

Я снова начинаю с создания модифицирующих методов 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();
 }
```

Можно модифицировать setCourses, чтобы он инициализировал вектор:

```
 public void imtializeCourses(Vector arg) {
  Assert.isTrue(_courses.isEmpty());
  Enumeration e = arg.elements();
  while (e.hasMoreElements()) {
     addCourse((Course) e.nextElement());
  }
 }
```

Я изменяю пользователей метода доступа, чтобы они применяли методы модификации, поэтому

```
 kent.getCourses().addElement(new Course ("Brutal Sarcasm", false));
```

превращается в

```
 kent.addCourse(new Course ("Brutal Sarcasm", false));
```

Последний шаг будет иным, потому что у векторов нет немодифируемой версии:

```
 class Person Vector getCourses() {
  return (Vector) _courses.clone();
 }
```

Теперь я инкапсулировал коллекцию. Никто не сможет изменить элементы коллекции, кроме как через методы person.

Пример: инкапсуляция массивов

Массивы применяются часто, особенно программистами, которые не знакомы с коллекциями. Я редко работаю с массивами, потому что предпочитаю функционально более богатые коллекции. Часто при инкапсуляции я перехожу от массивов к коллекциям.

На этот раз я начну с массива строк для skills (навыков):

```
 String[] getSkills() {
  return _skills;
 }
 void setSkills (String[] arg) {
  _skills = arg;
 }
 String[] _skills;
```

Я снова начинаю с создания методов модификации. Поскольку клиенту, вероятно, потребуется изменить значение в определенной позиции, мне нужна операция установки конкретного элемента:

```
 void setSkill(int index, String newSkill) {
  _skills[index] = newSkill;
 }
```

Установить значения для всего массива можно с помощью такой операции:

```
 void setSkills (String[] arg) {
```

```
  _skills = new String[arg.length];
  for (int i=0; i < arg.length; i++) setSkill(i, arg[i]);
 }
```

Здесь возможны многочисленные ошибки, если с удаляемыми элементами должны производиться какие-то действия. Ситуация осложняется, когда массив аргумента отличается по размеру от исходного массива. Эта еще один довод в пользу коллекций.

Теперь я могу начать поиск пользователей метода получения. Строку

```
kent getSkills()[1] = "Рефакторинг";
```

можно заменить следующей:

```
kent.setSkill(1, "Рефакторинг");
```

Выполнив все изменения, можно модифицировать метод получения чтобы он возвращал копию:

```
String[] getSkills() {
 String[] result = new String[_skills.length];
 System arraycopy(_skills, 0, result, 0, _skills.length);
 return result;
}
```

Теперь хорошо 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)

Требуется взаимодействие со структурой записи в традиционной программной среде.

Создайте для записи немой объект данных.

Мотивировка

Структуры записей часто встречаются в программных средах. Бывают разные основания для ввода их в объектно-ориентированные программы. Возможно, вы копируете унаследованную программу или связываетесь со структурированной записью с помощью традиционного API, или это запись базы данных. В таких случаях бывает полезно создать класс интерфейса для работы с таким внешним элементом. Проще всего сделать класс похожим на внешнюю запись. Другие поля и методы можно поместить в класс позднее. Менее очевиден, но очень убедителен случай массива, в котором элемент по каждому индексу имеет особый смысл. В этом случае применяется «Замена массива объектом» (Replace Array with Object).

Техника

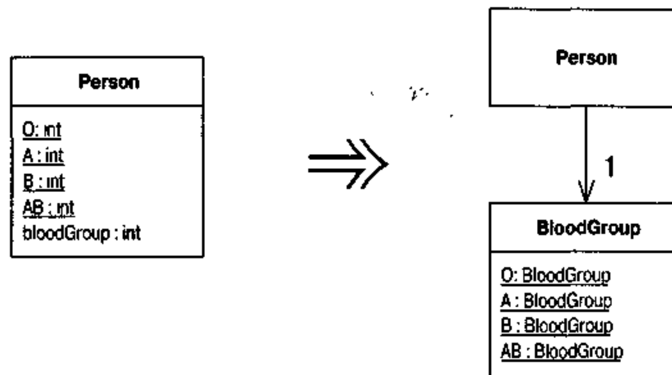
Создайте класс, который будет представлять запись.

Создайте в классе закрытое поле с методом получения и методом У тановки для каждого элемента данных.

Теперь у вас есть немой объект данных. В нем пока нет поведения, эта проблема может быть исследована позднее в ходе рефакторинга.

Замена кода типа классом (Replace Type Code with Class)

У класса есть числовой тип, который не влияет на его поведение. Замените число новым классом.



Мотивировка

Коды числового типа, или перечисления, часто встречаются в языках, основанных на С. Символические имена делают их вполне читаемыми. Проблема в том, что символическое имя является просто псевдонимом; компилятор все равно видит лежащее в основе число. Компилятор проверяет тип с помощью числа, а не символического имени. Все методы, принимающие в качестве аргумента код типа, ожидают число, и ничто не может заставить их использовать символическое имя. Это снижает читаемость кода и может стать источником ошибок.

Если заменить число классом, компилятор сможет проверять тип класса. Предоставив для класса фабричные методы, можно статически проверять, чтобы создавались только допустимые экземпляры и эти экземпляры передавались правильным объектам.

Однако прежде чем применять «Замену кода типа классом» (Replace Type Code with Class), необходимо рассмотреть возможность другой замены кода типа. Заменяйте код типа классом только тогда, когда код типа представляет собой чистые данные, т. е. не изменяет поведение внутри утверждения switch. Для начала, Java может выполнять переключение только по целому числу, но не по произвольному классу, поэтому замена может оказаться неудачной. Еще важнее, что все переключатели должны быть удалены с помощью «Замены условного оператора полиморфизмом» (Replace Conditional with Polymorphism). Для этого рефакторинга код типа сначала надо обработать с помощью «Замены кода типа подклассом» (Replace Type Code with Subclasses) или «Замены кода типа состоянием/стратегией» (Replace Type Code with State/Strategy). Даже если код типа не меняет поведения, зависящего от его значени может обнаружиться поведение, которое лучше поместить в клас кода типа, поэтому будьте готовы оценить применение одного-двух «Перемещений метода» (Move Method).

Техника

Создайте новый класс для кода типа.

В классе должны быть поле кода, соответствующее коду типа, и метод получения его значения. Он должен располагать статическими переменными для допустимых экземпляров класса и статическим методом, возвращающим надлежащий экземпляр исходя из первоначального кода.

Модифицируйте реализацию исходного класса так, чтобы в ней использовался новый класс.

Сохраните интерфейс, основанный на старом коде, но измените статические поля, чтобы для генерации кодов применялся новый класс. Измените другие использующие код методы, чтобы они получали числовые значения кода из нового класса.

Выполните компиляцию и тестирование.

После этого новый класс может проверять коды на этапе исполнения.

Для каждого метода исходного класса, в котором используется код, создайте новый метод, использующий вместо этого новый класс.

Для методов, в которых код участвует в качестве аргумента, следует создать новые методы, использующие в качестве аргумента новый класс. Для методов, которые возвращают код, нужно создать новые методы, возвращающие экземпляр нового класса. Часто разумно выполнить «Переименование метода» (Rename Method) для прежнего метода доступа, прежде чем создавать новый, чтобы сделать программу понятнее, когда в ней используется старый код.

Поочередно измените клиентов исходного класса, чтобы они исполь зовали новый интерфейс.

Выполняйте компиляцию и тестирование после модификации к дого клиента.

Может потребоваться изменить несколько методов, чтобы достичъ согласованности, достаточной для выполнения, компиляци тестирования.

Удалите прежний интерфейс, использующий коды и статичес объявления 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;
 }
}
```

Начну 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 = {0, 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, 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 исполнения. Для того чтобы от внесенных изменений действительно был толк, надо изменить всех входящих в класс Person, тогда они будут использовать класс группы крови вместо целых чисел.

Сначала я применяю «Переименование метода» (Rename Method) к методу доступа к группе крови в классе Person, чтобы сделать ясным новое положение дел:

```
class Person.{
    public int getBloodGroupCode() {
        return _bloodGroup.getCode();
    }
}
```

Затем я ввожу новый метод получения значения, использующий новый класс:

```
public BloodGroup getBloodGroup()O {
    return _bloodGroup;
}
```

Создаю также новый конструктор и метод установки значения, использующие новый класс:

```
public Person (BloodGroup bloodGroup ) {
    _bloodGroup = bloodGroup;
}
public void setBloodGroup(BloodGroup arg) {
    _bloodGroup = arg;
}
```

Теперь начинаю работу с клиентами Person. Хитрость в том, чтобы рабатывать клиентов по одному и продвигаться вперед небольшими шагами. В каждом клиенте могут понадобиться различные изменения, что делает задачу более сложной. Все ссылки на статические переменные должны быть заменены. Таким образом,

```
Person thePerson = new Person(Person.A);
```

превращается в

```
Person thePerson = new Person(BloodGroup.A);
```

Обращения к методу получения значения должны использовать новый метод, поэтому

```
thePerson.getBloodGroupCode();
```

превращается в

```
thePerson.getBloodGroup().getCode();
```

То же относится и к методам установки значения, поэтому

```
thePerson.setBloodGroup(Person.AB);
```

превращается в

```
thePerson.setBloodGroup(BloodGroup.AB);
```

Проделав это для всех клиентов Person, можно удалить метод получения значения, конструктор, статические определения и методы установки значения, использующие целые 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);}
}
```

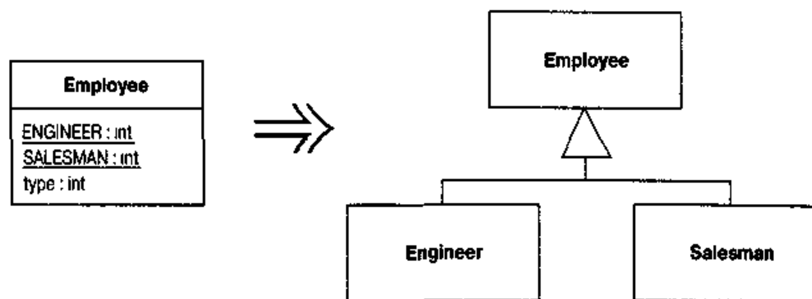
Можно также объявить закрытыми методы класса группы крови, использующие целочисленный код:

```
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, воздействующий на поведение класса.

Замените код типа подклассами.



Мотивировка

Если код типа не оказывает влияния на поведение, можно воспользоваться «Заменой кода типа классом» (Replace Type Code with Class). Однако если код типа влияет на поведение, то вариантное поведение лучше всего организовать с помощью полиморфизма.

Обычно на такую ситуацию указывает присутствие условных операторов типа case. Это могут быть переключатели или конструкции типа if-then-else. В том и другом случае они проверяют значение кода типа и в зависимости от результата выполняют различные действия. К таким участкам кода должен быть применен рефакторинг «Замена условного оператора полиморфизмом» (Replace Conditional with Polymorphism). Чтобы действовал данный рефакторинг, код типа должен быть заменен иерархией наследования, содержащей полиморфное по ведение. В такой иерархии наследования имеются подклассы для ка дого кода типа.

Простейший способ организовать такую структуру - «Замена к типа подклассами» (Replace Type Code with Subclasses). Надо взять класс, имеющий этот код типа, и создать подкласс для каждого типа. Однако бывают случаи, когда сделать это нельзя. Один из имеет место, когда код типа изменяется после создания объекта Другой - при наличии класса, для кода типа которого уже созданы подклассы по другим причинам. В обоих случаях надо пряменять «Замену кода типа состоянием/стратегией» (Replace Type Code with State/Strategy).

«Замена кода типа подклассами» (Replace Type Code with Subclasses) служит в основном вспомогательным шагом, позволяющим осуществить «Замену условного оператора полиморфизмом» (Replace Conditional with

Polymorphism). Побудить к применению «Замены кода типа подклассами» (Replace Type Code with Subclasses) должно наличие условных операторов. Если условных операторов нет, более правильным и менее критическим шагом будет «Замена кода типа классом» (Replace Type Code with Class).

Другим основанием для «Замены кода типа подклассами» (Replace Type Code with Subclasses) является наличие функций, относящихся только к объектам с определенными кодами типа. Осуществив этот рефакторинг, можно применить «Спуск метода» (Push Down Method) и «Спуск поля» (Push Down Field), чтобы стало яснее, что эти функции касаются только определенных случаев.

«Замена кода типа подклассами» (Replace Type Code with Subclasses) полезна тем, что перемещает осведомленность о вариантном поведении из клиентов класса в сам класс. При добавлении новых вариантов необходимо лишь добавить подкласс. В отсутствие полиморфизма пришлось бы искать все условные операторы и изменять их. Поэтому данный рефакторинг особенно полезен, когда варианты непрерывно изменяются.

Техника

Выполните самоинкапсуляцию кода типа.

Если код типа передается конструктору, необходимо заменить конструктор фабричным методом.

Создайте подкласс для каждого значения кода типа. Замените метод получения кода типа в подклассе, чтобы он возвращал надлежащее значение.

Данное значение жестко прошито в return (например, return 1). Может показаться непривлекательным, но это лишь временная мера до замены всех вариантов case.

После замены каждого значения кода типа подклассом выполните компиляцию и тестирование.

Удалите поле кода типа из родительского класса. 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;
 }
}
```

На первом шаге к коду типа применяется «Самоинкапсуляция поля» (Self Encapsulate Field):

```
int getType() {
 return _type;
}
```

Поскольку конструктор Employee использует код типа а качестве параметра, надо заменить его фабричным методом:

```
Employee create(int type) {
 return new Employee(type);
}
private Employee (int type) {
 _type = type;
}
```

Теперь можно приступить к преобразованию Engineer в подкласс. Сначала создается подкласс и замещающий метод для кода типа:

```
class Engineer extends Employee {
 int getType() {
    return Employee.ENGINEER;
 }
}
```

Необходимо также заменить фабричный метод, чтобы он создавал надлежащий объект:

```
class Employee {
 static Employee create(int type) {
    if (type == ENGINEER) return new Engineer();
    else return new Employee(type);
 }
}
```

Продолжаю поочередно, пока все коды не будут заменены подклассами. После этого можно избавиться от поля с кодом типа в Employee и сделать getType абстрактным методом. После этого фабричный метод будет выглядеть так:

```
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("Недопустимое значение кода типа");
 }
}
```

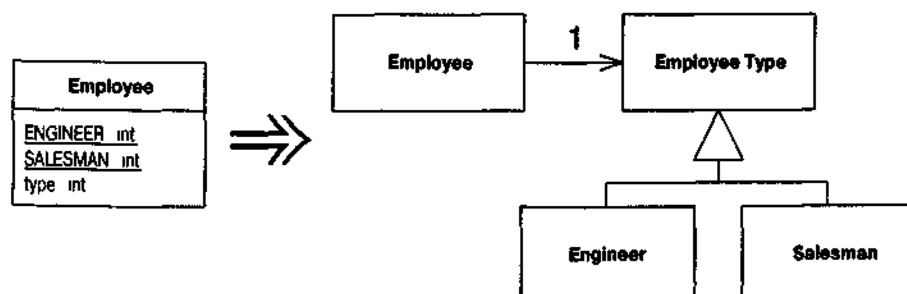
Конечно, такого оператора switch желательно было бы избежать. Но он здесь лишь один и используется только при создании объекта.

Естественно, что после создания подклассов следует применить «Спуск метода» (Push Down Method) и «Спуск поля» (Push Down Field) ко всем методам и полям, которые относятся только к конкретным типам служащих.

Замена кода типа состоянием/стратегией (Replace Type Code with State/Strategy)

Есть код типа, влияющий на поведение класса, но нельзя применить создание подклассов.

Замените код типа объектом состояния.



Мотивировка

Этот рефакторинг сходен с «Заменой кода типа подклассами» (Replace Type Code with Subclasses), но может быть применен, когда код типа изменяется на протяжении существования объекта или если созданию подклассов мешают другие причины. В ней применяется паттерн состояния или стратегии [Gang of Four].

Состояние и стратегия очень схожи между собой, поэтому рефакторинг будет одинаковым в обоих случаях. Выберите тот паттерн, который лучше всего подходит в конкретной ситуации. Если вы пытаетесь упростить отдельный алгоритм с помощью «Замены условного оператора полиморфизмом» (Replace Conditional with Polymorphism), лучше использовать стратегию. Если вы собираетесь переместить специфические для состояния данные и представляете себе объект как изменяющий состояние, используйте паттерн состояния.

Техника

Выполните самоинкапсуляцию кода типа.

Создайте новый класс и дайте ему имя в соответствии с назначением кода типа. Это будет объект состояния.

Добавьте подклассы объекта состояния, по одному для каждого кода типа.

Проще добавить сразу все подклассы, чем делать это по одному.

Создайте абстрактный метод получения данных объекта состояния для возврата кода типа. Создайте перегруженные методы получения данных для каждого подкласса объекта состояния, возвращающие правильный код типа.

Выполните компиляцию.

Создайте в старом классе поле для нового объекта состояния.

Настройте метод получения кода типа в исходном классе так, чтобы обработка делегировалась объекту состояния.

Настройте методы установки кода типа в исходном классе так, чтобы осуществлялось присваивание экземпляра надлежащего подкласса объекта состояния.

Выполните компиляцию и тестирование.

Пример

Я снова обращаюсь к утомительному и глупому 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("Не тот служащий});
     }
   }
```

Предполагается, что это замечательная, прогрессивная компания, позволяющая менеджерам вырастать до инженеров. Поэтому код типа изменяем, и применять подклассы нельзя. Первым делом, как всегда, самоинкапсулируется код 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("Incorrect Employee");
    }
   }
```

Теперь я объявляю класс состояния (как абстрактный класс с абстрактным методом возврата кода типа):

```
   abstract class EmployeeType {
    abstract int getTypeCode();
   }
```

Теперь создаются 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 настолько тривиально, что даже у меня все компилируется легко. Теперь я фактически подключаю подклассы к Employee, модифицируя 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("Неправильный код служащего");
     }
  }
 }
```

Это означает, что здесь у меня получилось утверждение типа switch. После завершения рефакторинга оно окажется единственным в коде и будет выполняться только при изменении типа. Можно также воспользоваться «Заменой конструктора фабричным методом» (Replace Constructor with Factory Method), чтобы создать фабричные методы для разных случаев. Все другие утверждения case можно удалить, введя в атаку «Замену условных операторов полиморфизмом» (Replace Conditional with Polymorphism).

Я люблю завершать «Замену кода типа состоянием/стратегией» (Replace Type Code with State/Strategy), перемещая в новый класс всю информацию о кодах типов и подклассы. Сначала я копирую определения кодов типов в тип служащего, создаю фабричный метод для типов служащих и модифицирую метод установки 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("Incorrect Employee Code");
     }
  }
  static final int ENGINEER = 0;
  static final int SALESMAN = 1;
  static final int MANAGER = 2;
 }
```

После этого я удаляю определения кода типа Employee и заменяю их ссылками на тип служащего:

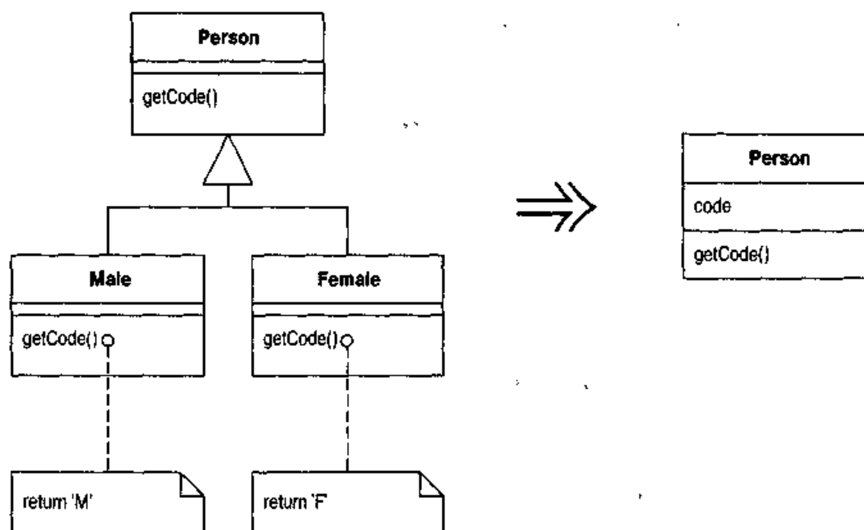
```
class Employee {
 int payAmount() {
   switch (getType()) {
      case EmployeeType.ENGINEER: return _mnonthlySalary;
      case EmployeeType.SALESMAN: return _monthlySalary + _commission;
      case EmployeeType.MANAGER: return _monthlySalary + _bonus;
      default: throw new RuntimeException("Incorrect Employee");
   }
 }
}
```

Теперь я готов применить «Замену условного оператора полиморфизмом» (Replace Conditional with Polymorphism) к payAmount.

Замена подкласса полями (Replace Subclass with Fields)

Есть подклассы, которые различаются только методами, возвращающими данные-константы.

Замените методы полями в родительском классе и удалите подклассы.



Мотивировка

Подклассы создаются, чтобы добавить новые функции или позволить меняться поведению. Одной из форм вариантного поведения служит константный метод [Beck]. Константный метод - это такой метод, который возвращает жестко закодированное значение. Он может быть очень полезным в подклассах, возвращающих различные значения для метода доступа. Вы определяете метод доступа в родительском классе и реализуете его с помощью различных значений в подклассах.

Несмотря на полезность константных методов, подкласс, состоящий только из них, делает слишком мало, чтобы оправдать свое существование. Такие подклассы можно полностью удалить, поместив в родительский класс поля. В результате убирается дополнительная сложность, связанная с подклассами.

Техника

Примените к подклассам «Замену конструктора фабричным методом» (Replace Constructor with Factory Method).

Если есть код со ссылками на подклассы, замените его ссылками на родительский класс.

Для каждого константного метода в родительском классе объявите поля вида final.

Объявите защищенный конструктор родительского класса для инициализации полей.

Создайте или модифицируйте имеющиеся конструкторы подклассов, чтобы они вызывали новый конструктор родительского класса.

Выполните компиляцию и тестирование.

Реализуйте каждый константный метод в родительском классе так, чтобы он возвращал поле, и удалите метод из подкласса.

После каждого удаления выполняйте компиляцию и тестирование.

После удаления всех методов подклассов примените «Встраивание метода» (Inline Method) для встраивания конструктора в фабричный метод родительского класса.

Выполните компиляцию и тестирование.

Удалите подкласс.

Выполните компиляцию и тестирование.

Повторяйте встраивание конструктора и удаление подклассов до тех пор, пока их больше не останется.

Пример

Начну с Person и подклассов, выделенных по полу:

```
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 есть реализации абстрактного метода, возвращающие жестко закодированные константы. Я удалю эти ленивые подклассы [Beck].

Сначала следует применить «Замену конструктора фабричным методом» (Replace Constructor with Factory Method). 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();
```

следующими:

```
Person kent = Person.createMale();
```

После замены всех этих вызовов не должно остаться ссылок на подклассы. Это можно проверить с помощью текстового поиска, а сделав класс закрытым, убедиться, что, по крайней мере, к ним нет обращений извне пакета.

Теперь объявляю поля для каждого константного метода в родительском классе:

```
class Person {
 private final boolean _isMale;
 private final char _code;
```

```
}
```

Добавляю в родительский класс защищенный конструктор:

```
class Person {
 protected Person (boolean isMale, char code) {
    _isMale = isMale;
    _code = code;
 }
}
```

Добавляю конструкторы, вызывающие этот новый конструктор:

```
class Male {
 Male() {
    super (true, 'M');
 }
}
class Female {
 Female() {
    super (false, 'F');
 }
}
```

После этого можно выполнить компиляцию и тестирование. Поля создаются и инициализируются, но пока они не используются. Я могу теперь ввести поля в игру, поместив в родительском классе методы доступа и удалив методы подклассов:

```
class Person {
 boolean isMale() { return _isMale;}
}
class Male {
 boolean isMale() { return true;}
}
```

Можно делать это по одному полю и подклассу за проход или, чувствуя уверенность, со всеми полями сразу.

В итоге все подклассы оказываются пустыми, поэтому я снимаю пометку abstract с класса Person и с помощью «Встраивания метода» (Inine Method) встраиваю конструктор подкласса в родительский класс:

```
class Person {
 static Person createMale(){
    return new Person(true, 'M');
 }
}
```

После компиляции и тестирования класс Male удаляется, и процедура повторяется для класса Female.

9 УПРОЩЕНИЕ УСЛОВНЫХ ВЫРАЖЕНИЙ

Логика условного выполнения имеет тенденцию становиться сложной, поэтому ряд рефакторингов направлен на то, чтобы упростить ее. Базовым рефакторингом при этом является «Декомпозиция условного оператора» (Decompose Conditional), цель которого состоит в разложении условного оператора на части. Ее важность в том, что логика переключения отделяется от деталей того, что происходит.

Остальные рефакторинги этой главы касаются других важных случаев. Применяйте «Консолидацию условного выражения» (Consolidate Conditional Expression), когда есть несколько проверок и все они имеют одинаковый результат. Применяйте «Консолидацию дублирующихся условных фрагментов» (Consolidate Duplicate Conditional Fragments), чтобы удалить дублирование в условном коде.

В коде, разработанном по принципу «одной точки выхода», часто обнаруживаются управляющие флаги, которые дают возможность условиям действовать согласно этому правилу. Поэтому я применяю «Замену вложенных условных операторов граничным оператором» (Replace Nested Conditional with Guard Clauses) для прояснения особых случаев условных операторов и «Удаление управляющего флага» (Remove Control Flag) для избавления от неудобных управляющих флагов.

В объектно-ориентированных программах количество условных операторов часто меньше, чем в процедурных, потому что значительную часть условного поведения выполняет полиморфизм. Полиморфизм имеет следующее преимущество: вызывающему не требуется знать об условном поведении, а потому облегчается расширение условий. В результате в объектно-ориентированных программах редко встречаются операторы switch (case). Те, которые все же есть, являются главными кандидатами для проведения «Замены условного оператора полиморфизмом» (Replace Conditional with Polymorphism). Одним из наиболее полезных, хотя и менее очевидных применений полиморфизма является «Введение объекта Null» (Introduce Null Object), чтобы избавиться от проверок на нулевое значение.

Декомпозиция условного оператора (Decompose Conditional)

Имеется сложная условная цепочка проверок (if-then-else). Выделите методы из условия, части «then» и частей «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);
```

Мотивировка

Очень часто сложность программы обусловлена сложностью условной логики. При написании кода, обязанного проверять условия и делать в зависимости от условий разные вещи, мы быстро приходим к созданию довольно длинного метода. Длина метода сама по себе осложняет его чтение, но если есть условные выражения, трудностей становится еще больше. Обычно проблема связана с тем, что код, как в проверках условий, так и в действиях, говорит о том, что происходит, но легко может затенять причину, по которой это происходит. Как и в любом большом блоке кода, можно сделать свои намерения более ясными, если выполнить его декомпозицию и заменить фрагменты кода вызовами методов, имена которых раскрывают назначение соответствующего участка кода. Для кода с условными операторами выгода еще больше, если проделать это как для части, образующей условие, так и для всех альтернатив. Таким способом можно выделить условие и ясно обозначить, 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 вложенным условным оператором, я обычно смотрю сначала, не надо ли выполнить «Замену вложенных условных операторов граничным оператором» (Replace Nested Conditional with Guard Clauses). Если в такой замене смысла нет, я провожу декомпозицию каждого условного оператора.

Пример

Допустим, требуется вычислить плату за какую-то услугу, для которой есть отдельный зимний и летний тариф:

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
 charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;
```

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 проведения рефакторинга. Однако на практике я провожу каждое выделение изолированно, выполняя после него компиляцию и тестирование.

Многие программисты в таких ситуациях не выделяют части, образующие условие. Часто условия кажутся короткими и не стоящими такого труда. 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 ясную информацию, чем из исходного кода. Если работать с оригиналом, приходится смотреть в код и разбираться в том, что он делает. В данном случае это сделать нетрудно, но даже здесь выделенный метод более похож на комментарий.

Консолидация условного выражения (Consolidate Conditional Expression)

Есть ряд проверок условия, 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;
 // compute the disability amount
}
```

```
double disabilityAmount() {
 if (isNotEligableForDisability() return 0;
 // compute the disability amount
}
```

Мотивировка

Иногда встречается ряд проверок условий, в котором все проверки различны, но результирующее действие одно и то же. Встретившись с этим, необходимо с помощью логических операций «и»/*или» объединить проверки в одну проверку условия, возвращающую один результат. Объединение условного кода важно по двум причинам. Во-первых, проверка становится более ясной, показывая, что в действительности проводится одна проверка, в которой логически складываются результаты других. Последовательность имеет тот же

результат, но говорит о том, что выполняется ряд отдельных проверок, которые случайно оказались вместе. Второе основание для проведения этого рефакторанга состоит в том, что он часто подготавливает почву для «Выделения метода» (Extract Method). Выделение условия - одно из наиболее полезных для прояснения кода действий. Оно заменяет изложение выполняемых действий причиной, по которой они выполняются. Основания в пользу консолидации условных выражений указывают также на причины, по которым ее выполнять не следует. Если вы считаете, что проверки действительно независимы и не должны рассматриваться как одна проверка, не производите этот рефакторинг. Имеющийся код уже раскрывает ваш замысел.

Техника

Убедитесь, что условные выражения не несут побочных эффектов.

Если побочные эффекты есть, вы не сможете в факторинг.

Замените последовательность условий одним условным предложением с помощью логических операторов.

Выполните компиляцию и тестирование.

Изучите возможность применения к условию «Выделения метода» (Extract Method).

Пример: логическое «или»

Имеется следующий код:

```
double disabilityAmount() {
 if (_semority < 2) return 0;
 if (_monthsDisabled > 12) return 0;
 if (_isPartTime) return 0;
// вычислить сумму оплаты по нетрудоспособности
}
```

Мы видим ряд условных операторов, возвращающих одинаковый результат. Для кода в такой последовательности проверки эквивалентны предложению с операцией « или»:

```
double disabilityAmount() {
 if ((_seniority < 2) || (_monthsDisabled > 12) || (_isPartTime))
    return 0;
 // вычислить сумму оплаты по нетрудоспособности
}
```

Взгляд на условие показывает, что, применив «Выделение метода» (Extract Method), можно сообщить о том, что ищет условный оператор (нетрудоспособность не оплачивается):

```
double disabilityAmount() {
 if (isNotEligibleForDisability()) return 0
 // вычислить сумму оплаты по нетрудоспособности
}
boolean isNotEligibleForDisability() {
 return ((_seniority < 2) || (_monthsDisabled > 12) || (_isPartTime));
}
```

Пример: логическое «и»

Предыдущий пример демонстрировал «или», но то же самое можно делать с помощью «и». Пусть ситуация выглядит так:

```
if (onVacation())
 if (lengthOfService() > 10) return 1;
return 0.5;
```

Этот код надо заменять следующим:

```
if (onVacation() && lengthOfService() > 10) return 1;
else return 0.5;
```

Часто получается комбинированное выражение, содержащее «и», «или» и «не». В таких случаях условия бывают запутанными, поэтому я стараюсь их упростить с помощью «Выделения метода» (Extract Method).

Если рассматриваемая процедура лишь проверяет условие и возвращает значение, я могу превратить ее в одно предложение 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.

Переместите 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();
```

Мотивировка

Иногда обнаруживается, что во всех ветвях условного оператора выполняется один и тот же фрагмент кода. В таком случае следует переместить этот код за пределы условного оператора. В результате становится яснее, что меняется, а что остается постоянным.

Техника

Выявите код, который выполняется одинаковым образом вне зависимости от значения условия.

Если общий код находится в начале, поместите его перед условным оператором.

Если общий код находится в конце, поместите его после условного оператора.

Если общий код находится в середине, посмотрите, модифицирует ли что-нибудь код, находящийся до него или после него. Если да, то общий код можно переместить до конца вперед или назад. После этого можно его переместить, как это описано для кода, находящегося в конце или в начале.

Если код состоит из нескольких предложений, надо выделить его в метод.

Пример

Данная 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 условного оператора:

```
if(isSpecialDeal())
 total = price * 0.95;
else
 total = price * 0.98;
send();
```

То же самое может возникать с исключительными ситуациями. Если код повторяется после оператора, вызывающего исключительную ситуацию, в блоке try и во всех блоках catch, можно переместить его в блок final.

Удаление управляющего флага (Remove Control Flag)

Имеется переменная, действующая как управляющий флаг для ряда булевых выражений.

Используйте вместо нее break или return.

Мотивировка

Встретившись с серией условных выражений, часто можно обнаружить управляющий флаг, с помощью которого определяется окончание просмотра:

```
set done to false
while not done
 if (condition)
    do something
    set done to true
 next step of loop
```

От таких управляющих флагов больше неприятностей, чем пользы. Их присутствие диктуется правилами структурного программирования, согласно которым в процедурах должна быть одна точка входа и одна точка выхода. Я согласен с тем, что должна быть одна точка входа (чего требуют современные языки программирования), но требование одной точки выхода приводит к сильно запутанным условным операторам, в коде которых есть такие неудобные флаги. Для того чтобы выбраться из сложного условного оператора, в языках есть команды break и continue. Часто бывает удивительно, чего можно достичь, избавившись от управляющего флага. Действительное назначение условного оператора становится гораздо понятнее.

Техника

Очевидный способ справиться с управляющими флагами предоставляют имеющиеся в Java операторы break и continue.

Определите значение управляющего флага, при котором происходит выход из логического оператора.

Замените присваивания значения для выхода операторами break или continue.

Выполняйте компиляцию и тестирование после каждой замены.

Другой подход, применимый также в языках без операторов break и continue, состоит в следующем:

Выделите логику в метод.

Определите значение управляющего флага, при котором происходит выход из логического оператора.

Замените присваивания значения для выхода оператором return.

Выполняйте компиляцию и тестирование после каждой замены.

Даже в языках, где есть break или continue, я обычно предпочитаю применять выделение и return. Оператор return четко сигнализирует, что никакой код в методе больше не выполняется. При наличии кода такого вида часто в любом случае надо выделять этот фрагмент.

Следите за тем, не несет ли управляющий флаг также информации о результате. Если это так, то управляющий флаг все равно необходим, либо можно возвращать это значение, если вы выделили метод.

Пример: замена простого управляющего флага оператором break

Следующая функция проверяет, не содержится ли в списке лиц кто-либо из парочки подозрительных, имена которых жестко закодированы:

```
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 таких случаях заметить управляющий флаг легко. Это фрагмент, в котором переменной found присваивается значение true. Ввожу операторы break по одному:

```
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;}
    }
 }
}
```

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 ссылки на управляющий флаг:

```
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 данного рефакторинга использует оператор return. Проиллюстрирую это вариантом, 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 found служит двум целям: задает результат и действует в качестве управляющего флага. Если я вижу такой код, то предпочитаю выделить 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 заменять 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 return возможно и тогда, когда нет еобходимости возвращать значение - просто используйте return без аргумента.

Конечно, здесь возникают проблемы, связанные с побочными эффектами функции. Поэтому надо применять «Разделение запроса и модификатора» (Separate Query from Modifier). Данный пример будет продолжен в соответствующем разделе.

Замена вложенных условных операторов граничным оператором (Replace Nested Conditional with Guard Clauses)

Метод использует условное поведение, из которого неясен нормальный путь выполнения.

Используйте граничные условия для всех особых случаев.

```
 double getPayAmount() {
  double result;
  if(_isDead) result = deadAmount();
  else {
     if(_isSeparated) result = separatedAmount();
     else {
        if(_isRetired) result = retiredAmount();
        else result = normalAmount();
     }
  }
  return result;
 }
```

```
 double getPayAmount() {
  if(_isDead) return deadAmount();
  if(_isSeparated) result = separatedAmount();
  if(_isRetired) result = retiredAmount();
  return normalAmount();
 }
```

Мотивировка

Часто оказывается, что условные выражения имеют один из двух видов. В первом виде это проверка, при которой любой выбранный ход событий является частью нормального поведения. Вторая форма представляет собой ситуацию, в которой один результат условного оператора указывает на нормальное поведение, а другой - на необычные условия.

Эти виды условных операторов несут в себе разный смысл, и этот смысл должен быть виден в коде. Если обе части представляют собой нормальное поведение, используйте условие с ветвями if и else. Если условие является необычным, проверьте условие и выполните return, если условие истинно. Такого рода проверка часто называется граничным оператором (guard clause [Beck]).

Главный смысл «Замены вложенных условных операторов граничным оператором» (Replace Nested Conditional with Guard Clauses) состоит в придании выразительности. При использовании конструкции if-then-else ветви if и ветви else придается равный вес. Это говорит читателю, что обе ветви обладают равной вероятностью и важностью. Напротив, защитный оператор говорит: «Это случается редко, и если все-таки произошло, надо сделать то-то и то-то и выйти».

Я часто прибегаю к «Замене вложенных условных операторов граничным оператором» (Replace Nested Conditional with Guard Clauses), когда работаю с программистом, которого учили, что в методе должны быть только одна точка входа и одна точка выхода. Одна точка входа обеспечивается современными языками, а в правиле одной точки выхода на самом деле пользы нет. Ясность - главный принцип: если метод понятнее, когда в нем одна точка выхода, сделайте ее единственной, в противном случае не стремитесь к этому.

Техника

Для каждой проверки вставьте граничный оператор.

Граничный оператор осуществляет возврат или возбуждает исключительную ситуацию.

Выполняйте компиляцию и тестирование после каждой замены проверки граничным оператором.

Если все граничные операторы возвращают одинаковый результат, примените «Консолидацию условных выражений» (Consolidate Conditional Expression).

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 = normalAmount();
    }
 }
 return result;
}
```

В этом коде проверки маскируют выполнение обычных действий, поэтому при использовании граничных операторов код станет яснее. Буду вводить их по одному и начну 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 замены по одной на каждом шагу:

```
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 я ее убиваю:

```
double getPayAmount() {
 if (_isDead) return deadAmount();
 if (_isSeparated) return separatedAmoun();
 if (_isRetired) return retiredAmount();
 return normalPayAmount();
}
```

Вложенный условный код часто пишут программисты, которых учили, что в методе должна быть только одна точка выхода. Я считаю, что это слишком упрощенное правило. Если метод больше не представляет для меня интереса, я указываю на это путем выхода из него. Заставляя читателя рассматривать пустой блок else, вы только создаете преграды на пути понимания кода.

Пример: обращение условий

Рецензируя рукопись этой книги, Джошуа Кериевски (Joshua Kerievsky) отметил, что часто «Замена вложенных условных операторов граничными операторами» (Replace Nested Conditional with Guard Clauses) осуществляется путем обращения условных выражений. Он любезно предоставил пример, что позволило мне не подвергать свое воображение дальнейшему испытанию:

```
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 буду выполнять замену поочередно, но на этот раз при вставке граничного оператора условие обращается:

```
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 буду обращать его в два этапа. Сначала я ввожу 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 условном операторе сохраняются такие отрицания, у меня мозги съезжают набекрень, поэтому я 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 помещать явные значения в операторы возврата из граничных операторов. Благодаря этому можно легко видеть результат, получаемый при срабатывании защиты (я бы также попробовал здесь применить «Замену магического числа символической константой» (Replace Magic Number with Symbolic Constant).

```
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 также удалить временную переменную:

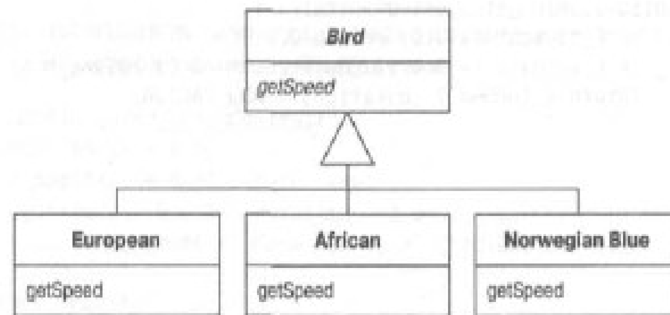
```
public double getAdjustedCapital() {
 if (_capital <= 0.0) return 0.0;
 if (_intRate <= 0.0 || _duration <= 0.0) return 0.0;
 return (_income / _duration) * ADJ_FACTOR;
}
```

Замена условного оператора полиморфизмом (Replace Conditional with Polymorphism)

Есть условный оператор, поведение которого зависит от типа объекта. Переместите каждую ветвь условного оператора в перегруженный 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 ("Should be unreachable");
   }
```



Мотивировка

Одним из наиболее внушительно звучащих слов из жаргона объектного программирования является полиморфизм. Сущность полиморфизма состоит в том, что он позволяет избежать написания явных условных операторов, когда есть объекты, поведение которых различно в зависимости от их типа.

В результате оказывается, что операторы switch, выполняющие переключение в зависимости от кода типа, или операторы if-then-else, выполняющие переключение в зависимости от строки типа, в объектно-ориентированных программах встречаются значительно реже.

Полиморфизм дает многие преимущества. Наибольшая отдача 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 условные операторы. Но при использовании подклассов достаточно создать новый подкласс и обеспечить в нем соответствующие методы. Клиентам класса не надо знать о подклассах, благодаря чему сокращается количество зависимостей в системе и упрощается ее модификация.

Техника

Прежде чем применять «Замену условного оператора полиморфизмом» (Replace Conditional with Polymorphism), следует создать необходимую иерархию наследования. Такая иерархия может уже иметься как результат ранее проведенного рефакторинга. Если этой иерархии нет, ее надо создать.

Создать иерархию наследования можно двумя способами: «Заменой кода типа подклассами» (Replace Type Code with Subclasses) и «Заменой кода типа состоянием/стратегией» (Replace Type Code with State/Strategy). Более простым вариантом является создание подклассов, поэтому по возможности следует выбирать его. Однако если код типа изменяется после того, как создан объект, применять создание подклассов нельзя, и необходимо применять паттерн «состояния/стратегии». Паттерн «состояния/стратегии» должен использоваться и тогда, когда подклассы данного класса уже создаются по другим причинам. Помните, что если несколько операторов case выполняют переключение по одному и тому же коду типа, для этого кода типа нужно создать лишь одну иерархию наследования.

Теперь можно предпринять атаку на условный оператор. Код, на который вы нацелились, может быть оператором switch (case) или оператором if.

Если условный оператор является частью более крупного метода, разделите условный оператор на части и примените «Выделение метода» (Extract Method).

При необходимости воспользуйтесь перемещением метода, чтобы поместить условный оператор в вершину иерархии наследования.

Выберите один из подклассов. Создайте метод подкласса, перегружающий метод условного оператора. Скопируйте тело этой ветви условного оператора в метод подкласса и настройте его по месту.

Для этого может потребоваться сделать некоторые закрытые члены надкласса защищенными.

Выполните компиляцию и тестирование.

Удалите скопированную ветвь из условного оператора.

Выполните компиляцию и тестирование.

Повторяйте эти действия с каждой ветвью условного оператора, пока все они не будут превращены в методы подкласса.

Сделайте метод родительского класса абстрактным.

Пример

Я обращусь к скучному и упрощенному примеру расчета зарплаты служащих. Применяются классы, полученные после «Замены кода типа состоянием/стратегией» (Replace Type Code with State/Strategy), поэтому объекты выглядят так, как показано на рисунке 9.1 (начало см. в примере из главы 8).

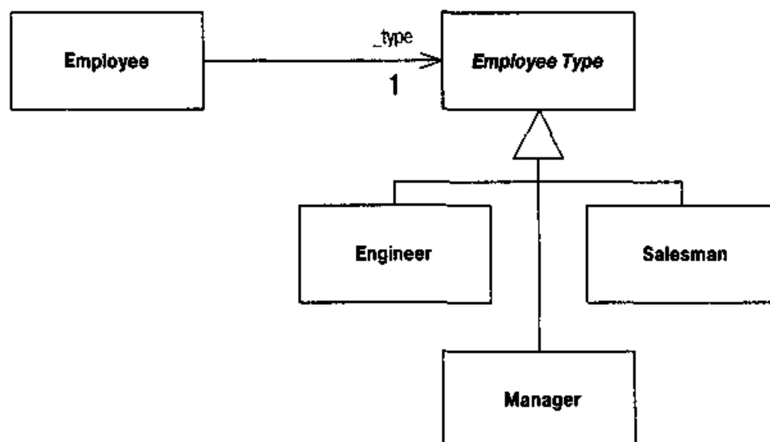


Рисунок 9.1 - Структура наследования

```
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

Оператор case уже должным образом выделен, поэтому с этим делать ничего не надо. Что надо, так это переместить его в EmployeeType, поскольку это тот класс, для которого создаются 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 Employee, я должен передать его в качестве аргумента. Некоторые из этих данных можно переместить в объект типа Employee, но это тема другого рефакторинга.

Если компиляция проходит успешно, я изменяю метод payAmount в классе Employee, чтобы он делегировал обработку новому классу:

```
  class Employee {
   int payAmount() {
      return _type.payAmount(this);
   }
  }
```

Теперь можно заняться оператором case. Это будет похоже на расправу мальчишек с насекомыми - я буду отрывать ему «ноги» одну за другой. Сначала я скопирую ветвь Engineer оператора case 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 ставлю ловушку в операторе case:

```
  class EmployeeType {
   int payAmount(Employee emp) {
      switch (getTypeCode()) {
         case ENGINEER:
            throw new RuntimeException ("Должна происходить перегрузка");
         case SALESMAN:
            return emp.getMonttilySalary() + emp.getCommission();
         case MANAGER:
            return emp.getMonthlySalary() + emp.getBonus();
         default:
            throw new RuntimeException("Incorrect Employee");
      }
   }
  }
```

Продолжаем выполнение, пока не будут удалены все «ноги» (ветви):

```
  class Salesman {
```

```
 int payAmount(Employee emp) {
    return emp.getMonthlySalary() + emp.getCommission();
 }
}
class Manager {
 int payAmount(Employee emp) {
    return emp.getMonthlySalary() + emp.getBonus ();
 }
}
```

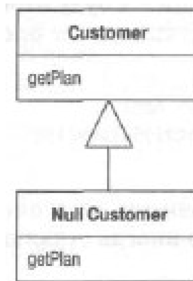
А теперь делаем метод родительского класса абстрактным:

```
class EmployeeType {
 abstract int payAmount(Employee emp);
 }
}
```

Введение объекта Null (Introduce Null Object)

Есть многократные проверки совпадения значения с null. Замените значение null объектом null.

```
if(customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```



Мотивировка

Сущность полиморфизма в том, что вместо того, чтобы спрашивать у объекта его тип и вызывать то или иное поведение в зависимости от ответа, вы просто вызываете поведение. Объект, в зависимости от своего типа, делает то, что нужно. Все это не так прозрачно, когда значением поля является null. Пусть об этом расскажет Рон Джеффриз:

Рон Джеффриз

Мы впервые стали применять паттерн нулевого объекта, когда Рич Гарзанити (Rich Garzaniti) обнаружил, что код системы часто проверяет, существует ли объект, прежде чем послать ему сообщение. Мы запрашивали у объекта его метод person, а затем сравнивали результат с null. Если объект присутствовал, мы запрашивали у него метод rate. Это делалось в нескольких местах, и повторение кода в них стало нас раздражать.

Поэтому мы создали объект отсутствующего лица, который сообщал, что у него нулевой rate (мы называем наши null-объекты отсутствующими объектами). Вскоре у отсутствующего лица было уже много методов, например rate. Сейчас у нас больше 80 классов нулевых объектов.

Чаще всего мы применяем нулевые объекты при выводе информации. Например, когда выводится информация о лице, то у соответствующего объекта может отсутствовать любой из примерно 20 атрибутов. Если бы они могли принимать значение null, вывод информации о лице стал бы очень сложным. Вместо этого мы подключаем различные нулевые объекты, которые умеют отображать себя правильным образом. В результате мы избавились от большого объема процедурного кода.

Наиболее остроумно мы применяем нулевой объект для отсутствующего сеанса Gemstone. Мы пользуемся базой данных Gemstone для готового кода, но разработку предпочитаем вести без нее и сбрасываем новый код в Gemstone примерно раз в неделю. 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. Другое 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 буфер. Отсутствующий буфер знает, что у него нулевое сальдо и нет значений. При таком подходе удается избежать создания десятков пустых буферов для каждого из тысяч наших служащих.

Интересная особенность применения нулевых объектов состоит в том, что 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 обнаружите нулевой объект, которого там быть не должно. Помните, что нулевые объекты постоянны - в них никогда ничего не меняется. Соответственно, мы реализуем их по паттерну «Одиночка» (Singleton pattern) [Gang of Four]. Например, при каждом запросе отсутствующего лица вы будете получать один и тот же экземпляр этого класса.

Подробнее о паттерне нулевого объекта можно прочесть у Вулфа [Woolf].

Техника

Создайте подкласс исходного класса, который будет выступать как нулевая версия класса. Создайте операцию isNull в исходном классе и нулевом классе. В исходном классе она должна возвращать false, а в нулевом классе - true.

Удобным может оказаться создание явного нулевого интерфейса для метода isNull.

Альтернативой может быть использование проверочного интерфейса для проверки на null.

Выполните компиляцию.

Найдите все места, где при запросе исходного объекта может возвращаться null, и отредактируйте их так, чтобы вместо этого возвращался нулевой объект.

Найдите все места, где переменная типа исходного класса сравнивается с null, и поместите в них вызов isNull.

Это можно сделать, заменяя поочередно каждый исходный класс вместе с его клиентами и выполняя компиляцию и тестирование после каждой замены.

Может оказаться полезным поместить нескольким утверждений assert, проверяющих на null, в тех местах, где значение null теперь не должно встречаться.

Выполните компиляцию и тестирование.

Найдите случаи вызова клиентами операции if not null и осуществления альтернативного поведения if null.

Для каждого из этих случаев замените операции в нулевом классе альтернативным поведением.

Удалите проверку условия там, где используется перегруженное поведение, 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;
}
```

У пользователя есть несколько характеристик. Возьмем три из них

```
class Customer {
 public String getName() {  }
 public BillingPlan getPlan() {  }
 public PaymentHistory getHistory() {  }
}
```

У истории платежей PaymentHistory есть свои собственные функции, например количество недель просрочки в прошлом году:

```
public class PaymentHistory {
 int getWeeksDelinquentInLastYear() {...}
}
```

Показываемые мной методы доступа позволяют клиентам получить эти данные. Однако иногда у участка нет пользователя. Кто-то мог уехать, а о въехавшем пока сведений нет. Это может произойти, вот почему следует обеспечить обработку null любым кодом, который использует 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 много клиентов site и customer, и все они должны делать проверки на null, выполняя одинаковые действия при обнаружении таковых. Похоже, пора создавать нулевой объект.

Первым делом создаем нулевой класс для customer и модифицируем класс Customer, чтобы он поддерживал запрос проверки на null:

```
class NullCustomer extends Customer {
 public boolean isNull() { return true;}
}
class Customer {
 public boolean isNull() { return false;}
 protected Customer() {} //требуется для NullCustomer
}
```

Если нет возможности модифицировать класс Customer, можно воспользоваться тестирующим интерфейсом.

При желании можно возвестить об использовании нулевого объекта посредством интерфейса:

```
interface Nullable {
 boolean isNull() {return true;}
}
class Customer implements Nullable {...}
```

Для создания нулевых клиентов я хочу ввести фабричный метод, благодаря чему клиентам не обязательно будет знать о существовании нулевого класса:

```
class Customer static Customer newNull() {
 return new NullCustomer();
}
```

Теперь наступает трудный момент. Я должен возвращать этот новый нулевой объект вместо null и заменить проверки вида foo == null проверками вида foo.isNull(). Полезно поискать все места, где запрашивается customer, и модифицировать их так, чтобы возвращать нулевого пользователя вместо null.

```
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 isNull(), а не == 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 null и отредактировать их. Если объект интенсивно передается, их может быть нелегко проследить. Необходимо найти все переменные типа customer и все места их использования. Разбить эту процедуру на мелкие шаги трудно. Иногда я обнаруживаю один источник, который используется лишь в нескольких местах, и тогда можно заменить один этот источник. Однако чаще всего приходится делать много пространных изменений. Вернуться к старой версии обработки нулевых объектов не слишком сложно, потому что обращения к isNull можно найти без особого труда, но все равно это запутанная процедура.

Когда этот этап завершен, выполнены компиляция и тестирование, можно вздохнуть с облегчением. Теперь начинается приятное. В данный момент я ничего не выигрываю от применения isNull вместо == null. Выгода появится тогда, когда я перемещу поведение в нулевой customer и уберу условные операторы. Эти шаги можно выполнять по одному. Начну с имени. В настоящее время у меня такой код клиента:

```
String customerName;
if (customer.isNull()) customerName = "occupant";
else customerName = customer.getName();
```

Добавляю к нулевому customer подходящий метод имени:

```
class NullCustomer {
 public String getName(){ return "occupant";}
}
```

Теперь можно убрать условный код:

```
String customerName = customer.getName();
```

То же самое можно проделать с любым другим методом, в котором есть разумный общий ответ на запрос. Я могу также выполнить надлежащие действия для модификаторов. Поэтому клиентский код

```
if (!customer.isNull())
 customer.setPlan(BillingPlan.special())
```

можно заменить на

```
customer setPlan(BillingPlan.special())
class NullCustomer {
 public void setPlan (BillingPlan arg) {...}
}
```

Помните, что такое перемещение поведения оправдано только тогда, когда большинству клиентов требуется один и тот же ответ. Обратите внимание: я сказал «большинству», а не «всем». Те клиенты, которым нужен ответ, отличный от стандартного, могут по-прежнему выполнять проверку с помощью isNull. Выигрыш появляется тогда, когда многим клиентам требуется одно и то же; они могут просто полагаться на нулевое поведение по умолчанию.

В этом примере несколько иной случай - код клиента, в котором используется результат обращения к customer:

```
if (customer.isNullO) 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 нулевого клиента, чтобы он возвращал нулевой класс при запросе истории:

```
class NullCustomer {
 public PaymentHistory getHistory(){ return PaymentHistory.newNull();}
}
```

И снова можно удалить условный код:

```
int weeksDelinquent = customer.getHistory().getWeeksDelinquentInLastYear();
```

Часто оказывается, что одни нулевые объекты возвращают другие нулевые объекты.

Пример: проверяющий интерфейс

Проверяющий интерфейс служит альтернативой определению метода isNull. При этом подходе создается нулевой интерфейс, в котором не определены никакие методы:

```
interface Null {}
```

Затем я реализую нулевой интерфейс в своих нулевых объектах:

```
class NullCustomer extends Customer implements Null
```

Проверка на null осуществляется после этого с помощью оператора instanceof:

```
aCustomer instanceof Null
```

Обычно я с ужасом бегу от оператора instanceof, но в данном случае его применение оправданно. Его особое преимущество в том, что не надо изменять класс customer. Это позволяет пользоваться нулевым объектом даже тогда, когда отсутствует доступ к исходному коду customer.

Другие особые случаи

При выполнении данного рефакторинга можно создавать несколько разновидностей нулевого объекта. Часто есть разница между отсутствием customer (новое здание, в котором никто не живет) и отсутствием сведений о customer (кто-то живет, но неизвестно, кто). В такой ситуации можно построить отдельные классы для разных нулевых случаев. Иногда нулевые объекты могут содержать фактические данные, например регистрировать пользование услугами неизвестным жильцом, чтобы впоследствии, когда будет выяснено, кто является жильцом, выставить ему счет.

В сущности, здесь должен применяться более крупный паттерн, называемый «особым случаем» (special case). Класс особого случая - это отдельный экземпляр класса с особым поведением. Таким образом, неизвестный клиент UnknownCustomer и отсутствующий клиент NoCustomer будут особыми случаями Customer. Особые случаи часто встречаются среди чисел. В Java у чисел с плавающей точкой есть особые случаи для положительной и отрицательной бесконечности и для «нечисла» (NaN). Польза особых случаев в том, что благодаря им сокращается объем кода, обрабатывающего ошибки. Операции над числами с плавающей точкой не генерируют исключительные ситуации. Выполнение любой операции, в которой участвует NaN, имеет результатом тоже NaN, подобно тому как методы доступа к нулевым объектам обычно возвращают также нулевые объекты.

Введение утверждения (Introduce Assertion)

Некоторая часть кода предполагает определенное состояние программы. Сделайте предположение явным с помощью оператора утверждения.

```
double getExpenseLimit() {
 // should have either expense limit or a primary project
```

```
    return (_expenseLimit != NULL_EXPENSE) ?
       _expenseLimit :
       _primaryProject.getMemberExpenseLimit();
   }
```

```
   double getExpenseLimit() {
    Assert.isTrue (_expenseLimit != NULL_EXPENSE || _primaryProject != null);
    return (_expenseLimit != NULL_EXPENSE) ?
       _expenseLimlt :
       _primaryProject.getMemberExpenseLimit();
   }
```

Мотивировка

Часто некоторые разделы кода могут работать, только если выполнены определенные условия. Простейшим примером служит вычисление квадратного корня, выполнимое, только если входное значение положительно. Для объекта это может быть предположение о том, что хотя бы одно поле в группе содержит значение.

Такие предположения часто не сформулированы и могут быть выявлены лишь при просмотре алгоритма. Иногда предположения относительно нормальных условий указаны в комментариях. Лучше всего, когда предположение формулируется явно в утверждении. Утверждение (assertion) представляет собой условный оператор, который всегда должен выполняться. Отказ утверждения свидетельствует об ошибке в программе. Поэтому отказ утверждения должен всегда приводить к появлению необрабатываемых исключительных ситуаций. Утверждения никогда не должны использоваться другими частями системы. В действительности, утверждения обычно удаляются из готового кода. Поэтому важно сигнализировать о том, что нечто представляет собой утверждение.

Утверждения способствуют организации общения и отладки. Их коммуникативная роль состоит в том, что они помогают читателю понять, какие допущения делает код. Во время отладки утверждения помогают перехватывать ошибки ближе к источнику их возникновения. По моим наблюдениям, помощь для отладки невелика, если код самотестирующийся, но я ценю коммуникативную роль утверждений.

Техника

Поскольку утверждения не должны оказывать влияния на работу системы, при их введении всегда сохраняется поведение.

Когда вы видите, что предполагается выполненным некоторое условие, добавьте утверждение, в котором это явно формулируется.

Используйте для организации поведения утверждения класс assert.

Не злоупотребляйте утверждениями. Не старайтесь с их помощью проверять 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 неудобно для сопровождения. Логика, охватывающая допущение, полезна, потому что заставляет заново обдумать часть кода. Если код работает без утверждения, то от последнего больше путаницы, чем пользы, а последующая модификация может быть затруднена.

Всегда проверяйте, будет ли работать код в случае отказа утверждения. Если да, то удалите утверждение.

Остерегайтесь дублирования кода в утверждениях. Дублирующийся код пахнет в них так же дурно, как во всех других местах. Не скупитесь применять «Выделение метода» (Extract Method) для устранения дублирования.

Пример

Вот простая история с предельными расходами. Служащим может назначаться индивидуальный предел расходов. Если они участвуют в основном проекте, то могут пользоваться предельными расходами этого основного проекта. У них необязательно должны быть предел расходов или основной проект, но одно из двух у них должен присутствовать обязательно. Данное допущение предполагается выполненным в коде, который учитывает пределы расходов:

```
   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 расходов - либо проекта, либо персональный. В коде такое утверждение следует сформулировать явно:

```
   double getExpenseLimit(){
    Assert.isTrue (_expenseLimit != NULL_EXPENSE || _primaryProject != null);
    return (_expenseLimit != NULL_EXPENSE) ?
       _expenseLimit :
       _primaryProject.getMemberExpenseLimit();
   }
```

Данное утверждение не меняет каких-либо аспектов поведения программы. В любом случае, если утверждение не выполнено, возникает исключительная ситуация на этапе исполнения: либо исключительная ситуация нулевого указателя в withinLimit, либо исключительная ситуация времени выполнения в Assert.isTrue. В некоторых ситуациях утверждение помогает обнаружить ошибку, потому что оно ближе находится к тому месту, где она возникла. Однако в основном утверждение сообщает о том, как работает код и какие предположения в нем сделаны.

Я часто применяю «Выделение метода» (Extract Method) 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 выполнение готового кода. Конечно, наличие такого вспомогательного класса, как Assert, облегчает положение. К сожалению, что бы ни случилось, выполняются все выражения внутри параметров утверждения. Единственным способом не допустить этого является код следующего типа:

```
   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 ее значении. Однако добавление условного предложения запутывает код, и многие программисты предпочитают использовать просто Assert, а затем, при создании готового кода, отфильтровывать все строки, в которых есть assert (например, с помощью Perl).

В классе Assert должны быть разные методы с удобными именами. Помимо isTrue в нем может присутствовать equals и shouldNeverReachHere (сюда-вы-никогда-не-должны-попасть).

10 УПРОЩЕНИЕ ВЫЗОВОВ МЕТОДОВ

Интерфейсы составляют суть объектов. Создание простых в понимании и применении интерфейсов - главное искусство в разработке хорошего объектно-ориентированного программного обеспечения. В данной главе изучаются рефакторинги, в результате которых интерфейсы становятся проще.

Часто самое простое и важное, что можно сделать, это дать методу другое имя. Присваивание имен служит ключевым элементом коммуникации. Если вам понятно, как работает программа, не надо бояться применить «Переименование метода» (Rename Method), чтобы передать это понимание. Можно также (и нужно) переименовывать переменные и классы. В целом такие переименования обеспечиваются простой текстовой заменой, поэтому я не добавлял для них особых методов рефакторинга.

Сами параметры играют существенную роль в интерфейсах. Распространенными рефакторингами являются «Добавление параметра» (Add Parameter) и «Удаление параметра» (Remove Parameter). Программисты, не имеющие опыта работы с объектами, часто используют длинные списки параметров, обычные в других средах разработки. Объекты позволяют обойтись короткими списками параметров, а проведение ряда рефакторингов позволяет сделать их еще короче. Если передается несколько значений из объекта, примените «Сохранение всего объекта» (Preserve Whole Object), чтобы свести все значения к одному объекту. Если этот объект не существует, можно создать его с помощью «Введения граничного объекта» (Introduce Parameter Object). Если данные могут быть получены от объекта, к которому у метода уже есть доступ, можно исключить параметры с помощью «Замены параметра вызовом метода» (Replace Parameter with Method). Если параметры служат для определения условного поведения, можно прибегнуть к «Замене параметра явными методами» (Replace Parameter with Explicit Methods). Несколько аналогичных методов можно объединить, добавив параметр с помощью «Параметризации метода» (Parameterize Method).

Даг Ли (Doug Lea) предостерег меня относительно проведения рефакторингов, сокращающих списки параметров. Как правило, они выполняются так, чтобы передаваемые параметры были неизменяемыми, как часто бывает со встроенными объектами или объектами значений. Обычно длинные списки параметров можно заменить неизменяемыми объектами, в противном случае рефакторинги из этой группы следует выполнять с особой осторожностью.

Очень удобное соглашение, которого я придерживаюсь на протяжении многих лет, это четкое разделение методов, изменяющих состояние (модификаторов), и методов, опрашивающих состояние (запросов). Несчетное число раз я сталкивался с неприятностями сам или видел, как попадают в беду другие в результате смешения этих функций. Поэтому, когда я вижу такое смешение, то использую «Разделение запроса и модификатора» (Separate Query from Modifier), чтобы избавиться от него.

Хорошие интерфейсы показывают только то, что должны, и ничего лишнего. Скрыв некоторые вещи, можно улучшить интерфейс. Конечно, скрыты должны быть все данные (я думаю, нет надобности напоминать об этом), но также и все методы, которые можно скрыть. При проведении рефакторинга часто требуется что-то на время открыть, а затем спрятать с помощью «Сокрытия метода» (Hide Method) или «Удаления метода установки» (Remove Setting Method).

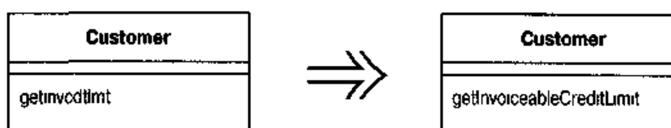
Конструкторы представляют собой особенное неудобство в Java и C++, поскольку требуют знания класса объекта, который надо создать. Часто знать его не обязательно. Необходимость в знании класса можно устранить, применив «Замену конструктора фабричным методом» (Replace Constructor with Factory Method).

Жизнь программирующих на Java отравляет также преобразование типов. Старайтесь по возможности избавлять пользователей классов от необходимости выполнять нисходящее преобразование, ограничивая его в каком-то месте с помощью «Инкапсуляции нисходящего преобразования типа» (Encapsulate Downcast).

В Java, как и во многих современных языках программирования, есть механизм обработки исключительных ситуаций, облегчающий обработку ошибок. Не привыкшие к нему программисты часто употребляют коды ошибок для извещения о возникших неприятностях. Чтобы воспользоваться этим новым механизмом обработки исключительных ситуаций, можно применить «Замену кода ошибки исключительной ситуацией» (Replace Error Code with Exception). Однако иногда исключительные ситуации не служат правильным решением, и следует выполнить «Замену исключительной ситуации проверкой» (Replace Exception with Test).

Переименование метода (Rename Method)

Имя метода не раскрывает его назначения. Измените имя метода.



Мотивировка

Важной частью пропагандируемого мной стиля программирования является разложение сложных процедур на небольшие методы. Если делать это неправильно, то придется изрядно помучиться, выясняя, что же делают эти маленькие методы. Избежать таких мучений помогает назначение методам хороших имен. Методам следует давать имена, раскрывающие их назначение. Хороший способ для этого - представить себе, каким должен быть комментарий к методу, и преобразовать этот комментарий в имя метода.

Жизнь такова, что удачное имя может не сразу придти в голову. В подобной ситуации может возникнуть соблазн бросить это занятие - в конце концов, не в имени счастье. Это вас соблазняет бес, не слушайте его. Если вы видите, что у метода плохое имя, обязательно измените его. Помните, что ваш код в первую очередь предназначен человеку, и только потом - компьютеру. Человеку нужны хорошие имена. Вспомните, сколько времени вы потратили, пытаясь что-то сделать, и насколько проще было бы, окажись у пары методов более удачные имена. Создание хороших имен - это мастерство, требующее практики; совершенствование этого мастерства - ключ к превращению в действительно искусного программиста. То же справедливо и в отношении других элементов сигнатуры метода. Если переупорядочение параметров проясняет суть, выполните его (см. «Добавление параметра» (Add Parameter) и «Удаление параметра» (Remove Parameter)).

Техника

Выясните, где реализуется сигнатура метода - в родительском классе или подклассе. Выполните эти шаги для каждой из реализаций.

Объявите новый метод с новым именем. Скопируйте тело прежнего метода в метод с новым именем и осуществите необходимую подгонку.

Выполните компиляцию.

Измените тело прежнего метода так, чтобы в нем вызывался новый метод.

Если ссылок на метод немного, вполне можно пропустить этот шаг.

Выполните компиляцию и тестирование.

Найдите все ссылки на прежний метод и замените их ссылками на новый. Выполняйте компиляцию и тестирование после каждой замены.

Удалите старый метод.

Если старый метод является частью интерфейса и его нельзя удалить, сохраните его и пометьте как устаревший (deprecated).

Выполните компиляцию и тестирование.

Пример

Имеется метод для получения номера телефона лица:

```
public String getTelephoneNumber() {
 return ( "(" + _officeAreaCode + ")"  + _officeNumber);
}
```

Я хочу переименовать метод в getOfficeTelephoneNumber. Начинаю с создания нового метода и копирования тела в новый метод. Старый метод изменяется так, чтобы вызывать новый:

```
class Person {
 public String getTelephoneNumber(){
 return getOfficeTelephoneNumber();
}
public String getOfficeTelephoneNumberO {
 return ( "(" + _officeAreaCode + ")" + _officeNumber);
}
```

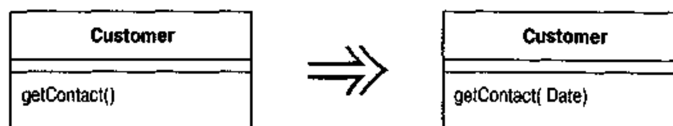
Теперь я нахожу места вызова прежнего метода и изменяю их так, чте-бы в них вызывался новый метод. После всех изменений прежний метод можно удалить.

Такая же процедура осуществляется, когда нужно добавить или удалить параметр.

Если мест, из которых вызывается метод, немного, я изменяю их для обращения к новому методу без использования прежнего метода в качестве делегирующего. Если в результате тесты «захромают», я вернусь назад и произведу изменения без лишней поспешности.

Добавление параметра (Add Parameter)

Метод нуждается в дополнительной информации от вызывающего. Добавьте параметр, который может передать эту информацию.



Мотивировка

«Добавление параметра» (Add Parameter) является очень распространенным рефакторингом, и вы наверняка уже производили его. Его мотивировка проста. Необходимо изменить метод, и изменение требует информации, которая ранее не передавалась, поэтому вы добавляете параметр.

В действительности, я хочу поговорить о том, когда не следует проводить данный рефакторинг. Часто есть альтернативы добавлению параметра, которые предпочтительнее, поскольку не приводят к увеличению списка параметров. Длинные списки параметров дурно пахнут, потому что их трудно запоминать и они часто содержат группы данных.

Взгляните на уже имеющиеся параметры. Можете ли вы запросить у одного из этих объектов необходимую информацию? Если нет, не будет ли разумно создать в них метод, предоставляющий эту информацию? Для чего используется эта информация? Не лучше ли было бы иметь это поведение в другом объекте - том, у которого есть эта информация? Посмотрите на имеющиеся параметры и представьте их себе вместе с новым параметром. Не лучше ли будет провести «Введение граничного объекта» (Introduce Parameter Object)?

Я не утверждаю, что параметры не надо добавлять ни при каких обстоятельствах; я часто делаю это, но не следует забывать об альтернативах.

Техника

Механика «Добавления параметра» (Add Parameter) очень похожа на «Переименование метода» (Rename Method).

Выясните, где реализуется сигнатура метода - в родительском классе или 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 подгонку.

Если требуется добавить несколько параметров, проще сделать это сразу.

Выполните компиляцию.

Измените тело прежнего метода так, чтобы в нем вызывался новый метод.

Если ссылок на метод немного, вполне можно пропустить этот шаг.

В качестве значения параметра можно передать любое значение, но обычно используется null для параметра-объекта и явно необычное значение для встроенных типов. Часто полезно использовать числа, отличные от нуля, чтобы быстрее обнаружить этот случай.

Выполните компиляцию и тестирование.

Найдите все ссылки на прежний метод и замените их ссылками на новый. Выполняйте компиляцию и тестирование после каждой замены.

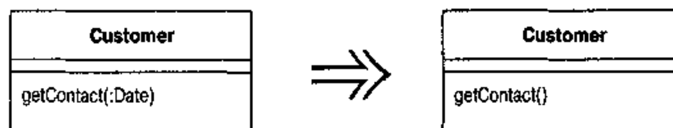
Удалите старый метод.

Если старый метод является частью интерфейса и его нельзя удалить, сохраните его и пометьте как устаревший.

Выполните компиляцию и тестирование.

Удаление параметра (Remove Parameter)

Параметр более не используется в теле метода. Удалите его.



Мотивировка

Программисты часто добавляют параметры и неохотно их удаляют. В конце концов, ложный параметр не вызывает никаких проблем, а в будущем может снова понадобиться.

Это нашептывает демон темноты и неясности; изгоните его из своей души! Параметр указывает на необходимую информацию; различие значений играет роль. Тот, кто вызывает ваш метод, должен озаботиться передачей правильных значений. Не удалив параметр, вы создаете лишнюю работу для всех, кто использует метод. Это нехороший компромисс, тем более что удаление параметров представляет собою простой рефакторинг.

Осторожность здесь нужно проявлять, когда метод является полиморфным. В этом случае может оказаться, что данный параметр используется в других реализациях этого метода, и тогда его не следует удалять. Можно добавить отдельный метод для использования в таких случаях, но нужно исследовать, как пользуются этим методом вызывающие, чтобы решить, стоит ли это делать. Если вызывающим известно, что они имеют дело с некоторым подклассом и выполняют дополнительные действия для поиска параметра либо пользуются информацией об иерархии классов, чтобы узнать, можно ли обойтись значением null, добавьте еще один метод без параметра. Если им не требуется знать о том, какой метод какому классу принадлежит, следует оставить вызывающих в счастливом неведении.

Техника

Техника «Удаления параметра» (Remove Parameter) очень похожа на «Переименование метода» (Rename Method) и «Добавление параметра» (Add Parameter).

Выясните, реализуется ли сигнатура метода в родительском классе или подклассе. Выясните, используют ли класс или родительский класс этот параметр. Если да, не производите этот рефакторинг.

Объявите новый метод без параметра. Скопируйте тело прежнего метода в метод с новым именем и осуществите необходимую подгонку.

Если требуется удалить несколько параметров, проще удалить их все сразу.

Выполните компиляцию.

Измените тело старого метода так, чтобы в нем вызывался новый метод.

Если ссылок на метод немного, вполне можно пропустить этот шаг.

Выполните компиляцию и тестирование.

Найдите все ссылки на прежний метод и замените их ссылками на новый. Выполняйте компиляцию и тестирование после каждой замены.

Удалите старый метод.

Если старый метод является частью интерфейса и его нельзя удалить, сохраните его и пометьте как устаревший (deprecated).

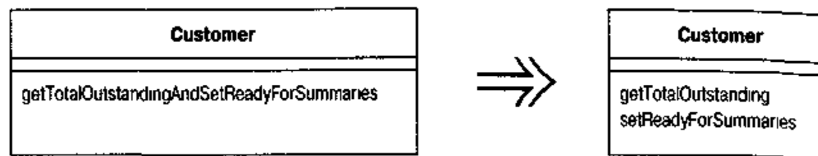
Выполните компиляцию и тестирование.

Поскольку я чувствую себя вполне уверенно при добавлении и удалении параметров, я часто делаю это сразу для группы за один шаг.

Разделение запроса и модификатора (Separate Query from Modifier)

Есть метод, возвращающий значение, но, кроме того, изменяющий состояние объекта.

Создайте два метода - один для запроса и один для модификации.



Мотивировка

Если есть функция, которая возвращает значение и не имеет видимых побочных эффектов, это весьма ценно. Такую функцию можно вызывать сколь угодно часто. Ее вызов можно переместить в методе в другое место. Короче, с ней значительно меньше проблем.

Хорошая идея - четко проводить различие между методами с побочными эффектами и теми, у которых их нет. Полезно следовать правилу, что у любого метода, возвращающего значение, не должно быть наблюдаемых побочных эффектов. Некоторые программисты рассматривают это правило как абсолютное [Meyer]. Я не придерживаюсь его в 100% случаев (как и любых других правил), но в основном стараюсь его соблюдать, и оно служит мне верой и правдой.

Обнаружив метод, который возвращает значение, но также обладает побочными эффектами, следует попытаться разделить запрос и модификатор.

Возможно, вы обратили внимание на слова «наблюдаемые побочные эффекты». Стандартная оптимизация заключается в кэшировании значения запроса в поле, чтобы повторные запросы выполнялись быстрее. Хотя при этом изменяется состояние объекта с кэшем, такое изменение не наблюдаемо. Любая последовательность запросов всегда возвращает одни и те же результаты для каждого запроса [Meyer].

Техника

Создайте запрос, возвращающий то же значение, что и исходный метод.

Посмотрите, что возвращает исходный метод. Если возвращается значение временной переменной, найдите место, где ей присваивается значение.

Модифицируйте исходный метод так, чтобы он возвращал результат обращения к запросу.

Все return в исходном методе должны иметь вид return newOuery().

Если временная переменная использовалась в методе с единственной целью захватить возвращаемое значение, ее, скорее всего, можно удалить.

Выполните компиляцию и тестирование.

Для каждого вызова замените одно 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

Вот функция, которая сообщает мне для системы безопасности имя злодея и посылает предупреждение. Согласно имеющемуся правилу посылается только одно предупреждение, 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 создает побочных эффектов:

```
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 провожу 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 методы, из которых происходит обращение, 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 пользуется для своей работы телом запроса. Я могу теперь применить к модификатору «Замещение алгоритма» (Substitute Algorithm) и воспользоваться этим:

```
void sendAlert(String[] people){
 if (!foundPerson(people).equals(“”)) sendAlert();
}
```

Проблемы параллельного выполнения

Те, кто работает в многопоточной системе, знают, что выполнение операций проверки и установки как единого действия является важной идиомой. Влечет ли это конфликт с «Разделением запроса и модификатора» (Separate Query from Modifier)? Я обсуждал эту проблему с Дагом Ли и пришел к выводу, что нет, но необходимо выполнить некоторые дополнительные действия. По-прежнему полезно разделить операции запроса и модификации, однако надо сохранить третий метод, осуществляющий и то и другое. Операция «запрос-модификация» должна вызывать отдельные методы запроса и модификации и синхронизироваться. Если операции запроса и модификации не синхронизированы, можно также ограничить их видимость пакетом или сделать закрытыми. Благодаря этому получается безопасная синхронизированная операция, разложенная на два более легко понимаемые 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.

Мотивировка

Иногда встречаются два метода, выполняющие сходные действия, но отличающиеся несколькими значениями. В этом случае можно упростить положение, заменив разные методы одним, который обрабатывает разные ситуации с помощью параметров. При таком изменении устраняется дублирование кода и возрастает гибкость, потому что в результате добавления параметров можно обрабатывать и другие ситуации.

Техника

Создайте параметризованный метод, которым можно заменить каждый повторяющийся метод.

Выполните компиляцию.

Замените старый метод вызовом нового.

Выполните компиляцию и тестирование.

Повторите для каждого метода, выполняя тестирование после каждой замены.

Иногда это оказывается возможным не для всего метода, а только для его части. В таком случае сначала выделите фрагмент в метод, а затем параметризуйте этот метод.

Пример

Простейший случай представляют методы, показанные ниже:

```
class Employee {
 void tenPercentRaise () { salary * = 1.1;}
 void fivePercentRaise () { salary *= 1.05;}
}
```

Их можно заменить следующим:

```
void raise (double factor) {
 salary *= (1 + factor);
}
```

Конечно, этот случай настолько прост, что его заметит каждый. 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 таким:

```
 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, использующий несколько значений, которые можно передать в качестве параметров.

Замена параметра явными методами (Replace Parameter with Explicit Methods)

Есть метод, выполняющий разный код в зависимости от значения и

раметра перечислимого типа.

Создайте отдельный метод для каждого .значения параметра.

```
 void setValue (String name, int value) {
  if (name.equals("height")) {
    _heignt = value;
    Return;
  }
  if {name.equals("width")) {
    _width = value;
    return;
  }
  Assert.shouldNeverReachHere();
 }
```

```
 void setHeight(int arg) {
  _height = arg;
 }
 void setWidth (int arg) {
  _width = arg;
 }
```

Мотивировка

«Замена параметра явными методами» (Replace Parameter with Explicit Methods) является рефакторингом, обратным по отношению к «Параметризации метода» (Parameterize Method). Типичная ситуация для ее применения возникает, когда есть параметр с дискретными значениями, которые проверяются в условном операторе, и в висимости от результатов проверки выполняется разный код. Вызывающий должен решить, что ему надо сделать, и установить для параметра соответствующее значение, поэтому можно создать различные методы и избавиться от условного оператора. При этом удается избежать условного поведения и получить контроль на этапе компиляции. Кроме того, интерфейс становится более прозрачным. Если используется параметр, то программисту, применяющему метод, приходится не только рассматривать имеющиеся в классе методы, но и определять для параметра правильное значение. Последнее часто плохо документировано.

Прозрачность ясного интерфейса может быть достаточным результатом, даже если проверка на этапе компиляции не приносит пользы.

Код Switch beOn() значительно яснее, чем Switch setState(true), даже если все действие заключается в установке внутреннего логического поля.

Не стоит применять «Замену параметра явными методами» (Replace Parameter with Explicit Methods), если значения параметра могут изменяться в значительной мере. В такой ситуации, когда переданный параметр просто присваивается полю, применяйте простой метод установки значения. Если требуется условное поведение, лучше применить «Замену условного оператора полиморфизмом» (Replace Conditional with Polymorphism).

Техника

Создайте явный метод для каждого значения параметра.

Для каждой ветви условного оператора вызовите соответствующий новый метод.

Выполняйте компиляцию и тестирование после изменения каждой ветви.

Замените каждый вызов условного метода обращением к соответствующему новому методу.

Выполните компиляцию и тестирование.

После изменения всех вызовов удалите условный метод.

Пример

Я предпочитаю создавать подкласс класса служащего, исходя из переданного параметра, часто в результате «Замены конструктора фабричным методом» (Replace Constructor with Factory Method):

```
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(“Incorrect type code value”);
 }
}
```

Поскольку это фабричный метод, я не могу воспользоваться «Заменой условного оператора полиморфизмом» (Replace Conditional with Polymorphism), т. к. я еще не создал объект. Я предполагаю, что новых классов будет немного, поэтому есть смысл в явных интерфейсах. Сначала создаю новые методы:

```
static Employee createEngineer() {
 return new Engineer();
```

```
 }
 static Employee createSalesman() {
  return new Salesman();
 }
 static Employee createManager() {
  return new Manager();
 }
```

Поочередно заменяю ветви операторов 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("Incorrect type code value");
  }
 }
```

Выполняю компиляцию и тестирование после изменения каждой ветви, 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("Incorrect type code value");
  }
 }
```

Теперь я перехожу к местам вызова старого метода create. Код вида

```
 Employee kent = Employee.create(ENGINEER);
```

я заменяю таким:

```
 Employee kent = Employee.createEngineer();
```

Проделав это со всеми местами вызова create, я могу удалить метод create. Возможно, удастся также избавиться от констант.

Сохранение всего объекта (Preserve Whole Object)

Вы получаете от объекта несколько значений, которые затем передавайте как параметры при вызове метода. Передавайте вместо этого весь объект.

```
 int law = daysTempRange().getLow();
 int high = daysTempRange().getHigh();
```

```
withinPlan = plan.withinRange(low, high);
```

```
withinPlan = plan.withinRange(daysTempRange());
```

Мотивировка

Такого рода ситуация возникает, когда объект передает несколько значений данных из одного объекта как параметры в вызове метода. Проблема заключается в том, что если вызываемому объекту в дальнейшем потребуются новые данные, придется найти и изменить все вызовы этого метода. Этого можно избежать, если передавать весь объект, от которого поступают данные. В этом случае вызываемый объект может запрашивать любые необходимые ему данные от объекта в целом. Помимо того что список параметров становится более устойчив к изменениям, «Сохранение всего объекта» (Preserve Whole Object) часто улучшает читаемость кода. С длинными списками параметров бывает трудно работать, потому что как вызывающий, так и вызываемый объект должны помнить, какие в нем были значения. Они также способствуют дублированию кода, потому что вызываемый объект не может воспользоваться никакими другими методами всего объекта для вычисления промежуточных значений.

Существует, однако, и обратная сторона. При передаче значений вызванный объект зависит от значений, но не зависит от объекта, из которого извлекаются эти значения. Передача необходимого объекта устанавливает зависимость между ним и вызываемым объектом. Если это запутывает имеющуюся структуру зависимостей, не применяйте «Сохранение всего объекта» (Preserve Whole Object). Другая причина, по которой, как говорят, не стоит применять «Сохранение всего объекта» (Preserve Whole Object), заключается в том, что если вызывающему объекту нужно только одно значение необходимого объекта, лучше передавать это значение, а не весь объект. Я не согласен с такой точкой зрения. Передачи одного значения и одного объекта равнозначны, по крайней мере, в отношении ясности (передача параметров по значению может потребовать дополнительных накладных расходов). Движущей силой здесь является проблема зависимости.

То, что вызываемому методу нужно много значений от другого объекта, указывает на то, что в действительности этот метод должен быть определен в том объекте, который дает ему значения. Применяя «Сохранение всего объекта» (Preserve Whole Object), рассмотрите в качестве возможной альтернативы «Перемещение метода» (Move Method).

Может оказаться, что целый объект пока не определен. Тогда необходимо «Введение граничного объекта» (Introduce Parameter Object).

Часто бывает, что вызывающий объект передает в качестве параметров несколько значений своих собственных данных. В таком случае можно вместо этих значений передать в вызове this, если имеются соответствующие 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 всего 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, в том случае, когда 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 температуру 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 данном простом случае это можно сделать за один шаг. Если бы участвовало больше параметров, можно было бы 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 применяю метод ко всему объекту, а не к одному 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);
   }
  }
```

Продолжаю, 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 wlthinPlan(HeatingPlan plan) {
    int low = daysTempRange().getLow();
    int high = daysTempRange().getHigh();
    return plan.withinRange(daysTempRange());
 }
}
```

Такое применение целых объектов вскоре позволяет понять, что полезно переместить поведение в целый объект, т. к. это облегчит работу с ним.

```
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);
```

Мотивировка

Если метод может получить передаваемое в качестве параметра значение другим способом, он так и должен поступить. Длинные списки па раметров трудны для понимания, и их следует по возможности сокращать.

Один из способов сократить список параметров заключается в том чтобы посмотреть, не может ли рассматриваемый метод получить не обходимые параметры другим путем. Если объект вызывает свой метод и вычисление для параметра не обращается к каким-либо парамет рам вызывающего метода, то должна быть

возможность удалить параметр путем превращения вычисления в собственный метод. Это верно и тогда, 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 может быть различным в каждом вызове (если, конечно, не заменить его методом). Нельзя также удалять параметр, если у получателя нет ссылки на отправителя и вы не хотите предоставить ему ее.

Иногда параметр присутствует в расчете на параметризацию метода в будущем. В этом случае я все равно избавился бы от него. Займитесь параметризацией тогда, когда это вам потребуется; может оказаться, что необходимый параметр вообще не найдется. Исключение из этого правила я бы сделал только тогда, когда результирующие изменения в интерфейсе могут иметь тяжелые последствия для всей программы, например потребуют длительной компиляции или изменения большого объема существующего кода. Если это тревожит вас, оцените, насколько больших усилий потребует такое изменение. Следует также разобраться, нельзя ли сократить зависимости, из-за которых это изменение столь затруднительно. Устойчивые интерфейсы - это благо,) но не надо консервировать плохие интерфейсы.

Техника

При необходимости выделите расчет параметра в метод.

Замените ссылки на параметр в телах методов ссылками на метод.

Выполняйте компиляцию и тестирование после каждой замены.

Примените к параметру «Удаление параметра» (Remove Parameter).

Пример

Еще один маловероятный вариант скидки для заказов выглядит так:

```
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 выделения расчета категории 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 ссылки на параметр в discountedPrice:

```
private double discountedPrice (int basePrice, int discountLevel) {
 if (getDiscountLevel() == 2) return basePrice * 0.1;
 else return basePrice * 0.05;
```

```
}
```

После этого можно применить «Удаление параметра» (Remove Parameter):

```
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 можно избавиться от временной переменной:

```
public double getPrice() {
 int basePrice = _quantity * _itemPrice;
 double finalPrice = discountedPrice (basePrice);
 return finalPrice;
}
```

После этого настает момент избавиться от другого параметра и его временной переменной. Остается следующее:

```
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;
}
```

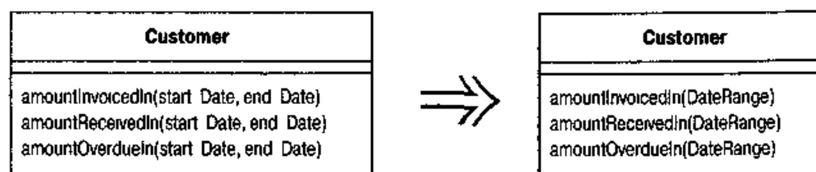
Поэтому можно также применить к discountedPrice «Встраивание метода» (Inline Method)

```
private double getPrice () {
 if (getDiscountLevel() == 2) return getBasePrice() * 0.1;
 else return getBasePrice() * 0.05;
}
```

Введение граничного объекта (Introduce Parameter Object)

Есть группа параметров, естественным образом связанных друг с другом.

Замените их объектом.



Мотивировка

Часто встречается некоторая группа параметров, обычно передаваемых вместе. Эта группа может использоваться несколькими методами одного или более классов. Такая группа классов представляет собой группу данных (data clump) и может быть заменена объектом, хранящим все эти данные. Целесообразно свести эти параметры в объект, чтобы сгруппировать данные вместе. Такой рефакторинг полезен, поскольку сокращает размер списков параметров, а в длинных списках параметров трудно разобраться. Определяемые в новом объекте методы доступа делают также код более последовательным, благодаря чему его проще понимать и модифицировать.

Однако получаемая выгода еще существеннее, поскольку после группировки параметров обнаруживается поведение, которое можно переместить в новый класс. Часто в телах методов производятся одинаковые действия со значениями параметров. Перемещая это поведение в новый объект, можно избавиться от значительного объема дублирующегося кода.

Техника

Создайте новый класс для представления группы заменяемых параметров. Сделайте этот класс неизменяемым.

Выполните компиляцию.

Для этой новой группы данных примените рефакторинг «Добавление параметра» (Add Parameter). Во всех вызовах метода используйте в качестве значения параметра null.

Если точек вызова много, можно сохранить старую сигнатуру и вызывать в ней новый метод. Примените рефакторинг сначала к старому методу. После этого можно изменять вызовы один за другим и в конце убрать старый метод.

Для каждого параметра в группе данных осуществите его удаление из сигнатуры. Модифицируйте точки вызова и тело метода, чтобы они использовали вместо этого значения нового объекта.

Выполняйте компиляцию и тестирование после удаления каждого параметра.

Убрав параметры, поищите поведение, которое можно было бы переместить в новый объект с помощью «Перемещения метода» (Move Method).

Это может быть целым методом или его частью. Если поведение составляет часть метода, примените к нему сначала «Выделение метода» (Extract Method), а затем переместите 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, хранящем коллекцию проводок и предоставляющем 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();
  client code
   double flow = anAccount.getFlowBetween(startDate, endDate);
```

Не знаю, в который уже раз я сталкиваюсь с парами значений, представляющих диапазон, например, даты начала и конца или верхние и нижние числовые границы. Можно понять, почему это происходит, в конце концов, раньше я сам делал подобные вещи. Но с тех пор как я увидел паттерн диапазона [Fowler, AP], я всегда стараюсь использовать не пары, а диапазоны. Первым делом объявляю простой объект 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 нет методов для модификации значений. Это мудрый шаг для того, чтобы избежать наложения ошибок. Поскольку в Java параметры передаются по значению, придание классу свойства неизменяемости имитирует способ действия параметров Java, поэтому такое допущение правильно 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;
  }
```

```
client code
 double flow = anAccount.getFlowBetween(startDate, endDate, null);
```

В этом месте мне надо только выполнить компиляцию, потому что я еще не менял никакого поведения.

Следующим шагом удаляется один из параметров, место которого занимает новый объект. Для этого я удаляю параметр start и модифицирую метод и обращения к нему 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;
}
client code
 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;
}
client code
 double flow = anAccount.getFlowBetween(new DateRange (startDate, endDate));
```

Я ввел граничный объект; однако я могу получить от этого рефакторинга больше пользы, если перемещу в новый объект поведение из других методов. В данном случае я могу взять код в условии, применить к нему «Выделение метода» (Extract Method) и «Перемещение метода» (Move Method) и получить:

```
class Account {
 double getFlowBetween (DateRange range) {
    double result = 0;
    Enumeration e = _entries.elements();
    while (e.hasMoreElements()) {
```

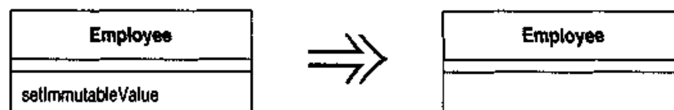
```
        Entry each = (Entry) e.nextElement();
        if (range.includes(each.getDate())) {
           result += each.getValue();
      }
    }
   return result;
  }
  class DateRange {
   boolean includes (Date arg) {
      return (arg.equals(_start) || arg.equals(_end) ||
              (arg.after(_start) && arg.before(_end)));
   }
  }
```

Обычно такие простые выделения и перемещения, как здесь, я выполняю за один шаг. Если возникнет ошибка, можно дать задний ход, после чего модифицировать код за два маленьких шага.

Удаление метода установки значения (Remove Setting Method)

Поле должно быть установлено в момент создания и больше никогда не изменяться.

Удалите методы, устанавливающие значение этого поля.



Мотивировка

Предоставление метода установки значения показывает, что поле может изменяться. Если вы не хотите, чтобы поле менялось после создания объекта, то не предоставляйте метод установки (и объявите поле с ключевым словом final). Благодаря этому ваш замысел становится ясен и часто устраняется всякая возможность изменения поля.

Такая ситуация часто имеет место, когда программисты слепо пользуются косвенным доступом к переменным [Beck]. Такие программисты применяют затем методы установки даже в конструкторе. Думаю, что это может объясняться стремлением к последовательности, но путаница, которую метод установки вызовет впоследствии, не идет с этим ни в какое сравнение.

Техника

Если поле не объявлено как final, сделайте это.

Выполните компиляцию и тестирование.

Проверьте, чтобы метод установки вызывался только в конструкторе или методе, вызываемом конструктором.

Модифицируйте конструктор или вызываемый им метод, чтобы он непосредственно обращался к переменным.

Это не удастся сделать, если есть подкласс, устанавливающий закрытые поля родительского класса. В таком случае надо попытаться предоставить защищенный метод родительского класса (в идеале - конструктор), устанавливающий эти значения. Как бы вы ни поступили, не давайте методу родительского класса имя, по которому его можно было бы принять за метод 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; }
}
```

Возникают проблемы нескольких видов. Первый случай - выполнение вычислений над аргументом:

```
class Account {
 private String _id;
 Account (String id) { setId(id); }
 void setId (String arg) { _id = "ID" + arg; }
}
```

Если изменение простое (как здесь) и конструктор только один, можно внести изменения в конструктор. Если изменение сложное или необходимо выполнять вызов из различных методов, я должен создать метод. В этом случае надо присвоить методу имя, делающее ясным его назначение:

```
class Account {
 private final String _id;
 Account (String id) { initializeId(id); }
 void initializeId (String arg) { _id = "ID" + arg; }
}
```

Неприятный случай возникает, когда есть подклассы, инициализирующие закрытые переменные родительского класса:

```
class InterestAccount extends Account {
 private double _interestRate;
 InterestAccount (String id, double rate) {
    setId(id);
    _interestRate = rate;
 }
}
```

Проблема в том, что нельзя непосредственно обратиться к id, чтобы присвоить ему значение. Самым лучшим; решением будет использовать конструктор родительского класса:

```
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;
 }
}
```

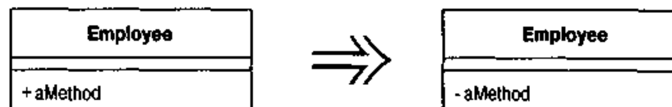
Следует также рассмотреть случай присвоения значения коллекции:

```
class Person {
 Vector getCourses() { return _courses; }
 void setCourses(Vector arg) { _courses = arg; }
 private Vector _courses;
}
```

Здесь я хочу заменить метод установки операциями добавления и удаления. Это описывается в «Инкапсуляции коллекции» (Encapsulate Collection).

Сокрытие метода (Hide Method)

Метод не используется никаким другим классом. Сделайте метод закрытым.



Мотивировка

Рефакторинг часто заставляет менять решение относительно видимости методов. Обнаружить случаи, когда следует расширить видимость метода, легко: метод нужен другому классу, поэтому ограничения видимости ослабляются. Несколько сложнее определить, не является ли метод излишне видимым. В идеале некоторое средство должно проверять все методы и определять, нельзя ли их скрыть. Если такого средства нет, вы сами должны регулярно проводить такую проверку.

Очень часто потребность в сокрытии методов получения и установки значений возникает в связи с разработкой более богатого интерфейса, предоставляющего дополнительное поведение, особенно если вы начинали с класса, мало что добавлявшего к простой инкапсуляции данных. По мере встраивания в класс нового поведения может обнаружиться, что в открытых методах получения и установки более нет надобности, и тогда можно их скрыть. Если сделать методы получения или установки закрытыми и использовать прямой доступ к переменным, можно удалить метод.

Техника

Регулярно проверяйте, не появилась ли возможность сделать метод более закрытым.

Используйте средства контроля типа lint, делайте проверки вручную периодически и после удаления обращения к методу из другого класса.

В особенности ищите такие случаи для методов установки.

Делайте каждый метод как можно более закрытым.

Выполняйте компиляцию после проведения нескольких сокрытий методов.

Компилятор проводит естественную проверку, поэтому нет необходимости в компиляции после каждого изменения. Если что-то не так, ошибка легко обнаруживается.

Замена конструктора фабричным методом (Replace Constructor with Factory Method)

Вы хотите при создании объекта делать нечто большее. Замените конструктор фабричным методом.

```
Employee (int type) {
 _type = type;
}
```

```
static Employee create (int type){
 return new Employee(type);
}
```

Мотивировка

Самая очевидная мотивировка «Замены конструктора фабричным методом» (Replace Constructor with Factory Method) связана с заменой кода типа созданием подклассов. Имеется объект, который обычно создается с кодом типа, но теперь требует подклассов. Конкретный подкласс определяется кодом типа. Однако конструкторы могут возвращать только экземпляр запрашиваемого объекта. Поэтому надо заменить конструктор фабричным методом [Gang of Four]. Фабричные методы можно использовать и в других ситуациях, когда возможностей конструкторов оказывается недостаточно. Они важны при «Замене значения ссылкой» (Change Value to Reference). Их можно также применять для задания различных режимов создания, выходящих за рамки числа и типов параметров.

Техника

Создайте фабричный метод. Пусть в его теле вызывается текущий конструктор.

Замените все обращения к конструктору вызовами фабричного метода.

Выполняйте компиляцию и тестирование после каждой замены.

Объявите конструктор закрытым.

Выполните компиляцию.

Пример

Скорый, но скучный и избитый пример дает система зарплаты служащих. Пусть класс служащего такой:

```
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 типа, поэтому мне нужен фабричный метод:

```
static Employee create(int type) {
 return new Employee(type);
}
```

После этого я изменяю все места вызова данного конструктора так, чтобы вызывался этот новый 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 выигрыш не велик; польза основана на том факте, что я отделил получателя вызова для создания объекта от класса создаваемого объекта. Если позднее я применю «Замену кода типа подклассами» (Replace Type Code with Subclasses), 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("Incorrect type code value");
    }
 }
}
```

Плохо в этом то, что есть оператор switch. Если придется добавлять новый подкласс, потребуется вспомнить об обновлении этого оператора switch, а я склонен к забывчивости.

Хороший способ справиться с этим - воспользоваться Class.forName. Первое, что надо сделать, это изменить тип параметра, что, в сущности, является разновидностью «Переименования метода» (Rename Method). Сначала я создаю новый метод, принимающий строку в качестве аргумента:

```
static Employee {
 create (String name) {
    try {
       return (Employee) Class.forName(name).newInstance();
    }
    catch (Exception e) {
       throw new IllegalArgumentException ("Unable to instantiate" + name);
    }
 }
}
```

После этого я преобразую принимающий целое create, 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("Incorrect type code value");
    }
 }
}
```

Затем можно работать с кодом, вызывающим create, заменяя такие предложения, как

```
Employee.create(ENGINEER);
```

на

```
Employee.create("Engineer");
```

По завершении можно удалить версию этого метода с целочисленным параметром.

Такой подход хорош тем, что устраняет необходимость в обновлении метода create при добавлении новых подклассов Employee. Однако в таком подходе недостает проверки на этапе компиляции: орфографическая ошибка приводит к ошибке этапа исполнения. Если это важно, я использую явный метод (см. ниже), но тогда я должен добавлять новый метод всякий раз, когда вводится новый подкласс. Это компромисс между гибкостью и безопасностью типа. К счастью, если принято неверное решение, можно изменить его, обратившись к «Параметризации метода» (Parameterize Method) или «Замене параметра явными методами» (Replace Parameter with Explicit Methods).

Другая причина остерегаться class.forName связана с тем, что он открывает клиентам имена подклассов. Это не очень страшно, поскольку можно использовать другие строки и осуществлять другое поведение с помощью фабричного метода. Это веское основание, чтобы не удалять фабрику с помощью «Встраивания метода» (Inline Method).

Пример: создание подклассов явными методами

Можно применить другой подход, чтобы скрыть подклассы с помощью явных методов. Это полезно, когда есть лишь несколько подклассов, которые не изменяются. Поэтому можно иметь абстрактный класс Person (лицо) с подклассами Male (мужчина) и Female (женщина). Начну с определения в родительском классе фабричного метода для каждого подкласса:

```
class Person   {
 static Person createMale(){
    return new Male();
 }
 static Person createFemale() {
    return new Female();
 }
}
```

После этого можно заменить вызовы вида

```
Person kent = new Male();
```

такими:

```
Person kent = Person.createMale();
```

В результате родительский класс обладает знаниями о подклассах. Если это нежелательно, нужна более сложная схема, например схема «product trader» [Blummer и Riehle]. Чаще всего, впрочем, такие сложности не нужны, и этот подход прекрасно работает.

Инкапсуляция нисходящего преобразования типа (Encapsulate Downcast)

Метод возвращает объект, к которому вызывающий должен применить нисходящее преобразование типа. Переместите нисходящее преобразование внутрь метода.

```
Object lastReading () {
 return readings.lastElement();
}
```

```
Reading lastReading () {
 return (Reading) readings.lastElement();
}
```

Мотивировка

Нисходящее преобразование типа - одна из самых неприятных вещей, которыми приходится заниматься в строго типизированных ОО-языках. Оно неприятно, потому что кажется ненужным: вы сообщаете компилятору

то, что он должен сообразить сам. Но поскольку сообразить бывает довольно сложно, приходится делать это самому. Это особенно распространено в Java, где отсутствие шаблонов означает, что преобразование типа нужно выполнять каждый раз, когда объект выбирается из коллекции.

Нисходящее преобразование типа может быть неизбежным злом, но применять его следует как можно реже. Тот, кто возвращает значение из метода и знает, что тип этого значения более специализирован, чем указанный в сигнатуре, возлагает лишнюю работу на своих клиентов. Вместо того чтобы заставлять их выполнять преобразование типа, всегда следует передавать им насколько возможно узко специализированный тип.

Часто такая ситуация возникает для методов, возвращающих итератор или коллекцию. Лучше посмотрите, для чего люди используют этот итератор, и создайте соответствующий метод.

Техника

Поищите случаи, когда приходится выполнять преобразование типа результата, возвращаемого методом.

Такие случаи часто возникают с методами, возвращающими коллекцию или итератор.

Переместите преобразование типа в метод.

Для методов, возвращающих коллекцию, примените «Инкапсуляцию коллекции» (Encapsulate Collection).

Пример

Есть метод с именем lastReading, возвращающий последнее данное (показание прибора) из вектора:

```
Object lastReading() {
 return readings.lastElement();
}
```

Следует заменить его таким:

```
Reading lastReading() {
 return (Reading) readings.lastElement();
}
```

Хорошо делать это там, где находятся классы коллекций. Пусть, например, эта коллекция из Reading находится в классе Site, и есть такой код:

```
Reading lastReading = (Reading) theSite.readings().lastElement();
```

Избежать преобразования типов и скрыть используемую коллекцию можно с помощью следующего кода:

```
Reading lastReading = theSite.lastReading();
class Site {
 Reading lastReading() {
    return (Reading) readings().lastElement();
 }
}
```

При изменении метода, в результате которого возвращается подкласс, меняется сигнатура, но сохраняется работоспособность имеющегося кода, потому что компилятор умеет использовать подкласс вместо родительского класса. Конечно, при этом должно быть гарантировано, что подкласс не делает ничего, нарушающего контракт родительского класса.

Замена кода ошибки исключительной ситуацией (Replace Error Code with Exception)

Метод возвращает особый код, индицирующий ошибку.

Возбудите вместо этого исключительную ситуацию.

```
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;
    }
```

Мотивировка

В компьютерах, как и в жизни, иногда происходят неприятности, на которые надо как-то реагировать. Проще всего остановить программу и вернуть код ошибки. Это компьютерный эквивалент самоубийства из-за опоздания на самолет. (Я бы не остался в живых, будь у меня хоть десять жизней.) Хотя я и пытаюсь шутить, но в решении компьютера о самоубийстве есть свои достоинства. Если потери от краха программы невелики, а пользователь терпелив, то можно и остановить программу. Однако в важных программах должны приниматься более радикальные меры.

Проблема в том, что та часть программы, которая обнаружила ошибку, не всегда является той частью, которая может решить, как с этой ошибкой справиться. Когда некоторая процедура обнаруживает ошибку, она должна сообщить о ней в вызвавший ее код, а тот может переслать ошибку вверх по цепочке. Во многих языках для отображения ошибки отводится специальное устройство вывода. В системах Unix и основанных на C традиционно возвращается код, указывающий на успешное или неудачное выполнение программы.

В Java есть лучший способ - исключительные ситуации. Их преимущество в том, что они четко отделяют нормальную обработку от обработки ошибок. Благодаря этому облегчается понимание программ а создание понятных программ, как, я надеюсь, вы теперь верите, это достоинство, которое уступает только благочестию.

Техника

Определите, будет ли исключительная ситуация проверяемой или непроверяемой.

Если вызывающий отвечает за проверку условия перед вызовом, сделайте исключительную ситуацию непроверяемой.

Если исключительная ситуация проверяемая, создайте новую исключительную ситуацию или используйте существующую.

Найдите все места вызова и модифицируйте их для использования исключительной ситуации.

Если исключительная ситуация непроверяемая, модифицируйте места вызова так, чтобы перед обращением к методу проводилась соответствующая проверка. Выполняйте компиляцию и тестирование после каждой модификации.

Если исключительная ситуация проверяемая, модифицируйте места вызова так, 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, будет ли исключительная ситуация проверяемой (или непроверяемой).

Создайте новый метод, использующий данную исключительную ситуацию.

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 странно ли, что в учебниках по программированию часто предполагается, что нельзя снять со счета больше, чем остаток на нем, в то вре мя как в реальной жизни часто это сделать можно?

```
class Account {
 int withdraw(int amount) {
    if (amount > _balance)
       return -1;
    else {
       _balance -= amount;
       return 0;
    }
 }
 private int _balance;
}
```

Желая изменить этот код так, чтобы использовать исключительную ситуацию, я сначала должен решить, будет ли она проверяемой или непроверяемой. Решение зависит от того, будет ли проверка остатка на счете перед снятием с него суммы входить в обязанности вызывающего или процедуры, выполняющей снятие. Если проверка остатка на счете вменяется в обязанность вызывающего, то вызов процедуры для снятия превышающей остаток суммы будет ошибкой программирования. Поскольку это программная ошибка, т. е. «дефект», я должен использовать непроверяемую исключительную ситуацию. Если проверка остатка на счету возлагается на процедуру снятия, я должен объявить исключительную ситуацию в интерфейсе. Таким способом я сообщаю вызывающему, что он должен ждать возможной исключительной ситуации и принять надлежащие меры.

Пример: непроверяемая исключительная ситуация

Возьмем сначала случай непроверяемой исключительной ситуации. Я ожидаю, что проверку выполнит вызывающий. Сперва я рассматриваю места вызова. В данном случае код возврата не должен использоваться нигде - это было бы ошибкой программирования. Увидев код типа

```
if (account.withdraw(amount) == -1)
 handleOverdrawn();
else
 doTheUsualThing();
```

я должен заменить его таким, например:

```
if (!account.canWithdraw(amount))
 handleOverdrawn();
else {
 account.withdraw(amount);
 doTheUsualThing();
}
```

После каждого изменения я могу выполнить компиляцию и тестирование.

Теперь надо убрать код ошибки и добавить генерирование исключительной ситуации в случае ошибки. Поскольку поведение является исключительным (по определению), я использую защитное предложение для проверки условия:

```
void withdraw(int amount) {
 if (amount > _balance)
    throw new IllegalArgumentException ("Слишком велика сумма");
 _balance -= amount;
}
```

Поскольку это ошибка программирования, следует сигнализировать о ней еще более явно при помощи 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 (“Отказ утверждения” + comment);
    }
 }
}
```

Пример: проверяемая исключительная ситуация

В случае проверяемой исключительной ситуации я действую несколько иначе. Сначала я создаю подходящий новый класс исключительной ситуации (или использую имеющийся):

```
class BalanceException extends Exception {}
```

Затем я модифицирую обращения к методу, чтобы они выглядели так:

```
try {
 account.withdraw(amount);
 doTheUsualThing();
}
catch (BalanceException e) {
 handleOverdrawn();
}
```

После этого я модифицирую метод, снимающий сумму со счета, используя в нем исключительную ситуацию:

```
void withdraw(int amount) throws BalanceException {
 if (amount > _balance) throw new BalanceException();
 _balance -= amount;
}
```

Если мест вызова много, то придется выполнять большую модификацию без промежуточных компиляции и тестирования.

В таких случаях можно применять временный промежуточный метод. Начну с того же случая, 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 состоит в создании нового метода withdraw, использующего исключительную ситуацию:

```
void newWithdraw(int amount) throws BalanceException {
    if (amount > _balance) throw new BalanceException();
    _balance -= amount;
}
```

Затем я модифицирую текущий метод withdraw так, 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 чего - заменить каждое из обращений к прежнему методу вызовом нового:

```
try {
    account.newWithdraw(amount);
    doTheUsualThing();
}
catch (BalanceException e) {
    handleOverdrawn();
}
```

При наличии обоих (старого и нового) методов можно выполнять компиляцию и тестирование после каждой модификации. Завершив изменения, я могу удалить старый метод и применить «Переименование метода» (Rename Method), чтобы дать новому методу старое имя.

Замена исключительной ситуации проверкой (Replace Exception with Test)

Возбуждается исключительная ситуация при выполнении условия, которое вызывающий мог сначала проверить.

Измените код вызова 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];
}
```

Мотивировка

Исключительные ситуации представляют собой важное достижение языков программирования. Они позволяют избежать написания сложного кода в результате «Замены кода ошибки исключительной ситуацией» (Replace Error Code with Exception). Как и многими другими удовольствиями, исключительными ситуациями иногда злоупотребляют, и они перестают быть приятными. Исключительные ситуации должны использоваться для локализации исключительного поведения, связанного с неожиданной ошибкой. Они не должны служить заменой проверкам выполнения условий. Если разумно предполагать от вызывающего проверки условия перед операцией, то следует обеспечить возможность проверки, а вызывающий не должен этой возможностью пренебрегать.

Техника

Поместите впереди проверку и скопируйте код из блока в соответствующую ветвь оператора if.

Поместите в блок catch утверждение, которое будет уведомлять о том, что этот блок выполняется.

Выполните компиляцию и тестирование.

Удалите блок catch, а также блок try, если других блоков catch нет.

Выполните компиляцию и тестирование.

Пример

Для этого примера возьмем объект, управляющий ресурсами, создание которых обходится дорого, но возможно повторное их использование. Хороший пример такой ситуации дают соединения с базами данных. У администратора соединений есть два пула, в одном из которых находятся ресурсы, доступные для использования, а в другом - уже выделенные. Когда клиенту нужен ресурс, администратор предоставляет его из пула доступных и переводит в пул выделенных. Когда клиент высвобождает ресурс, администратор возвращает его обратно. Если клиент запрашивает ресурс, когда свободных ресурсов нет, администратор создает новый ресурс.

Метод для выдачи ресурсов может выглядеть так:

```
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 неожиданным происшествием, поэтому использовать исключительную ситуацию не следует.

Чтобы убрать исключительную ситуацию, 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 таком варианте исключительная ситуация не должна возникать никогда. 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("available was empty on pop");
       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 удалю блок try полностью:

```
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 «Консолидацию дублирующихся условных фрагментов» (Consolidate Duplicate Conditional Fragments).

```
Resource getResource() {
 Resource result;
 if (_available.isEmpty())
    result = new Resource();
 else
    result = (Resource) _available.pop();
 _allocated.push(result);
 return result;
}
```

11 РЕШЕНИЕ ЗАДАЧ ОБОБЩЕНИЯ

Обобщение порождает собственную группу рефакторингов, в основном связанных с перемещением методов по иерархии наследования. «Подъем поля» (Pull Up Field) и «Подъем метода» (Pull Up Method) перемещают функцию вверх по иерархии, а «Спуск поля» (Push Down Field) и «Спуск метода» (Push Down Method) перемещают функцию вниз. Поднимать вверх конструкторы несколько труднее, и эти проблемы решаются с помощью «Подъема тела конструктора» (Pull Up Constructor Body). Вместо спуска конструктора часто более удобной оказывается «Замена конструктора фабричным методом» (Replace Constructor with Factory Method).

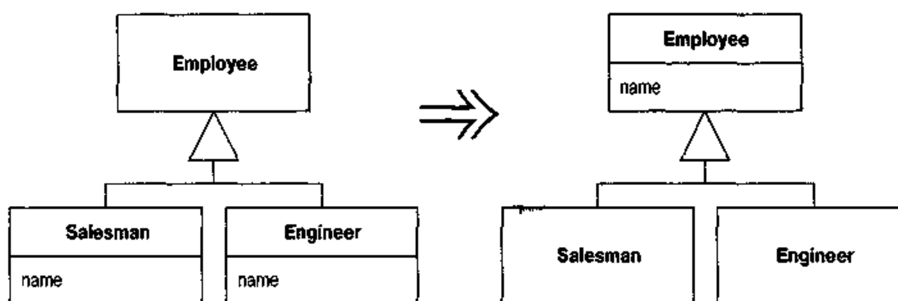
Если есть методы со сходными схемами тела, но отличающиеся в деталях, можно воспользоваться «Формированим шаблона метода» (Form Template Method), чтобы разделить отличия и сходства.

Помимо перемещения функции по иерархии можно изменять иерархию, создавая новые классы. «Выделение подкласса» (Extract Subclass), «Выделение родительского класса» (Extract Superclass) и «Выделение интерфейса» (Extract Interface) осуществляют это путем формирования новых элементов в различных местах. «Выделение интерфейса» (Extract Interface) особенно важно, когда нужно пометить небольшую часть функции для системы типов. Если в иерархии оказываются ненужные классы, их можно удалить с помощью «Свертывания иерархии» (Collapse Hierarchy).

Иногда обнаруживается, что иерархическое представление - не лучший способ моделирования ситуации, и вместо него требуется делегирование. «Замена наследования делегированием» (Replace Inheritance with Delegation) позволяет провести это изменение. Иногда жизнь поворачивается иначе, и приходится применять «Замену делегирования наследованием» (Replace Delegation with Inheritance).

Подъем поля (Pull Up Field)

В двух подклассах есть одинаковое поле. Переместите поле в родительский класс.



Мотивировка

Если подклассы разрабатываются независимо или объединяются при проведении рефакторинга, в них часто оказываются дублирующиеся функции. В частности, дублироваться могут некоторые поля. Иногда у таких полей оказываются одинаковые имена, но это необязательно. Единственный способ разобраться в происходящем - посмотреть на поля и понять, как с ними работают другие методы. Если они используются сходным образом, можно их обобщить.

В результате дублирование уменьшается в двух отношениях. Удаляются дублирующиеся объявления данных и появляется возможность переместить из подклассов в родительский класс поведение, использующее поля.

Техника

Рассмотрите все случаи использования полей-кандидатов на перемещение и убедитесь, что эти случаи одинаковы.

Если у полей разные имена, переименуйте их, дав то имя, которое вы считаете подходящим для поля в родительском классе.

Выполните компиляцию и тестирование.

Создайте новое поле в родительском классе.

Если поля закрытые, следует сделать поле родительского класса защищенным, чтобы подклассы могли ссылаться на него.

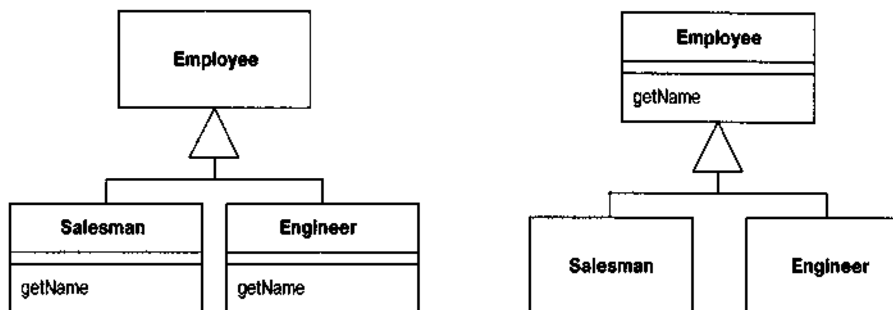
Удалите поля из подклассов.

Выполните компиляцию и тестирование.

Рассмотрите возможность применения к новому полю «Самоинкапсуляции поля» (Self Encapsulate Field).

Подъем метода (Pull Up Method)

В подклассах есть методы с идентичными результатами. Переместите их в родительский класс.



Мотивировка

Дублирование поведения необходимо исключать. Хотя два дублирующихся метода прекрасно работают в существующем виде, они представляют собой питательную среду для возникновения ошибок в будущем. При наличии дублирования всегда есть риск, что при модификации одного метода второй будет пропущен. Обычно находить дубликаты трудно.

Простейший случай «Подъема метода» (Pull Up Method) возникает, когда тела обоих методов одинаковы, что указывает на произведенные копирование и вставку. Конечно, ситуация не всегда столь очевидна. Можно просто выполнить рефакторинг и посмотреть, как пройдут тесты, но тогда придется полностью положиться на свои тесты. Обычно бывает полезно поискать различия в методах, к которым применяется рефакторинг; часто в них обнаруживается поведение, протестировать которое забыли.

Часто условия для «Подъема метода» (Pull Up Method) возникают после внесения некоторых изменений. Иногда в разных классах обнаруживаются два метода, которые можно параметризовать так, что они приводят, в сущности, к одному и тому же методу. Тогда, если двигаться самыми мелкими шагами, надо параметризовать каждый метод в отдельности, а затем обобщить их. Если чувствуете себя уверенно, можете выполнить все в один прием.

Особый случай «Подъема метода» (Pull Up Method) возникает, когда некоторый метод подкласса перегружает метод родительского класса, но выполняет те же самые действия.

Самое неприятное в «Подъеме метода» (Pull Up Method) то, что в теле методов могут быть ссылки на функции, находящиеся в подклассе, а не в родительском классе. Если функция представляет собой метод, можно обобщить и его либо создать в родительском классе абстрактный метод. Чтобы это работало, может потребоваться изменить сигнатуру метода или создать делегирующий метод.

Если есть два сходных, но неодинаковых метода, можно попробовать осуществить «Формирование шаблона метода» (Form Template Method).

Техника

Изучите методы и убедитесь, что они идентичны.

Если методы выполняют, по-видимому, одно и то же, но не идентичны, примените к одному из них замену алгоритма, чтобы сделать идентичными.

Если у методов разная сигнатура, измените ее на ту, которую хотите использовать в родительском классе.

Создайте в родительском классе новый метод, скопируйте в него тело одного из методов, настройте и скомпилируйте.

Если вы. работаете со строго типизированным языком и метод вы зывает другой метод, который присутствует в обоих подклассах, но не в родительском классе, объявите абстрактный метод в родительском классе.

Если метод использует поле подкласса, обратитесь к «Подъему поля» (Pull Up Field) или «Самоинкапсуляции поля» (Self Encapsulate Field), объявите абстрактный метод получения и используйте его.

Удалите один метод подкласса.

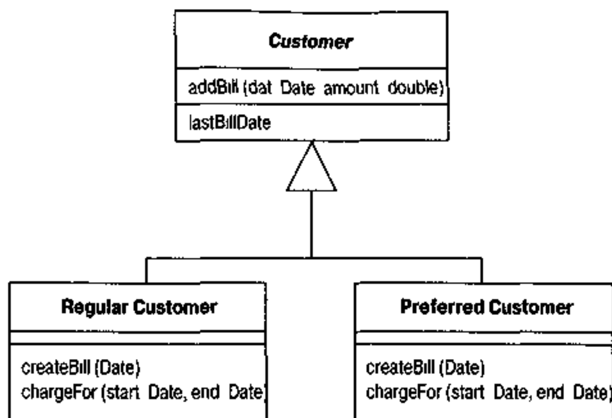
Выполните компиляцию и тестирование.

Продолжайте удаление методов подклассов и тестирование, пока не останется только метод родительского класса.

Посмотрите, кто вызывает этот метод, и выясните, нельзя ли заменить тип объекта, с которым производятся манипуляции, родительским классом.

Пример

Рассмотрим класс клиента с двумя подклассами: обычным клиентом и привилегированным клиентом.



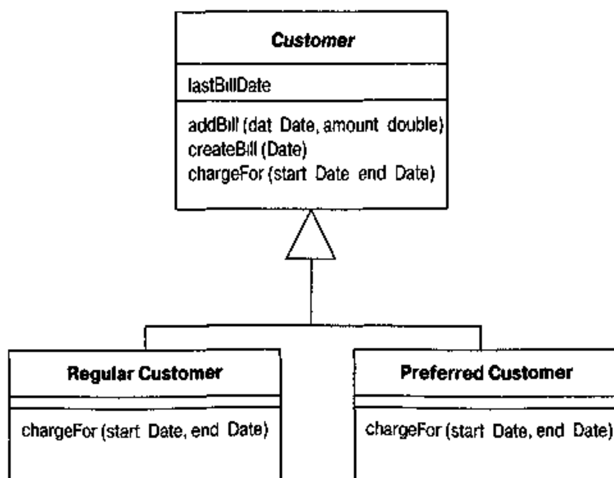
Метод createBill идентичен в обоих классах:

```
void createBill (Date date) {
 double chargeAmount = chargeFor (lastBillDate, date);
 addBill (date, charge);
}
```

Я не могу переместить метод в родительский класс, потому что chargeFor различный в каждом подклассе. Сначала я должен объявить его в родительском классе 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 из другого подкласса, компилирую и тестирую:



Подъем тела конструктора (Pull Up Constructor Body)

Имеются конструкторы подклассов с почти идентичными телами. Создайте конструктор в родительском классе: 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;
    }
```

Мотивировка

Конструкторы - хитрая штука. Это не вполне обычные методы, поэтому при работе с ними возникает больше ограничений. Когда обнаруживаются методы подклассов с одинаковым поведением, первой мыслью должно быть выделение общего поведения в метод и подъем его в родительский класс. Однако для конструкторов общим поведением часто является построение экземпляра. В таком случае нужен конструктор родительского класса, вызываемый подклассами. Часто это все - тело конструктора. Прибегнуть к «Подъему метода» (Pull Up Method) здесь нельзя, потому что конструкторы не могут наследоваться (отвратительно, правда?).

Если этот рефакторинг становится сложным, можно попробовать воспользоваться вместо него «Заменой конструктора фабричным методом » (Replace Constructor with Factory Method).

Техника

Определитеконструктор родительскогокласса.

Переместите общий начальный код из подкласса в конструктор родительского класса.

Может оказаться, что это весь код.

Попробуйте переместить общий код в начало конструктора.

Вызовите конструктор родительского класса в качестве первого шага в конструкторе подкласса.

Если общим является весь код, этот вызов будет единственной строкой в конструкторе подкласса.

Выполните компиляцию и тестирование.

Если есть общий код далее, примените к нему «Выделение метода» (Extract Method) и выполните «Подъем метода» (Pull Up Method).

Пример

Вот пример классов менеджера и служащего:

```
    class Employee {
     protected String _name;
     protected String _id;
    }
    class Manager extends Employee {
     public Manager (String name, Striftg id, int grade) {
        _name = name;
        _id = id;
        _grade = grade;
     }
     private int _grade;
    }
```

Поля Employee должны быть установлены в конструкторе для Employee. Я определяю конструктор и делаю его защищенным, что должно предупреждать о том, что подклассы должны его вызывать:

```
    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 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(); //это выполняют все подклассы
 }
 boolean isPriviliged(){
    return _grade > 4;
 }
}
```

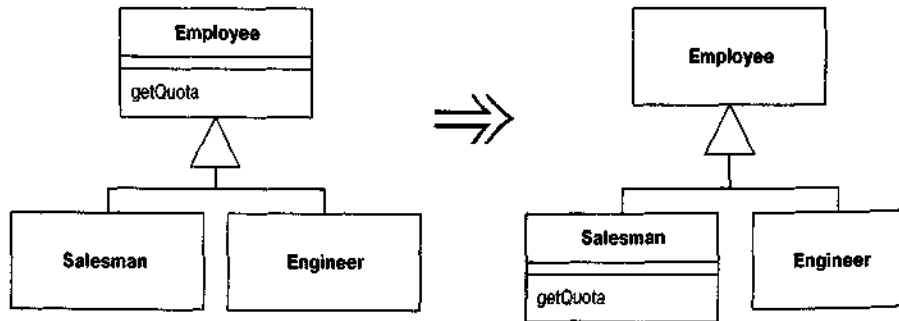
Я не могу переместить поведение assignCar в конструктор родительского класса, потому что оно должно выполняться после присваивания grade полю, поэтому нужно применить «Выделение метода» (Extract Method) и «Подъем метода» (Pull Up Method).

```
class Employee {
 void initialize() {
    if (isPriviliged())
       assignCar();
 }
}
class Manager {
 public Manager (String name, String id, int grade) {
    super (name, id):
    _grade = grade;
    Initialize();
 }
}
```

Спуск метода (Push Down Method)

В родительском классе есть поведение, относящееся только к некоторым из его подклассов.

Переместить поведение в эти подклассы.



Мотивировка

«Спуск метода» (Push Down Method) решает задачу, противоположную «Подъему метода» (Pull Up Method). Я применяю его для того, чтобы переместить поведение из родительского класса в конкретный подкласс, обычно потому, что оно имеет смысл только в нем. Это часто происходит при «Выделении подкласса» (Extract Subclass).

Техника

Объявите метод во всех подклассах и скопируйте тело в каждый подкласс.

Может потребоваться объявить поля как защищенные, чтобы, метод смог к ним обращаться. Обычно это делается, если планируется позднее опустить поле. Либо нужно использовать метод доступа в родительском классе. Если метод доступа закрытый, необходимо объявить его как защищенный.

Удалите метод из родительского класса.

Может потребоваться модифицировать вызывающий его код, так чтобы использовать подкласс в объявлениях переменных и параметров.

Если есть смысл в обращении к методу через переменную родительского класса, не планируется удаление метода из каких-либо подклассов и родительский класс является абстрактным, можно объявить метод в родительском классе как абстрактный.

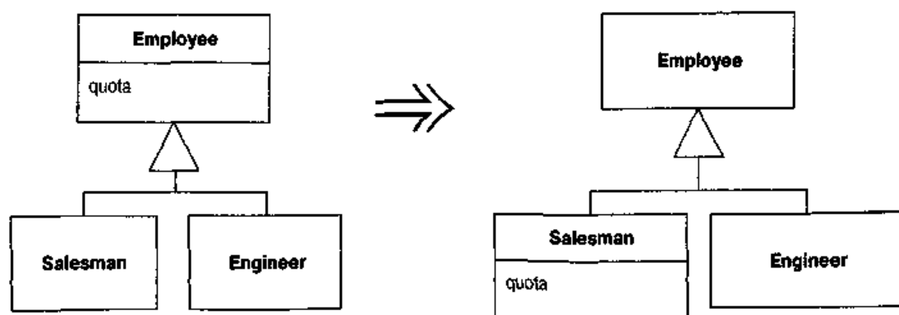
Выполните компиляцию и тестирование.

Удалите метод из всех подклассов, в которых он не нужен.

Выполните компиляцию и тестирование.

Спуск поля (Push Down Field)

Есть поле, используемое лишь некоторыми подклассами» Переместите поле в эти подклассы.



Мотивировка

«Спуск поля» (Push Down Field) решает задачу, противоположную «Подъему поля» (Pull Up Field). Он применяется, когда поле требуется не в родительском классе, а в некоторых подклассах.

Техника

Объявите поле во всех подклассах.

Удалите поле из родительского класса.

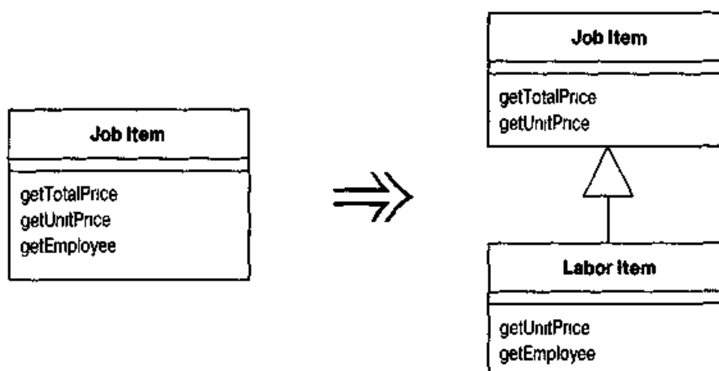
Выполните компиляцию и тестирование.

Удалите поле из всех подклассов, где оно не нужно.

Выполните компиляцию и тестирование.

Выделение подкласса (Extract Subclass)

В классе есть функции, используемые только в некоторых случаях. Создайте подкласс для этого подмножества функций.



Мотивировка

Главным побудительным мотивом применения «Выделения подкласса» (Extract Subclass) является осознание того, что в классе есть поведение, используемое в одних экземплярах и не используемое в других. Иногда об этом свидетельствует код типа, и тогда можно обратиться к «Замене кода типа подклассами» (Replace Type Code with Subclasses) или «Замене кода типа состоянием/стратегией» (Replace Type Code with State /Strategy). Но применение подклассов не обязательно предполагает наличие кода типа.

Основной альтернативой «Выделению подкласса» (Extract Subclass) служит «Выделение класса» (Extract Class). Здесь нужно сделать выбор между делегированием и наследованием. «Выделение подкласса» (Extract Subclass) обычно легче осуществить, но оно имеет некоторые ограничения. После того как объект создан, нельзя изменить его поведение, основанное на классе. Можно изменить основанное на классе поведение с помощью «Выделения класса» (Extract Class), просто подключая разные компоненты. Кроме того, для представления одного набора вариантов поведения можно использовать только подклассы. Если требуется, чтобы поведение класса изменялось несколькими разными способами, для всех из них, кроме одного, должно быть применено делегирование.

Техника

Определите новый подкласс исходного класса.

Создайте для нового подкласса конструкторы.

В простых случаях можно скопировать аргументы родительского класса и вызывать конструктор родительского класса с помощью super.

Если нужно скрыть использование подкласса от клиента, можно воспользоваться «Заменой конструктора фабричным методом» (Replace Constructor with Factory Method).

Найдите все вызовы конструкторов родительского класса. Если в них нужен подкласс, замените их вызовом нового конструктора.

Если конструктору подкласса нужны другие аргументы, измените его с помощью «Переименования метода» (Rename Method). Если некоторые параметры конструктора родительского класса больше не нужны, примените «Переименование метода» (Rename Method) также к нему.

Если больше нельзя напрямую создавать экземпляры родительско го класса, объявите его абстрактным.

С помощью «Спуска метода» (Push Down Method) и «Спуска поля» (Push Down Field) поочередно переместите функции в подкласс.

В противоположность «Выделению класса» (Extract Class) обычно проще работать сначала с методами, а потом с данными.

При спуске открытого метода может потребоваться переопреде лить тип переменной или параметра вызывающего, чтобы вызы вался новый метод. Компилятор обнаруживает такие случаи.

Найдите все поля, определяющие информацию, на которую теперь указывает иерархия (обычно это булева величина или код типа). Устраните их с помощью «Самоинкапсуляции поля» (Self Encapsulate Field) и замены метода получения значения полиморфными константными методами. Для всех пользователей этого поля

должен быть проведен рефакторинг «Замена условного оператора полиморфизмом» (Replace Conditional with Polymorphism).

Если есть методы вне класса, использующие метод доступа, попробуйте с помощью «Перемещения метода» (Move Method) перенести метод в этот класс; затем примените «Замену условного оператора полиморфизмом» (Replace Conditional with Polymorphism).

Выполняйте 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, потому что некоторое поведение и данные требуются только в этом случае. Создам новый класс:

```
class LaborItem extends JobItem {}
```

Прежде всего, мне нужен конструктор для этого класса, потому что у JobItem нет конструктора без аргументов. Для этого я копирую сигнатуру родительского конструктора:

```
    public LaborItem (int unitPrice, int quantity, boolean isLabor, Employee employee)
{
     super (unitPrice, quantity, isLabor, employee);
    }
```

Этого достаточно, чтобы новый подкласс компилировался. Однако этот конструктор путаный: одни аргументы нужны для LaborItem, а другие - нет. Я займусь этим позднее.

На следующем этапе осуществляется поиск обращений к конструктору JobItem и случаи, когда вместо него следует вызывать конструктор LaborItem. Поэтому утверждения вида

```
    JobItem j1 = new JobItem (0, 5, true, kent);
```

превращаются в

```
    JobItem j1 = new LaborItem (0, 5, true, kent);
```

На данном этапе я не трогал тип переменной, а изменил лишь тип конструктора. Это вызвано тем, что я хочу использовать новый тип только там, где это необходимо. В данный момент у меня нет специфического интерфейса для подкласса, поэтому я не хочу пока объявлять какие-либо разновидности.

Теперь подходящее время, чтобы привести в порядок списки параметров конструктора. К каждому из них я применяю «Переименование метода» (Rename Method). Сначала я обращаюсь к родительскому классу. Я создаю новый конструктор и объявляю прежний защищенным (подклассу он по-прежнему нужен):

```
    class JobItem {
     protected JobItem (int unitPnce, 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 «Переименование метода» (Rename Method) к конструктору подкласса:

```
    class LaborItem {
     public LaborItem (int quantity, Employee employee) {
        super (0, quantity, true, employee);
     }
    }
```

Я продолжаю пока использовать защищенный конструктор родительского класса.

Теперь я могу начать спускать в подкласс функции JobItem. Сперва я займусь методами. Начну с применения «Спуска метода» (Push Down Method) к getEmployee:

```
    class LaborItem {
     public Employee getEmployee() {
        return _employee;
     }
    }
    class JobItem {
     protected Employee _employee;
```

```
    }
```

Поскольку поле _employee позднее будет спущено в подкласс, я пока объявляю его защищенным.

После того как поле _employee защищено, я могу привести в порядок конструкторы, чтобы _employee инициализировалось только в подклассе, куда оно спускается:

```
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, поэтому можно удалить это поле. Лучше всего сделать это, сначала применив «Самоинкапсуляцию поля» (Self Encapsulate Field), а затем изменив метод доступа, 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; }
}
```

После этого можно избавиться от поля isLabor.

Теперь можно посмотреть на пользователей методов isLabor. Они должны быть подвергнуты рефакторингу «Замена условного оператора полиморфизмом» (Replace Conditional with Polymorphism). Беру метод

```
class JobItem {
 public int getUnitPrice(){
    return (isLabor()) ? _employee.getRate() : _unitPrice;
 }
}
```

и заменяю его следующим:

```
class JobItem {
 public int getUnitPrice(){
    return _unitPrice;
 }
}
class LaborItem {
 public int getUnitPrice(){
    return _employee.getRate();
 }
```

```
    }
```

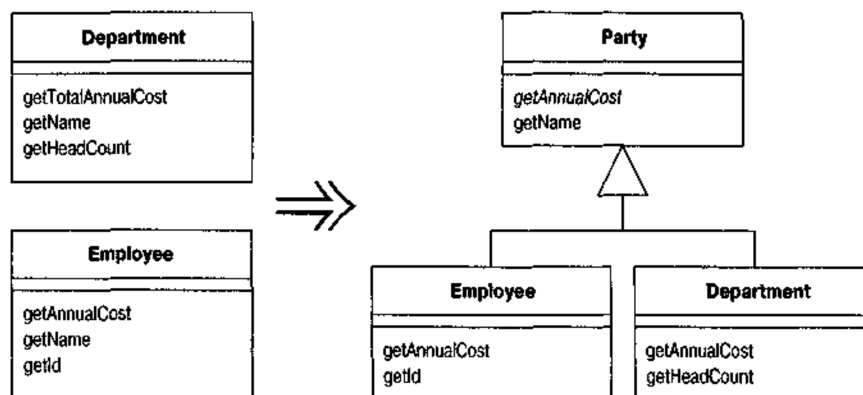
После того как группа методов, использующих некоторые данные, перемещена в подкласс, к этим данным можно применить «Спуск поля» (Push Down Field). Если я не могу применить его из-за того, что эти данные используются некоторым методом, это свидетельствует о необходимости продолжить работу с методами, применяя «Спуск метода» (Push Down Method) или «Замену условного оператора полиморфизмом» (Replace Conditional with Polymorphism).

Поскольку цена узла (unitPrice) используется только элементами, не представляющими трудовые затраты (элементы работ, являющиеся запчастями), я могу снова применить к JobItem «Выделение подкласса» (Extract Subclass) и создать класс, представляющий запчасти. В итоге класс JobItem станет абстрактным.

Выделение родительского класса (Extract Superclass)

Имеются два класса со сходными функциями.

Создайте для них общий родительский класс и переместите в него общие функции.



Мотивировка

Дублирование кода представляет собой один из главных недостатков систем. Если одно и то же делается в нескольких местах, то когда придется сделать что-то другое, надо будет редактировать больше мест, чем следует.

Одним из видов дублирования кода является наличие двух классов, выполняющих сходные задачи одинаковым способом или сходные задачи разными способами. Объекты предоставляют встроенный механизм для упрощения такой ситуации с помощью наследования. Однако часто общность оказывается незамеченной до тех пор, пока не будут созданы какие-то классы, и тогда структуру наследования требуется создавать позднее.

Альтернативой служит «Выделение родительского класса» (Extract Superclass). По существу, необходимо сделать выбор между наследованием и делегированием. Наследование представляется более простым способом, если у двух классов одинаковы как интерфейс, так и поведение. Если выбор сделан неправильно, всегда можно в дальнейшем применить «Замену наследования делегированием» (Replace Inheritance with Delegation).

Техника

Создайте пустой абстрактный родительский класс; сделайте исходные классы его подклассами.

Поочередно применяйте «Подъем поля» (Pull Up Field), «Подъем метода» (Pull Up Method) и «Подъем тела конструктора» (Pull Up Constructor Body), чтобы переместить общие элементы в новый родительский класс.

Обычно лучше сначала переместить поля.

Если в подклассах есть методы с разными сигнатурами, но одинаковым назначением, примените «Переименование метода» (Rename Method), чтобы привести их к одинаковому наименованию, а затем выполните «Подъем метода» (Pull Up Method).

Если есть методы с одинаковыми сигнатурами, но различающимися телами, объявите общую сигнатуру как абстрактный метод в родительском классе.

Если есть методы с разными телами, выполняющие одно и то же, можно попробовать применить «Замещение алгоритма» (Substitute Algorithm), чтобы скопировать тело одного метода в другой. Если это удастся, то можно воспользоваться «Подъемом метода» (Pull Up Method).

После каждого подъема в родительский класс выполняйте компиляцию и тестирование.

Изучите методы, оставшиеся в подклассах. Посмотрите, есть ли в них общие участки; если да, можно применить к ним «Выделение метода» (Extract Method) с последующим «Поднятием метода» (Pull Up Method). Если общий поток команд сходен, может открыться возможность применить «Формирование шаблона метода» (Form Template Method).

После подъема в родительский класс всех общих элементов проверьте каждого клиента подклассов. Если они используют только общий интерфейс, можно заменить требуемый тип родительским классом.

Пример

Для этого случая 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();
}
```

Здесь есть пара областей, обладающих общностью. Во-первых, как у служащих, так и у отделов есть имена или названия (name). Во-вторых, для обоих есть годовой бюджет (annualCost), хотя методы их расчета слегка различаются. Я выделяю родительский класс для обеих этих функций. На первом этапе создается новый родительский класс, а имеющиеся родительские классы определяются как его подклассы:

```
abstract class Party {}
class Employee extends Party
```

```
class Department extends Party
```

Теперь я начинаю поднимать функции в родительский класс. Обычно проще сначала выполнить «Подъем поля» (Pull Up Field):

```
class Party {
 protected String _name;
}
```

После этого можно применить «Подъем метода» (Pull Up Method) к методам доступа:

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

Я предпочитаю, чтобы поля были закрытыми. Для этого мне нужно выполнить «Подъем тела конструктора» (Pull Up Constructor Body), чтобы присвоить 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 назначение, поэтому у них должно быть одинаковое название. Сначала я применяю «Переименование метода» (Rename Method), чтобы привести их к одному и тому же названию:

```
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;
 }
}
```

Их тела пока различаются, поэтому я не могу применить «Подъем метода» (Pull Up Method), однако я могу объявить в родительском классе абстрактный метод:

```
abstract public int getAnnualCost();
```

Осуществив эти очевидные изменения, я рассматриваю клиентов обоих классов, чтобы выяснить, можно ли изменить их так, чтобы они использовали новый родительский класс. 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:

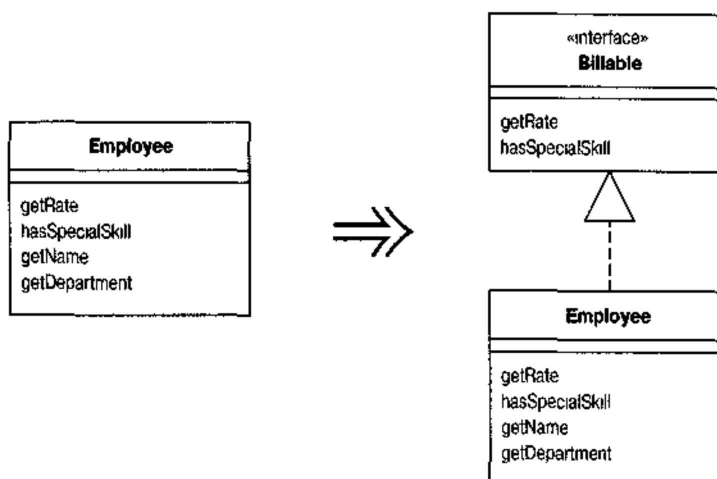
```
class Department {
 public int getAnnualCost(){
   Enumeration e = getStaff();
   int result = 0;
   while (e.hasMoreElements()) {
     Party each = (Party) e.nextElement();
     result += each.getAnnualCost();
   }
   return result;
 }
}
```

Такое поведение открывает новую возможность. Я могу рассматривать применение паттерна «компоновщик»(Composite) [Gang of Four] к Department и Employee. Это позволит мне включать один отдел в другой. В результате создается новая функциональность, так что нельзя, строго говоря, назвать это рефакторингом. Если бы требовалась компоновка, я получил бы ее, изменив имя поля staff, чтобы картина была нагляднее. Такое изменение повлекло бы соответствующее изменение имени addStaff и замену параметра на Party. В окончательной редакции метод headCount должен быть сделан рекурсивным. Это можно осуществить, создав метод headCount подсчета численности для Employee, который просто возвращает 1, и применив «Замещение алгоритма» (Substitute Algorithm) к подсчету численности отдела, чтобы суммировать значения headCount его составляющих.

Выделение интерфейса (Extract Interface)

Несколько клиентов пользуются одним и тем же подмножеством интерфейса класса или в двух классах часть интерфейса является общей.

Выделите это подмножество в интерфейс.



Мотивировка

Классы взаимодействуют друг с другом разными способами. Взаимодействие с классом часто означает обращение ко всем предоставляемым классом функциям. В других случаях группа клиентов использует только определенное подмножество предоставляемых классом функций. Бывает, что класс должен работать с любым классом, который может обрабатывать определенные запросы.

Для двух последних случаев часто полезно заставить подмножество обязанностей выступать от своего собственного имени, чтобы сделать отчетливым его использование в системе. Благодаря этому легче видеть,

как разделяются обязанности в системе. Если для поддержки подмножества необходимы новые классы, проще точно увидеть, что подходит подмножеству.

Во многих объектно-ориентированных языках такая возможность поддерживается множественным наследованием. Для всех сегментов поведения создаются классы, которые объединяются в реализации. В Java наследование одиночное, но допускается сформулировать и реализовать такого рода требование с помощью интерфейсов. Интерфейсы оказали большое влияние на подход программистов к проектированию приложений Java. Даже программирующие на Smalltalk полагают, что интерфейсы являются шагом вперед!

Есть некоторое сходство между «Выделением родительского класса» (Extract Superclass) и «Выделением интерфейса» (Extract Interface). «Выделение интерфейса» позволяет выявлять только общие интерфейсы, но не общий код. Применяя «Выделение интерфейса», можно получить душок дублирующегося кода. Эту проблему можно уменьшить, применив «Выделение класса» (Extract Class) для помещения поведения в компонент и делегирования ему обработки. Если объем общего поведения существенен, то применить «Выделение родительского класса» (Extract Superclass) проще, но вы получите только один родительский класс.

Интерфейсы бывают кстати, когда классы играют особые роли в разных ситуациях. Воспользуйтесь «Выделением интерфейса» (Extract Interface) для каждой роли. Еще один удобный случай возникает, когда требуется описать выходной интерфейс класса, т. е. операции, которые класс выполняет на своем сервере. Если в будущем предполагается разрешить использование серверов других видов, все они должны реализовывать этот интерфейс.

Техника

Создайте пустой интерфейс.

Объявите в интерфейсе общие операции.

Объявите соответствующие классы как реализующие интерфейс.

Измените клиентские объявления типов, 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 того 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 есть много других характеристик помимо информации о ставке оплаты и специальных навыках, но в данном приложении требуются только они. Тот факт, что требуется только это подмножество, можно подчеркнуть, определив для него интерфейс:

```
interface Billable {
 public int getRate();
 public boolean hasSpecialSkill();
}
```

После этого можно объявить Employee как реализующий этот интерфейс:

```
class Employee implements Billable
```

Когда это сделано, можно изменить объявление charge, чтобы показать, что используется только эта часть поведения Employee:

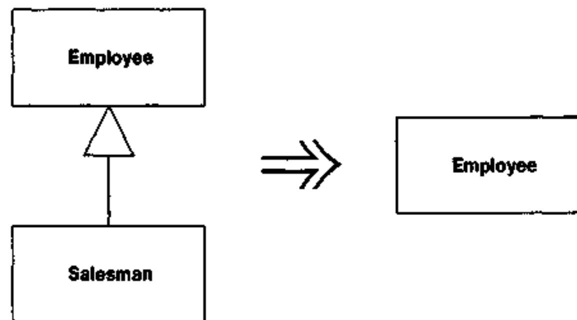
```
double charge(Billable emp, int days) {
 int base = emp.getRate() * days;
 if (emp.hasSpecialSkill())
    return base * 1.05;
 else
```

```
        return base;
    }
```

В данном случае получена скромная выгода в виде документированности кода. Такая выгода не стоит труда для одного метода, но если бы несколько классов стали применять интерфейс Billable, это было бы полезно. Крупная выгода появится, когда я захочу выставлять счета еще и для компьютеров. Все, что мне для этого нужно - реализовать в них интерфейс Billable, и тогда можно включать компьютеры в табель учета времени.

Свертывание иерархии (Collapse Hierarchy)

Родительский класс и подкласс мало различаются. Объедините их в один.



Мотивировка

Если поработать некоторое время с иерархией классов, она сама может легко стать запутанной. При проведении рефакторинга иерархии час то методы и поля перемещаются вверх или вниз. По завершении вполне может обнаружиться подкласс, не вносящий никакой дополнительной ценности, поэтому нужно объединить классы вместе.

Техника

Выберите, который из классов будет удаляться - родительский класс или подкласс.

Воспользуйтесь «Подъемом поля» (Pull Up Field) и «Подъемом метода» (Pull Up Method) или «Спуском поля» (Push Down Field) и «Спуском метода» (Push Down Method) для перемещения поведения и данных удаляемого класса в класс, с которым он объединяется.

Выполняйте компиляцию и тестирование для каждого перемещения.

Замените ссылки на удаляемый класс ссылками на объединенный класс. Это воздействует на объявления переменных, типы параметров и конструкторы.

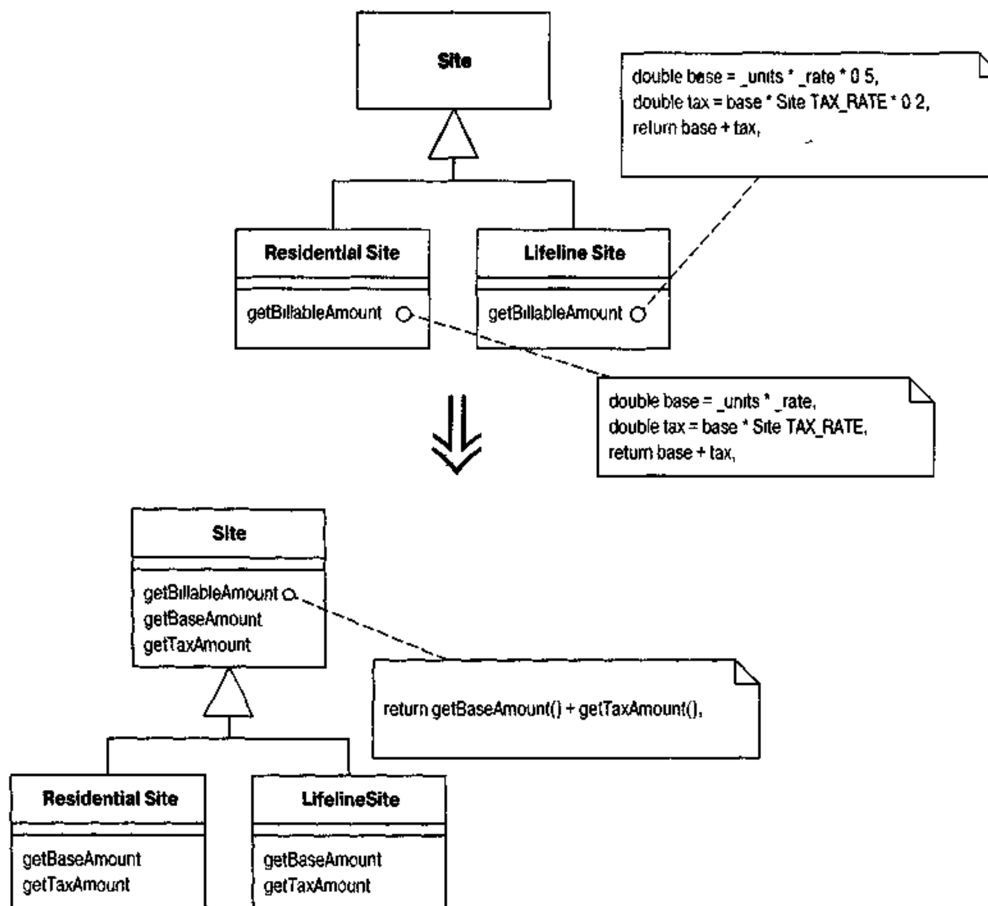
Удалите пустой класс.

Выполните компиляцию и тестирование.

Формирование шаблона метода (Form Template Method)

Есть два метода в подклассах, выполняющие аналогичные шаги в одинаковом порядке, однако эти шаги различны.

Образуйте из этих шагов методы с одинаковой сигнатурой, чтобы исходные методы стали одинаковыми. После этого можно их под нятъ в родительский класс.



Мотивировка

Наследование представляет собой мощный инструмент для устранения дублирования поведения. Встретив два аналогичных метода в подклассах, мы хотим объединить их в родительском классе. Но что если они не в точности совпадают? Что тогда делать? Мы по-прежнему должны устранить как можно больше дублирования, но сохранить существенные различия.

Часто встречается случай, когда два метода выполняют во многом аналогичные шаги в одинаковой последовательности, но эти шаги разные. В таком случае можно переместить последовательность шагов в родительский класс и позволить полиморфизму выполнить свою роль в обеспечении того, чтобы различные шаги выполняли свои действия по-разному. Такой прием называется шаблоном метода (template method) [Gang of Four].

Техника

Выполните декомпозицию методов, чтобы все выделенные методы полностью совпадали или полностью различались.

С помощью «Подъема метода» (Pull Up Method) переместите идентичные методы в родительский класс.

К различающимся методам примените «Переименование метода» (Rename Method) так, чтобы сигнатуры всех методов на каждом шаге были одинаковы.

В результате исходные методы, становятся одинаковыми в том смысле, что они осуществляют одну и ту же последовательность вызовов методов, но вызовы, по-разному обрабатываются подклассами.

Выполняйте компиляцию и тестирование после каждого изменения сигнатуры.

Примените «Подъем метода» (Pull Up Method) к одному из исходных методов. Определите сигнатуры различных методов как абстрактные методы родительского класса.

Выполните компиляцию и тестирование.

Удалите остальные методы и выполняйте компиляцию и тестирование после каждого удаления.

Пример

Закончу начатое в главе 1. Там у меня был класс Customer с двумя методами для печати выписки по счету клиента. Метод statement выводит выписки в ASCII:

```
public String statement() {
 Enumeration rentals = _rentals.elements();
 String result = "Учет аренды для" + getName() + "\n";
 while (rentals.hasMoreElements()) {
   Rental each = (Rental) rentals.nextElement();
   // показать результаты для этой аренды
   result += "\t" + each.getMovie().getTitle()+ "\t" +
           String.valueOf(each.getCharge()) + "\n";
 }
 // добавить нижний колонтитул
 result += "Сумма задолженности составляет" +
         String.valueOf(getTotalCharge()) + "\n";
 result += "Вы заработали" + String.valueOf(getTotalFrequentRenterPoints()) +
           "очков за активность";
 return result;
}
```

в то время как htmlStatement делает выписку в HTML:

```
public String htmlStatement() {
 Enumeration rentals = _rentals.elements();
 String result = "<H1> Операции аренды для <EM>"+ getName() +
           "</EM></H1><P>\n";
 while (rentals.hasMoreElements()) {
   Rental each = (Rental) rentals.nextElement();
   // показать результаты по каждой аренде
   result += each.getMovie().getTitle() + String.valueOf(each.getCharge()) +
        "<BR>\n";
 }
 // добавить нижний колонтитул
 result += "<P>Ваша задолженность составляет <EM>" +
         String.valueOf(getTotalCharge()) + "</EM><P>\n";
 result += "На этой аренде вы заработали <EM>"+
         String.valueOf(getTotalFrequentRenterPoints()) +
         "</EM> очков за активность<P>";
 return result;
}
```

Прежде чем применить «Формирование шаблона метода» (Form Template Method), я должен устроить так, чтобы эти два метода появились в подклассах некоторого общего родительского класса. Я делаю это с помощью объекта методов [Beck], создавая отдельную иерархию стратегий для вывода выписок (рисунок 11.1).

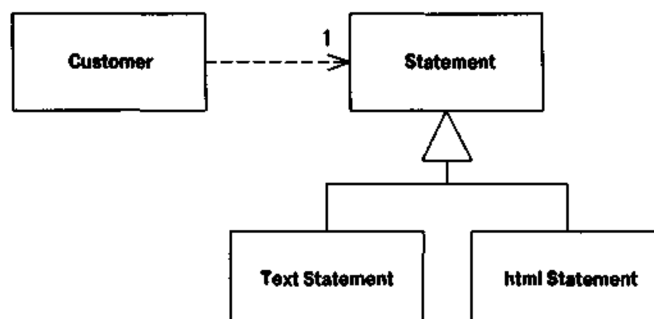


Рисунок 11.1 - Применение стратегии к методам statement

```
class Statement {}
class TextStatement extends Statement {}
class HtmlStatement extends Statement {}
```

Теперь с помощью «Перемещения метода» (Move Method) я переношу эти два метода statement 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 = "Операции аренды для" + aCustomer.getNameO + "\n";
    while (rentals.hasMoreElements()) {
       Rental each = (Rental) rentals.nextElement();
       // показать результаты по этой аренде
       result += "\t"+ each.getMovie().getTitle()+ "\t" +
                String.valueOf(each.getCharge()) + "\n";
    }
    // добавить нижний колонтитул
    result += "Ваша задолженность составляет" +
             String.valueOf(aCustomer.getTotalCharge()) + "\n";
    result += "Вы заработали" +
             String.valueOf(aCustomer.getTotalFrequentRenterPoints()) +
             "очков за активность";
    return result;
 }
}
class HtmlStatement {
 public String value(Customer aCustomer) {
    Enumeration rentals = aCustomer.getRentals();
    String result = "<H1> Операции аренды для <EM>" + aCustomer.getName() +
             "</EM></H1><P>\n";
    while (rentals.hasMoreElements()) {
       Rental each = (Rental) rentals.nextElement();
       // показать результаты по каждой аренде
       result += each.getMovie().getTitle()+
                String.valueOf(each.getCharge()) + "<BR>\n";
    }
    // добавить нижний колонтитул
    result += "<P>Ваша задолженность составляет <EM>" +
```

```
            String.valueOf (aCustomer.getTotalCharge()) +
            "</EM><P>\n";
    result += "На этой аренде вы заработали <EM>" +
            String.valueOf (aCustomer.getTotalFrequentRenterPoints()) +
            "</EM> очков за активность<P>";
    return result;
 }
}
```

Перемещая методы, я переименовал методы statement для лучшего соответствия стратегии. Я дал им одинаковое имя, потому что различие между ними теперь лежит в классе, а не в методе. (Для тех, кто изучает пример: мне пришлось также добавить метод getRentals в Customer и ослабить видимость getTotalCharge и getTotalFrequentRenterPoints.)

При наличии двух аналогичных методов в подклассах можно начать применение «Формирования шаблона метода» (Form Template Method). Ключ к этому рефакторингу лежит в разделении различающегося кода и совпадающего, применяя «Выделение метода» (Extract Method) для извлечения тех участков, которые различны в двух методах. При каждом выделении я создаю методы с различными телами, но одинаковой сигнатурой.

Первый пример - вывод заголовка. В обоих методах информация получается из Customer, но результирующая строка по-разному форматируется. Я могу выделить форматирование этой строки в отдельные методы с одинаковой сигнатурой:

```
class TextStatement {
 String headerString(Customer aCustomer) {
    return "Операции аренды для" + aCustomer.getName() + "\n";
 }
 public String value(Customer aCustomer) {
    Enumeration rentals = aCustomer.getRentals();
    String result = headerString(aCustomer);
    while (rentals.hasMoreElements()) {
       Rental each = (Rental) rentals.nextElement();
       // показать результаты по этой аренде
       result += "\t"+ each.getMovie().getTitle()+ "\t" +
               String.valueOf(each.getCharge()) + "\n";
    }
    // добавить нижний колонтитул
    result += "Ваша задолженность составляет" +
            String.valueOf(aCustomer.getTotalCharge()) + "\n";
    result += "Вы заработали" +
            String.valueOf(aCustomer.getTotalFrequentRenterPoints()) +
            "очков за активность";
    return result;
 }
}


class HtmlStatement {
 String headerString(Customer aCustomer) {
    return "<H1>0перации аренды для <EM>" + aCustomer.getNameO + "</EM></H1><P>\n";
 }
 public String value(Customer aCustomer) {
    Enumeration rentals = aCustomer.getRentals();
```

```
    String result = headerString(aCustomer)
      Rental each = (Rental) rentals.nextElement();
      // показать результаты по каждой аренде
      result += each.getMovie().getTitle()+
            String.valueOf(each.getCharge()) + "<BR>\n";
    }
    // добавить нижний колонтитул
    result += "<P>Ваша задолженность составляет <EM>" +
          String.valueOf (aCustomer.getTotalCharge()) +
          "</EM><P>\n";
    result += "На этой аренде вы заработали <EM>" +
          String.valueOf (aCustomer.getTotalFrequentRenterPoints()) +
          "</EM> очков за активность<P>";
    return result;
  }
}
```

Выполняю компиляцию и тестирование, а затем продолжаю работу с другими элементами. Я выполнил все шаги сразу. Вот 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 составляет" +
          String.valueOf (aCustomer.getTotalCharge()) + "\n" +
          "Вы заработали" +
          String.valueOf(aCustomer.getTotalFrequentRenterPoints()) +
          "очков за активность";
 }
}
class HtmlStatement {
 public String value(Customer aCustomer) {
    Enumeration rentals = aCustomer.getRentals();
    String result = headerStnng(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>Ваша задолженность составляет <EM>"+
            String.valueOf (aCustomer.getTotalCharge()) +
            "</EM><P>\n"+
            "На этой аренде вы заработали <EM>"+
            String.valueOf (aCustomer.getTotalFrequentRenterPoints()) +
            "</EM> очков за активность <P>";
    }
   }
```

Когда сделаны эти изменения, оба метода value выглядят очень схоже. Поэтому я применяю «Подъем метода» (Pull Up Method) к одному из них, выбирая версию текста произвольно. 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.nextElenent();
        result += eachRentalString(each);
      }
      result += footerString(aCustomer);
      return result;
    }
    abstract String headerString(Customer aCustomer);
    abstract String eachRentalString (Rental aRental);
    abstract String footerString (Customer aCustomer);
   }
```

Я удаляю метод value из TextStatement, компилирую и тестирую. Если все в порядке, я удаляю метод value из выписки HTML, компилирую и тестирую снова. Результат показан на рисунке 11.2.

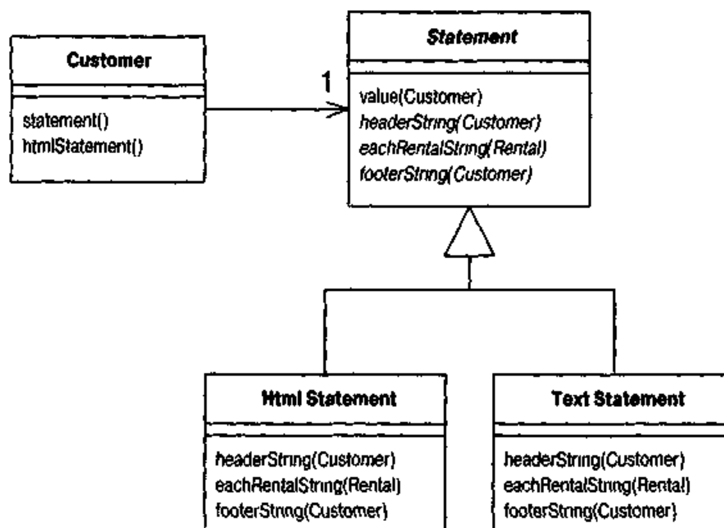


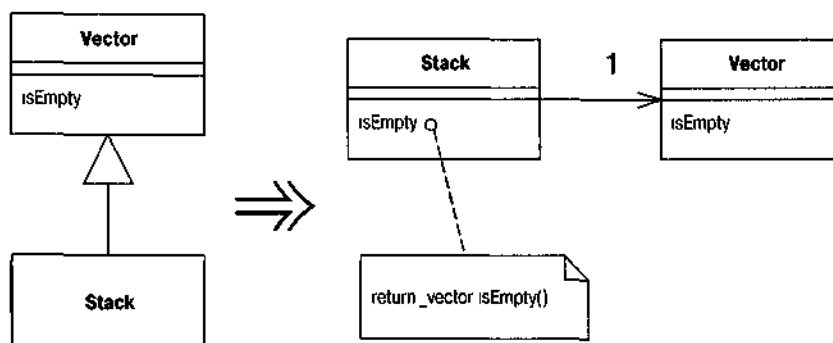
Рисунок 11.2 - Классы после формирования шаблона метода

После проведения этого рефакторинга легко добавлять новые виды утверждений. Для этого требуется лишь создать подкласс утверждения, замещающий три абстрактных метода.

Замена наследования делегированием (Replace Inheritance with Delegation)

Подкласс использует только часть интерфейса родительского класса или не желает наследовать данные.

Создайте поле для родительского класса, настройте методы, чтобы они делегировали выполнение родительскому классу, и удалите подклассы.



Мотивировка

Наследование - замечательная вещь, но иногда это не совсем то, что нам требуется. Часто, открывая наследование класса, мы затем обнаруживаем, что многие из операций родительского класса в действительности неприемлемы для подкласса. В этом случае имеющийся интерфейс неправильно отражает то, что делает класс. Либо оказывается, что целая группа наследуемых данных неприемлема для подкласса. Либо обнаруживается, что в родительском классе есть защищенные методы, не имеющие особого смысла в подклассе.

Можно оставить все как есть и условиться говорить, что хотя это подкласс, но в нем используется лишь часть функций родительского класса. Но это приводит к коду, который говорит нечто, отличное от ваших намерений, - беспорядок, подлежащий неукоснительной ликвидации.

Применяя вместо наследования делегирование, мы открыто заявляем, что используем делегируемый класс лишь частично. Мы управляем тем, какую часть интерфейса взять, а какую - игнорировать. Цена этому - дополнительные делегирующие методы, писать которые скучно, но так просто, что трудно ошибиться.

Техника

Создайте в подклассе поле, ссылающееся на экземпляр родительского класса. Инициализируйте его этим экземпляром.

Измените все методы, определенные в подклассе, чтобы они использовали поле делегирования. После изменения каждого метода выполните компиляцию и тестирование.

Вы не сможете заменить методы, вызывающие метод super, определенный в подклассе, иначе можно войти в бесконечную рекурсию. Эти методы можно заменить только после того, как будет разорвано наследование.

Удалите объявление родительского класса и замените присвоение делегирования присвоением новому объекту.

Для каждого метода родительского класса, используемого клиентом, добавьте простой делегирующий метод.

Выполните компиляцию и тестирование.

Пример

Одним из классических примеров плохого наследования является стек, оформленный как подкласс вектора. Java 1.1 поступает так в своих утилитах (ай-яй-яй!), но в данном случае 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 со стеком лишь четыре операции: push, pop, size и isEmpty. Последние две наследуются из Vector.

Я начинаю делегирование с поля для делегируемого вектора. Я связываю это поле с this, чтобы иметь возможность чередовать применение делегирования и наследования, пока выполняю этот рефакторинг:

```
private Vector _vector = this;
```

Теперь начинаю замену методов, чтобы заставить их использовать делегирование. Начинаю с push:

```
public void push(Object element) { _vector.insertElementAt(element, 0); }
```

Здесь можно выполнить компиляцию и тестирование, и все должно по-прежнему работать. Теперь займемся pop:

```
public Object pop() {
 Object result = _vector.firstElement();
 _vector.removeElementAt(0);
 return result;
}
```

Завершив эти методы подкласса, я должен разорвать связь с родительским классом:

```
class MyStack extends Vector {
 private Vector _vector = new Vector();
}
```

После этого добавляются простые делегирующие методы для методов родительского класса, используемых клиентами:

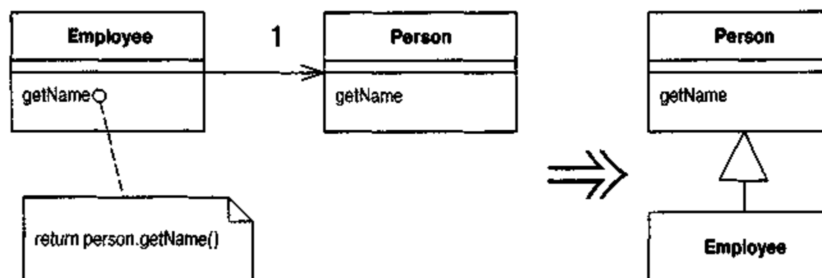
```
public int size() { return _vector.size(); }
public boolean isEmpty() { return _vector.isEmpty(); }
```

Теперь я могу выполнить компиляцию и тестирование. Если я забыл добавить делегирующий метод, то узнаю об этом при компиляции.

Замена делегирования наследованием (Replace Delegation with Inheritance)

Вы используете делегирование и часто пишете много простых делегирований для интерфейса в целом.

Сделайте делегирующий класс подклассом делегата.



Мотивировка

Это оборотная сторона «Замены наследования делегированием» (Replace Inheritance with Delegation). Если обнаруживается, что используются все методы делегируемого класса, и надоело писать эти простые методы делегирования, можно довольно просто вернуться назад к наследованию.

Есть некоторые опасности, о которых нужно помнить при этом. Если вы используете не все методы делегируемого класса, то не следует применять «Замену делегирования наследованием» (Replace Delegation with Inheritance), т. к. подкласс всегда должен следовать интерфейсу родительского класса. Если методы делегирования утомляют, можно воспользоваться другими вариантами. Можно разрешить клиентам обращаться к делегируемому классу самостоятельно, применив «Удаление посредника» (Remove Middle Man). Можно воспользоваться «Выделением родительского класса» (Extract Superclass), чтобы отделить общий интерфейс, а затем наследовать новому классу. Также можно применить «Выделение интерфейса» (Extract Interface).

Другая ситуация, которой необходимо остерегаться, возникает, когда делегат совместно используется несколькими объектами и может быть изменен. В таком случае нельзя заменить делегирование наследованием, потому что данные не смогут использоваться совместно. Совместное использование данных - это обязанность, которую нельзя перевести обратно в наследование. Когда объект неизменяем, совместное использование данных не вызывает проблем, потому что можно просто выполнить копирование, и никто ничего не заметит.

Техника

Сделайте делегирующий объект подклассом делегата.

Выполните компиляцию.

При этом может обнаружиться конфликт имен методов; у методов могут быть одинаковые имена, но существовать различия в типе возвращаемых значений, исключительных ситуациях и видимости. Внесите исправления с помощью «Переименования методов» (Rename Method).

Заставьте поле делегирования ссылаться на поле в самом объекте.

Удалите простые методы делегирования.

Выполните компиляцию и тестирование.

Замените все другие делегирования обращениями к самому объекту.

Удалите поле делегирования.

Пример

Простой класс Employee делегирует 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 getLastNameO {
return _name substring(_name lastlndex0f(      )+1), } }
```

Первым шагом просто объявляется подкласс:

```
class Employee extends Person
```

Компиляция в этот момент предупредит о возможных конфликтах методов. Они возникают, если методы с одинаковым именем возвращают значения различных типов или генерируют разные исключительные ситуации. Все проблемы такого рода исправляются с помощью «Переименования метода» (Rename Method). В данном простом примере таких затруднений не возникает.

Следующим шагом заставляем поле делегирования ссылаться на сам объект. Я должен удалить все простые методы делегирования, такие как getName и setName. Если их оставить, возникнет ошибка переполнения стека, обусловленного бесконечной рекурсией. В данном случае надо удалить getName и setName из Employee.

Создав работающий класс, можно изменить те методы, в которых применяются методы делегирования. Я перевожу их на непосредственное использование вызовов:

```
public String toString () { return “Emp” + getLastName(); }
```

Избавившись от всех методов, использующих методы делегирования, я могу освободиться от поля _person.

12 КРУПНЫЕ РЕФАКТОРИНГИ

Кент Бек и Мартин Фаулер

В предшествующих главах были показаны отдельные «ходы» рефакторинга. Чего не хватает, так это представления об «игре» в целом. Рефакторинг проводится с какой-то целью, а не просто для того, чтобы приостановить разработку (по крайней мере, обычно рефакторинг проводится с некоторой целью). Так как же выглядит игра в целом?

Правила игры

На что вы, несомненно, обратите внимание в последующем изложении, так это на далеко не такое тщательное описание шагов, как в предыдущих рефакторингах. Это связано с тем, что при проведении крупных рефакторингов ситуация существенно изменяется. Мы не можем точно сказать вам, что надо делать, потому что не знаем точно, что будет у вас перед глазами, когда вы будете это делать. Когда вы вводите в метод новый параметр, то механизм ясен, потому что ясна область действия. Когда же вы разбираетесь с запутанным наследованием, то в каждом случае беспорядок особенный.

Кроме того, надо представлять себе, что описываемые здесь рефакторинги отнимают много времени. Любой рефакторинг из глав 6-11 можно выполнить за несколько минут, в крайнем случае, за час. Над некоторыми же крупными рефакторингами мы работали в течение месяцев или лет, причем в действующих системах. Когда есть функционирующая система и надо расширить ее функциональность, убедить руководство, что надо остановиться на пару месяцев, пока вы будете приводить в порядок код, довольно тяжело. Вместо этого вы должны поступать, как Ганзель и Гретель, откусывая по краям чуть-чуть сегодня, еще чуть-чуть завтра.

При этом вами должна руководить потребность в каких-то дополнительных действиях. Выполняйте рефакторинги, когда требуется добавить функцию и исправить ошибки. Начав рефакторинг, вам не обязательно доводить его до конца. Делайте столько, сколько необходимо, чтобы выполнить реально стоящую задачу. Всегда можно продолжить на следующий день.

Данная философия отражена в примерах. Демонстрация любого рефакторинга из этой книги легко могла бы занять сотню страниц. Нам об этом известно, потому что Мартин попытался сделать это. Поэтому мы сжали примеры до нескольких схематичных диаграмм.

Из-за того что они отнимают так много времени, крупные рефакторинги не приносят мгновенного вознаграждения, как рефакторинги, описанные в других главах. Придется лишь довольствоваться верой, что с каждым днем вы делаете жизнь для своей программы немного безопаснее.

Крупные рефакторинги требуют определенного согласия во всей команде программистов, чего не надо для маленьких рефакторингов. Крупные рефакторинги определяют направление для многих и многих модификаций. Вся команда должна осознать, что «в игре» находится один из крупных рефакторингов, и действовать соответственно. Нежелательно оказаться в положении тех двух парней, машина которых застряла около вершины горы. Они вылезают, чтобы толкать машину, каждый со своего конца. После получаса безуспешных попыток сдвинуть ее тот, который впереди, говорит: «Никогда не думал, что столкнуть машину вниз с горы так тяжело». На что второй отвечает: «Что ты имеешь в виду под «вниз с горы»?»

Почему важны крупные рефакторинги

Если в крупных рефакторингах нет многих из тех качеств, которые составляют ценность маленьких (предсказуемости, заметного прогресса, немедленного вознаграждения), почему они так важны, что мы решили включить их в эту книгу? Потому что без них вы рискуете потратить время и силы на обучение проведению рефакторинга, а затем и на практический рефакторинг и не получить никаких выгод. Нам это было бы неприятно. Мы такого потерпеть не можем.

Если серьезно, то рефакторингом занимаются не потому, что это весело, а потому, что вы рассчитываете сделать со своими программами после проведения рефакторинга то, чего без него вы просто не смогли бы сделать.

Накопление малопонятных проектных решений в итоге просто удушает программу подобно водорослям, которые душат канал. Благодаря рефакторингу вы гарантируете, что полное понимание вами того, как должна быть спроектирована программа, всегда находит в ней отражение. Как сорные водоросли проворно распространяют свои усики, так и не до конца понятые проектные решения быстро распространяют свое действие по всей программе. Ни какого-либо одного, ни даже десяти отдельных действий не будет достаточно, чтобы искоренить проблему.

Четыре крупных рефакторинга

В этой главе мы описываем четыре примера крупных рефакторингов. Это примеры, показывающие, о какого рода вещах идет речь, а не попытки охватить необъятное. Исследования и практика проведения

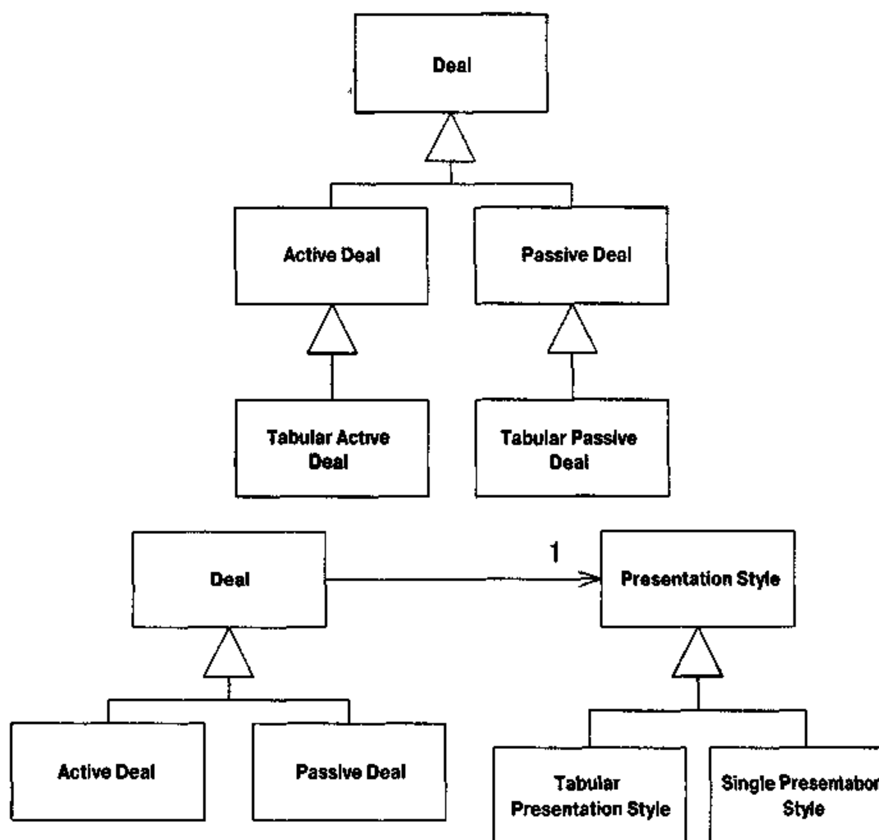
рефакторинга до сих пор концентрировались на небольших рефакторингах. Предстоящий разговор о крупных рефакторингах весьма необычен и основан преимущественно на опыте Кента, у которого в крупных проектах его больше, чем у кого-либо другого.

«Разделение наследования» (Tease Apart Inheritance) имеет дело с запутанной иерархией наследования, в которой различные варианты представляются объединенными вместе так, что это сбивает с толку. «Преобразование процедурного проекта в объекты» (Convert Procedural Design to Object) содействует решению классической проблемы преобразования процедурного кода. Множество программистов пользуется объектно-ориентированными языками, не имея действительного представления об объектах, поэтому данный вид рефакторинга приходится часто выполнять. Если вы увидите код, написанный с классическим двухзвенным подходом к интерфейсам пользователя и базам данных, то вам понадобится «Отделение предметной области от представления» (Separate Domain from Presentation), чтобы разделить бизнес-логику и код интерфейса пользователя. Опытные объектно-ориентированные разработчики поняли, что такое разделение жизненно важно для долго живущих и благополучных систем. «Выделение иерархии» (Extract Hierarchy) упрощает чрезмерно сложные классы путем превращения их в группы подклассов.

Разделение наследования (Tease Apart Inheritance)

Есть иерархия наследования, которая выполняет одновременно две задачи.

Создайте две иерархии и используйте делегирование для вызова одной из другой.



Мотивировка

Наследование - великая вещь. Оно способствует написанию чрезвычайно «сжатого» кода в подклассах. Отдельные методы могут приобретать непропорциональное своим размерам значение благодаря своему местоположению в иерархии.

Механизм наследования обладает такой мощью, что случаи его неправильного применения не должны удивлять. А неправильное употребление может накапливаться постепенно. Вы вводите небольшой класс для решения небольшой задачи. На следующий день добавляете другие подклассы для решения той же задачи в других местах иерархии. И через неделю (или месяц, или год) уже плаваете в «макаронном коде». Без весел.

Запутанное наследование представляет собой потенциальный источник неприятностей, потому что оно приводит к дублированию кода - бичу программистов. Оно осложняет модификацию, потому что стратегии решения определенного рода проблемы распространены по всей программе. Наконец, труднее понимать результирующий код. Нельзя просто сказать: «Вот эта иерархия вычисляет результаты». Приходится говорить:

«Да, она вычисляет результаты, а вот там находятся подклассы для табличных версий, и у каждого из них есть подклассы для каждой из стран».

Легко 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 делаете два дела с помощью одной иерархии.

Техника

Выделите различные задачи, выполняемые иерархией. Создайте двумерную сетку (или трехмерную, или четырехмерную, если у вас есть такая замечательная миллиметровка) и пометьте оси разными задачами. Предполагается, что более чем для двух измерений данный рефакторинг надо применять многократно (конечно, по одному за раз).

Определите, какая задача важнее и должна быть сохранена в текущей иерархии, а какую надо переместить в другую иерархию.

Примените «Выделение класса» (Extract Class) в общем родительском классе, чтобы создать объект для вспомогательной задачи, и добавьте переменную экземпляра, хранящую этот объект.

Создайте подклассы выделенного объекта для каждого из подклассов исходной иерархии. Инициализируйте переменную экземпляра, созданную на предыдущем шаге, экземпляром этого подкласса.

Примените «Перемещение метода» (Move Method) в каждом из этих подклассов для переноса поведения из подкласса в выделенный объект.

Когда в подклассе не останется больше кода, удалите его.

Продолжайте, пока не исчезнут все подклассы. Посмотрите, нельзя ли применить к новой иерархии дополнительные рефакторинги, например «Подъем метода» (Pull Up Method) или «Подъем поля» (Pull Up Field).

Примеры

Рассмотрим пример запутанной иерархии (рисунок 12.1).

Эта иерархия стала такой потому, что класс Deal первоначально использовался для отображения только одной сделки. Затем кому-то пришла в голову мысль отображать таблицу сделок. Немного поэкспериментировав со сделанным на скорую руку классом ActiveDeal, можно убедиться, что без особого труда можно действительно отображать таблицу. Что, таблица пассивных сделок тоже нужна? Нет проблем, еще один небольшой подкласс, и все готово.

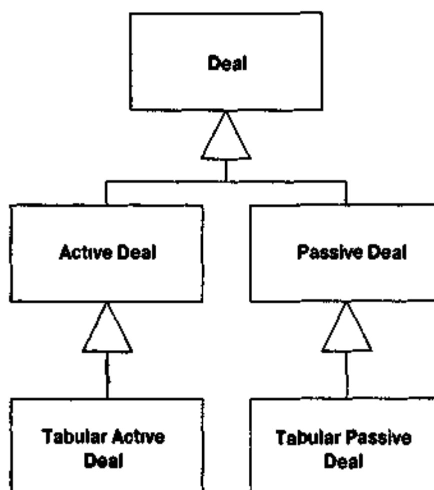


Рисунок 12.1 - Запутанная иерархия

Через два месяца код таблицы усложнился, но найти для него место непросто, время поджимает - обычная история. Добавить новый вид сделки теперь трудно, потому что логика сделки перепутана с логикой представления.

Согласно нашему рецепту, первым делом надо определить задачи, которые выполняет иерархия. Одна задача состоит в фиксации отклонений, соответствующих типу сделки. Другая задача - фиксировать отклонения, соответствующие стилю представления. Поэтому сетка будет такая:

Deal	Active Deal	Passive Deal
Tabular Deal		

Следующим шагом мы должны решить, какая задача важнее. Связь объекта со сделкой гораздо важнее стиля представления, поэтому мы оставим в покое Deal и выделим стиль представления в собственную иерархию. На практике, вероятно, следует оставлять на месте ту задачу, с которой связан больший объем кода, чтобы осуществлять меньше перемещений.

Затем мы должны применить «Выделение класса» (Extract Class) для создания стиля представления (рисунок 12.2).

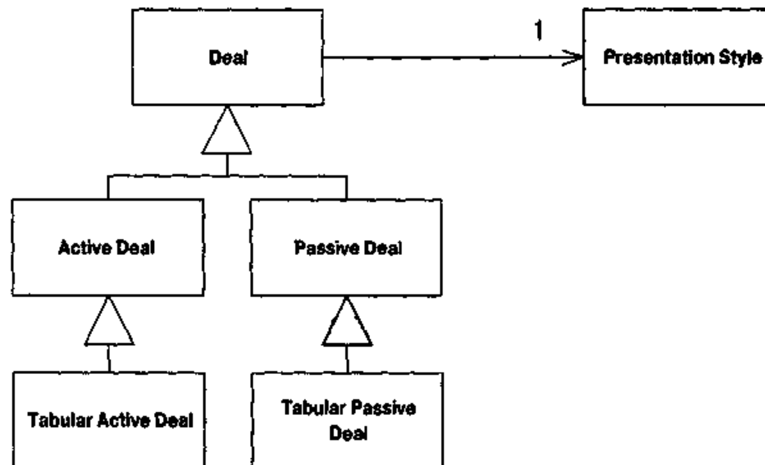


Рисунок 12.2 - Добавление стиля представления

Следующее, что мы должны сделать, это создать подклассы выделенного класса для каждого из подклассов исходной иерархии (рисунок 12.3) и инициализировать переменную экземпляра соответствующим подклассом:

```
ActiveDeal constructor
 ...presentation = new SingleActivePresentationStyle();
```

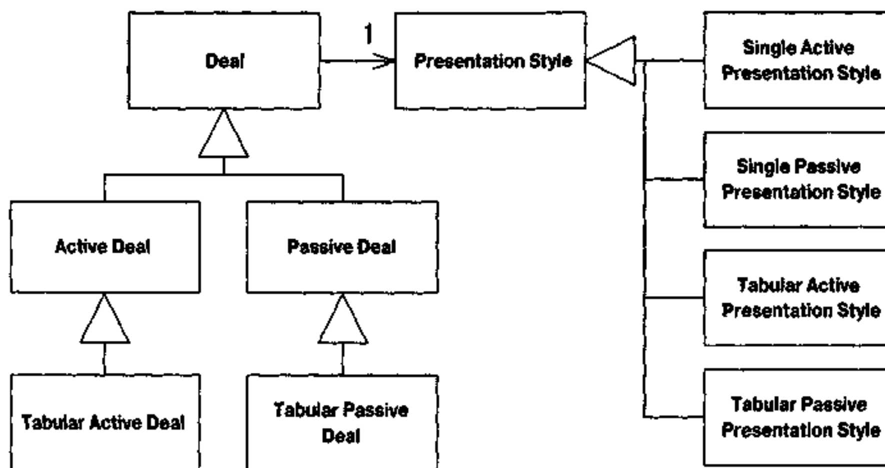


Рисунок 12.3 - Добавление подклассов стиля представления

Возможно, вы скажете: «А не стало ли у нас теперь больше классов, чем было? Разве это облегчит нам жизнь?» Что ж, иногда надо сделать шаг назад, чтобы сделать два шага вперед. В таких случаях запутанной иерархии, как данный, иерархия выделенного объекта почти всегда может быть значительно упрощена после того, как объект выделен. Однако более надежно выполнять рефакторинг, двигаясь пошагово, чем перескочить сразу на десять шагов вперед к уже упрощенной конструкции.

Теперь мы применим «Перемещение метода» (Move Method) и «Перемещение поля» (Move Field), чтобы перенести относящиеся к представлению методы и переменные подклассов сделки в подклассы стиля представления. Мы не сможем хорошо изобразить это на том примере, который приведен, поэтому представьте себе, как это происходит, в своем воображении. Во всяком случае, в результате в классах TabularActiveDeal и TabularPassiveDeal кода остаться не должно, поэтому мы их удаляем (рисунок 12.4).

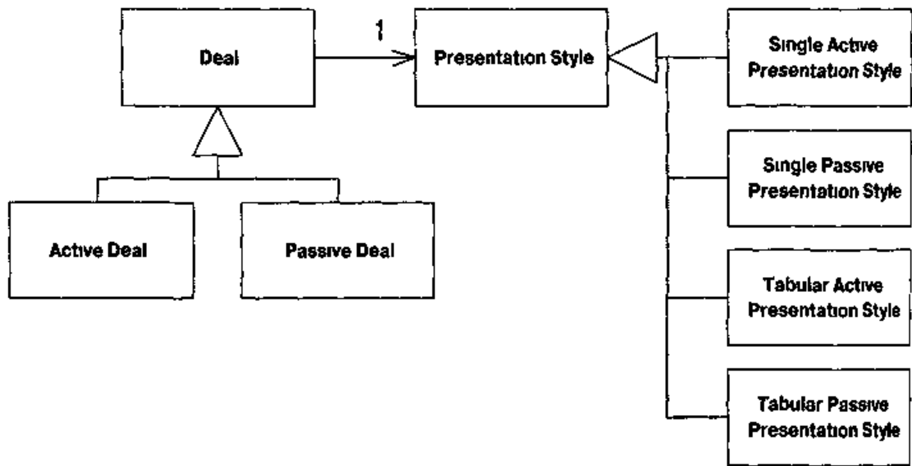


Рисунок 12.4 - Табличные подклассы Deal удалены

После разделения двух задач можно отдельно работать над упрощением каждой из них. Когда этот рефакторинг завершается, всегда появляется возможность значительно упростить выделенный класс, а часто и еще больше упростить исходный объект. Следующим шагом мы освободимся от разделения на «активный-пассивный» в стиле представления рисунка, как показано на рисунке 12.5.

Даже различие между одиночным и табличным представлениями может быть зафиксировано в значениях нескольких переменных. Подклассы становятся вообще ненужными (рисунок 12.6).

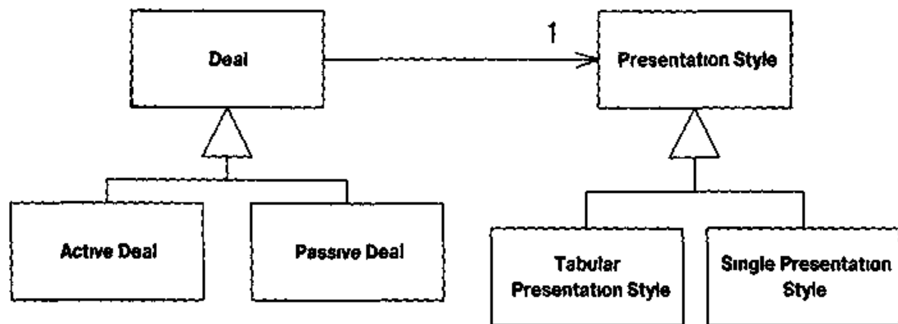


Рисунок 12.5 - Теперь иерархии разделены

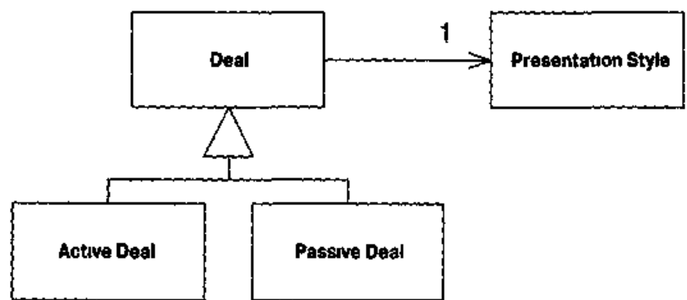
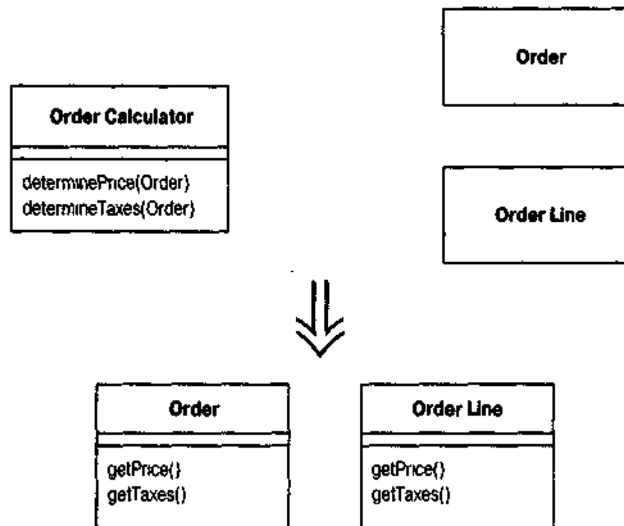


Рисунок 12.6 - Различия в представлении могут устанавливаться парой переменных

Преобразование процедурного проекта в объекты (Convert Procedural Design to Objects)

Имеется код, написанный в процедурном стиле.

Преобразуйте записи данных в объекты, разделите поведение и переместите поведение в объекты.



Мотивировка

Один наш клиент при запуске нового проекта потребовал, чтобы разработчики неукоснительно придерживались двух принципов: (1) использовали Java и (2) не использовали объекты.

Можно посмеяться, но хотя Java является объектно-ориентированным языком, применение объектов не ограничивается вызовом конструктора. Правильному применению объектов надо учиться. Часто приходится сталкиваться с проблемой, когда код процедурного вида надо сделать более объектно-ориентированным. Типичную ситуацию представляют длинные процедурные методы в классе, хранящем мало данных, и «немые» объекты данных, в которых нет ничего, кроме методов доступа к полям. Если надо преобразовать чисто процедурную программу, то и «немых» объектов может не оказаться, но для начала и это неплохо.

Мы не утверждаем, что объектов с поведением и малым количеством данных (или полным их отсутствием) вообще быть не должно. Мы часто используем маленькие объекты стратегий, когда требуется варьировать поведение. Однако такие процедурные объекты обычно невелики и применяются, когда возникает особая потребность в гибкости.

Техника

Преобразуйте все типы записей в «немые» объекты данных с методами доступа.

При работе с реляционной базой данных преобразуйте каждую таблицу в немой объект данных.

Поместите весь процедурный код в один класс.

Можно сделать класс единичным экземпляром (для облегчения реинициализации) либо объявить методы статическими.

К каждой длинной процедуре примените «Выделение метода» (Extract Method) и сопутствующие рефакторинги, чтобы разбить ее на части. Разложив процедуры на части, примените «Перемещение метода» (Move Method) для переноса их в надлежащий «немой» класс данных.

Продолжайте, пока из первоначального класса не будет удалено все поведение. Если исходный класс был чисто процедурным, можно с большим удовольствием удалить его.

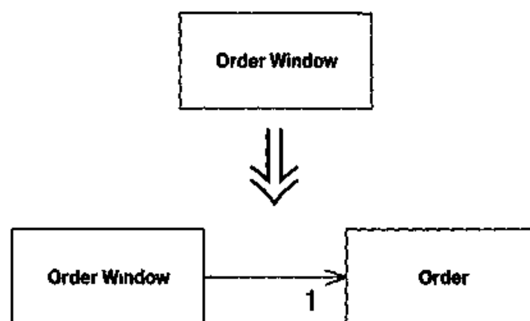
Пример

Пример, приведенный в главе 1, хорошо иллюстрирует необходимость «Преобразования процедурного проекта в объекты» (Convert Procedural Design to Objects), особенно на первом этапе, когда разлагается на части и распределяется по классам метод statement. По завершении можно применять к ставшим интеллектуальными объектам данных другие рефакторинги.

Отделение предметной области от представления (Separate Domain from Presentation)

Имеются классы GUI, содержащие логику предметной области.

Выделите логику предметной области в отдельные классы предметной области



Мотивировка

В разговорах об объектах можно услышать о «модели-представлении-контроллере» (MVC). Эта идея послужила фундаментом связи между графическим интерфейсом пользователя (GUI) и объектами предметной области в Smalltalk-80.

В основе 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 кода визуализации, зато содержат всю бизнес-логику. В результате сложная программа разделяется на части, которые проще модифицировать. Кроме того, появляется возможность существования нескольких представлений одной и той же бизнес-логики. Тот, кто имеет опыт работы с объектами, пользуется таким разделением инстинктивно, и оно доказало свою ценность.

Но многие из тех, кто работает с GUI, не придерживаются такой архитектуры. В большинстве сред с GUI «клиент/сервер» реализована двухзвенная конструкция: данные находятся в базе данных, а логика находится в классах представления. Среда часто навязывает такой стиль проектирования, мешая поместить логику в другое место.

Java - это правильная объектно-ориентированная среда, позволяющая создавать невизуальные объекты предметной области, содержащие бизнес-логику. Однако часто можно столкнуться с кодом, написанным в двухзвенном стиле.

Техника

Создайте для каждого окна класс предметной области.

Если есть сетка, создайте класс, представляющий строки этой сетки. Используйте для окна коллекцию из класса предметной области, которая будет хранить объекты предметной области из строки.

Изучите данные в окне. Если они применяются только для целей интерфейса пользователя, оставьте их в окне. Если они задействованы в логике предметной области, но фактически не отображаются в окне, примените «Перемещение метода» (Move Method), чтобы перенести их в объект предметной области. Если же они фигурируют как в интерфейсе пользователя, так и в логике предметной области, примените «Дублирование видимых данных» (Duplicate Observed Data), чтобы они находились в обоих местах и были синхронизированы.

Изучите логику в классе представления. Воспользуйтесь «Выделением метода» (Extract Method) для отделения логики, относящейся к представлению, от логики предметной области. Отделив логику предметной области, примените «Перемещение метода» (Move Method) для перенесения ее в объект предметной области.

По окончании вы будете располагать классами представления, обрабатывающими GUI, и объектами предметной области, содержащими всю бизнес-логику. Объекты предметной области будут недостаточно структурированы, но с этим справятся дальнейшие рефакторинги.

Пример

Есть программа, дающая пользователям возможность вводить информацию о заказах и получать их стоимость. Ее GUI показан на рисунке 12.7. Класс представления взаимодействует с реляционной базой данных, структура которой показана на рисунке 12.8.

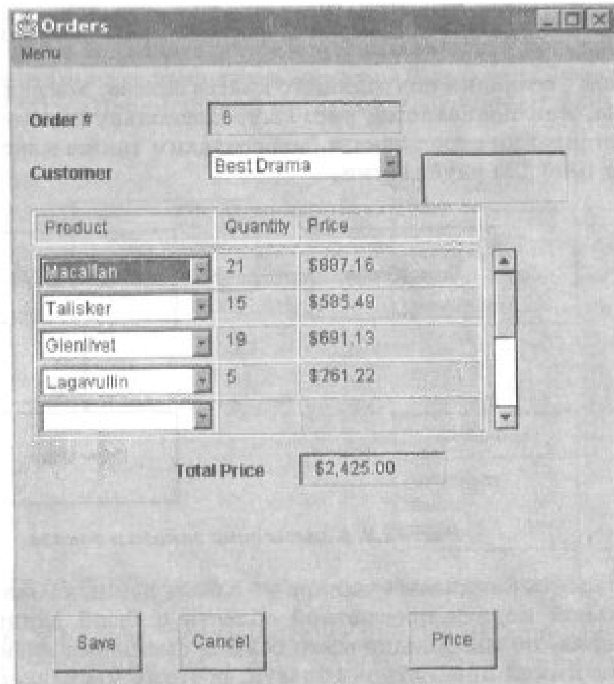


Рисунок 12.7 - Интерфейс пользователя в начальной программе

Все поведение, для GUI и для определения стоимости заказа, находится в одном классе. OrderWindow.

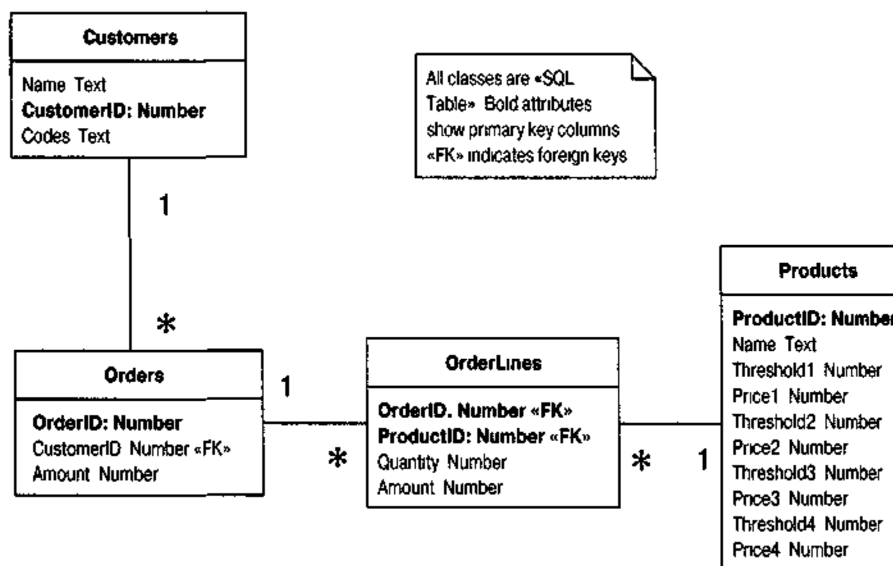


Рисунок 12.8 - База данных для программы ввода заказов

Начнем с создания подходящего класса заказа. Мы свяжем его с окном заказа, как показано на рисунке 12.9. Поскольку в окне есть сетка для представления строк заказа, мы создадим также класс строки заказа (order line) для рядов сетки.

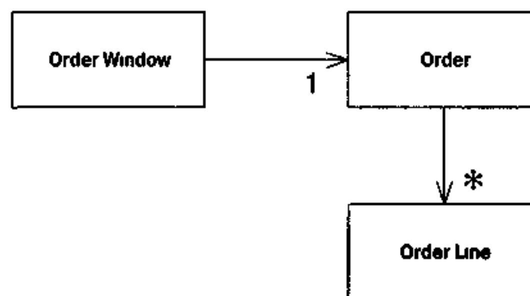


Рисунок 12.9 - Классы окна заказа и заказа

Действовать начинаем с окна, а не с базы данных. Связывание первоначальной модели предметной области с базой данных - разумная стратегия, но мы больше всего боимся смешения логики представления с

логикой предметной области, поэтому мы разделяем их, идя от окна, а остальное подвергнем рефакторингу позднее.

В таких программах полезно посмотреть на SQL-запросы, имеющиеся в окне. Данные, получаемые из запросов SQL, относятся к предметной области.

Проще всего разобраться с данными предметной области, не отображаемыми в GUI непосредственно. В данном примере в базе данных есть поле codes, находящееся в таблице customers. Код не отображается непосредственно в GUI; он преобразуется в более удобочитаемый текст. По существу, это поле является простым классом, например строкой, а не компонентом AWT. Можно благополучно применить «Перемещение поля» (Move Field), чтобы перенести поле в класс предметной области.

С другими полями нам не так повезло. Они содержат компоненты AWT, которые отображаются в окне и используются в объектах предметной области. К ним надо применить «Дублирование видимых данных» (Duplicate Observed Data). В результате поле предметной области помещается в класс order, а соответствующее поле AWT помещается в окно заказа.

Это медленный процесс, но в итоге можно поместить все поля логики предметной области в класс предметной области. Удобно вести его, если попробовать переместить все запросы SQL в класс предметной области. В результате можно переместить логику базы данных и данные предметной области в класс предметной области. Приятное чувство законченности возникает после удаления импорта java.sql из окна заказа. Для этого надо выполнить немало «Выделений метода» (Extract Method) и «Перемещений метода» (Move Method).

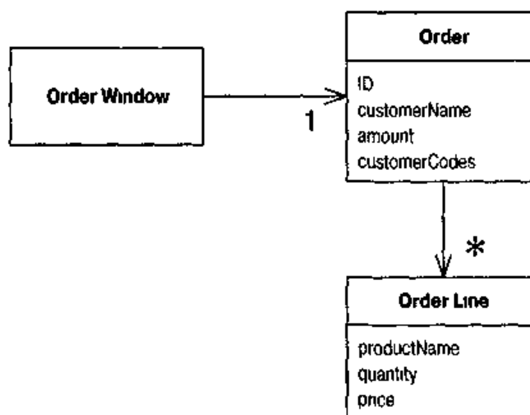


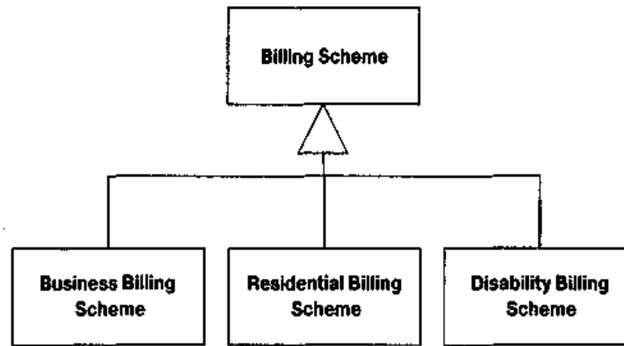
Рисунок 12.10 - Распределение данных по классам предметной области

Полученные классы, показанные на рисунке 12.10, еще далеки от хорошей структуризации, но в этой модели логика предметной области уже отделена. При проведении этого рефакторинга надо внимательно следить за возможными опасностями. Если наибольшую угрозу представляет смешение логики представления и логики предметной области, полностью разделите их, прежде чем делать еще что-либо существенное. Если важнее другие вещи, например стратегии получения цен продуктов, уберите из окна логику этой важной части и занимайтесь ее реорганизацией, чтобы получить приемлемую структуру для области высокого риска. Весьма вероятно, что из окна заказа надо будет убрать всю логику предметной области. Если можно изменить композицию классов программы, оставив часть логики в окне, так и сделайте, чтобы сначала решить проблему наибольшей опасности.

Выделение иерархии (Extract Hierarchy)

Есть класс, выполняющий слишком много работы, частично или полностью через многочисленные условные операторы.

Создайте иерархию классов, в которой каждый подкласс представляет конкретный случай.



Мотивировка

При эволюционном проектировании класс часто представляется как реализация одной идеи, а впоследствии обнаруживается, что на самом деле он реализует две или три идеи, а то и все десять. Сначала создается 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 случае. Через месяц обнаруживается еще одна такая возможность. Через год наступает полная неразбериха: всюду раскиданы флаги и условные выражения.

Столкнувшись с таким «швейцарским армейским ножиком», который умеет открывать консервные банки, подрезать деревья, лениво высвечивать лазером пункты на презентациях, - да, надо надеяться, и резать что-то, надо найти стратегию, позволяющую расплести отдельные пряди. Описываемая здесь стратегия действует, только если условная логика сохраняет статичность в течение срока жизни объекта, в противном случае, прежде чем начать отделять одни случаи от других, может потребоваться провести «Выделение класса» (Extract Class).

Пусть вас не пугает, что «Выделение иерархии» (Extract Hierarchy) - такой рефакторинг, который нельзя завершить за один день. Для распутывания сложного проекта могут потребоваться недели или месяцы. Сделайте то, что просто и очевидно, а потом прервитесь. Займитесь на несколько дней работой, приносящей видимый результат. Узнав что-то, вернитесь и выполните еще несколько простых и очевидных действий.

Техника

Мы поместили здесь два набора технических правил. В первом случае вы не знаете, какие варианты возникнут. Тогда надо действовать пошагово, как описано ниже:

Определите один из вариантов.

Если варианты могут меняться в течение срока жизни объекта, примените «Выделение класса» (Extract Class) для оформления этого аспекта в виде отдельного класса.

Создайте подкласс для этого особого случая и примените к оригиналу «Замену конструктора фабричным методом» (Replace Constructor with Factory Method). Модифицируйте фабричный метод так, чтобы он возвращал экземпляр подкласса, если это уместно.

Один за другим скопируйте в подкласс методы, содержащие условную логику, затем упростите методы, исходя из того, что можно определенно сказать об экземплярах подкласса, чего нельзя сказать об экземплярах родительского класса.

При необходимости применяйте к родительскому классу «Выделение метода» (Extract Method), чтобы разделить условные и безусловные части методов.

Продолжайте отделение особых случаев, пока не станет возможным объявить родительский класс как абстрактный.

Удалите тела методов родительского класса, перегруженные во всех подклассах, и сделайте соответствующие объявления в родительском классе абстрактными.

Если варианты совершенно ясны с самого начала, можно воспользоваться другой стратегией, например:

Создайте подкласс для каждого варианта.

Примените «Замену конструктора фабричным методом» (Replace Constructor with Factory Method), который должен возвращать соответствующий класс для каждого варианта.

Если варианты помечены кодом типа, примените «Замену кода типа подклассами» (Replace Type Code with Subclasses). Если варианты могут меняться в течение срока существования класса, примените «Замену кода типа состоянием/стратегией» (Replace Type Code with State/Strategy).

К методам, содержащим условную логику, примените «Замену условных операторов полиморфизмом» (Replace Conditional with Polymorphism). Если не изменяется весь метод, отделите изменяющуюся часть с помощью «Выделения метода» (Extract Method).

Пример

Пример представляет собой неочевидный случай. Чтобы посмотреть, как действовать в очевидном случае, проследите проведение рефакторингов для «Замены кода типа подклассами» (Replace Type Code with Subclasses), «Замены кода типа состоянием/стратегией» (Replace Type Code with State/Strategy) и «Замены условных операторов полиморфизмом» (Replace Conditional with Polymorphism).

Начнем с программы, которая выписывает счета за электричество. Исходные объекты показаны на рисунке 12.11.

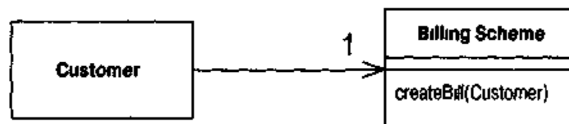


Рисунок 12.11 - Схема выписки счетов клиентам

Схема выписки счетов содержит массу условной логики для действий в различных обстоятельствах. Зимой и летом применяются разные тарифы, различные схемы начисления используются для жилых домов, малых предприятий, получающих социальную страховку, и инвалидов. Итоговая логика делает класс BillingScheme достаточно сложным.

На первом шаге мы выберем аспект варианта, который постоянно всплывает в условной логике. Им могут оказаться различные условия, зависящие от того, распространяется ли на клиента схема оплаты для инвалидов. Это может быть флаг в Customer, BillingScheme или где-либо еще.

Создадим подкласс варианта. Чтобы пользоваться этим подклассом, надо обеспечить его создание и применение. Смотрим на конструктор для BillingScheme. Сначала применяем «Замену конструктора фабричным методом» (Replace Constructor with Factory Method). Потом смотрим на полученный фабричный метод и выясняем, как логика зависит от инвалидности. После этого создаем блок операторов, который возвращает, когда надо, схему начисления оплаты для инвалидов.

Рассматриваем разные методы в BillingScheme и ищем те, в которых есть условная логика, зависящая от инвалидности. Одним из таких методов будет CreateBill, поэтому копируем его в подкласс (рисунок 12.12).

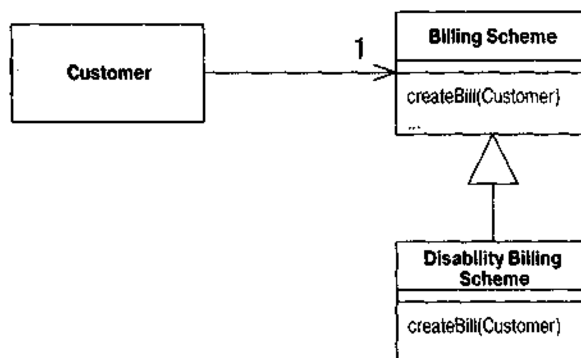


Рисунок 12.12 - Добавление подкласса для инвалидов

Теперь изучаем экземпляр createBill, находящийся в подклассе, и упрощаем его, поскольку знаем, что теперь он находится в контексте схемы начисления оплаты для инвалидов. Поэтому код вида

```
if (disabilityScheme()) doSomething;
```

можно заменить на

```
doSomething();
```

Если инвалидность исключается из схемы для предприятий, можно удалить условно выполняемый код в схеме для предприятий.

При этом мы хотим обеспечить отделение изменяющегося кода от остающегося неизменным. Для этого мы применяем «Выделение метода» (Extract Method) и «Декомпозицию условного оператора» (Decompose Conditional). Делаем это с различными методами BillingScheme, пока не почувствуем, что справились с большей частью условных операторов по инвалидности. Потом выбираем другой вариант, например социальное страхование, и то же самое проделываем с ним. При этом, занимаясь 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 могут быть свои варианты начисления налогов. Мы хотим обеспечить присутствие в двух подклассах методов с одинаковой сигнатурой. При этом может понадобиться изменить класс для инвалидов, чтобы привести в соответствие два подкласса. Обычно обнаруживается, что с разработкой новых вариантов система аналогичных и различающихся методов стабилизируется, что облегчает работу с новыми вариантами.

13 РЕФАКТОРИНГ, ПОВТОРНОЕ ИСПОЛЬЗОВАНИЕ И РЕАЛЬНОСТЬ

Уильям Апдайк

С Мартином Фаулером я познакомился в Ванкувере во время OOPSLA 92. За несколько месяцев до того я завершил свою диссертацию по рефакторингу объектно-ориентированных сред ([1]) в Университете штата Иллинойс. Изучая возможность продолжения своих исследований по рефакторингу, я рассматривал и другие варианты, например в медицинской информатике. Мартин в то время работал над приложением в медицинской информатике, что дало нам тему для разговора за завтраком в Ванкувере. Как рассказывает Мартин в начале книги, мы в течение нескольких минут обсуждали мои исследования по рефакторингу. В тот период его интерес к данной теме был невелик, но с тех пор, как вы теперь знаете, он значительно вырос.

На первый взгляд, рефакторинг возник в академических исследовательских лабораториях. На самом деле он появился в практике разработки программного обеспечения, когда программисты, применяющие объектно-ориентированный подход, использовавшие тогда Smalltalk, столкнулись с потребностью в технологиях, обеспечивающих лучшую поддержку процесса разработки в среде, или, в общем случае, поддержку процесса модификации. Это породило исследования, доведенные до той точки, в которой мы почувствовали, что они «готовы для прайм-тайма», т. е. когда широкий круг профессиональных программистов может воспользоваться выгодами рефакторинга.

Когда Мартин предложил мне написать главу для этой книги, у меня возникло несколько мыслей. Я мог бы описать ранние исследования рефакторинга - ту эпоху, когда Ральф Джонсон и я, получив весьма разную техническую подготовку, объединились, чтобы заняться поддержкой процесса модификации объектно-ориентированного программного обеспечения. Я мог бы описать автоматизацию поддержки рефакторинга -- область моих исследований, весьма отличную от задач этой книги. Можно было бы поделиться некоторыми полученными мною уроками, касающимися отношения между рефакторингом и повседневными заботами профессиональных разработчиков программного обеспечения, особенно работающих в промышленности над большими проектами. Понимание, достигнутое мной во время исследования рефакторинга, часто оказывалось полезным в широком диапазоне областей - при оценке программных технологий и формулировании стратегий развития продуктов, в разработке прототипов и продуктов в сфере телекоммуникаций и при обучении и консультировании групп разработчиков проектов.

Я решил кратко остановиться на целом ряде вопросов, имеющих отношение к этим областям. Как следует из названия главы, глубокое понимание рефакторинга часто имеет более широкое применение в таких проблемах, 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 уделено в ней практическим заботам 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

Я проработал несколько лет в BellLabs, перед тем как заняться своими исследованиями. Большую часть этого времени я был занят в том подразделении компании, которое разрабатывало электронные системы коммутации. К таким продуктам предъявляются очень высокие требования как в отношении надежности, так и в отношении скорости обработки телефонных звонков. Тысячи человеко-лет были вложены в разработку и развитие таких систем. Срок жизни продуктов достигал десятилетий. Большая часть стоимости разработки этих систем обусловлена не разработкой первоначального изделия, а постоянной адаптацией и модификацией систем. Средства упрощения и удешевления таких модификаций принесли бы компании большую прибыль.

Поскольку BellLabs финансировала мои диссертационные исследования, 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 предложил тему исследований, которая охватывала как объектно-ориентированную технологию, так и поддержку процесса модификации и эволюционирования программного обеспечения, я сразу за нее ухватился.

Мне говорили, что когда люди заканчивают свои диссертации, то редко остаются безучастными к выбранной ими теме. Одни, устав от темы, быстро переключаются на что-то другое. Другие сохраняют в отношении нее энтузиазм. Я оказался в последнем лагере.

Когда я вернулся в BellLabs после получения ученой степени, произошла странная вещь. Тех, с кем я работал, рефакторинг волновал в гораздо меньше, чем меня.

Я отчетливо 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 беседа развивалась успешно. Мой энтузиазм в отношении темы передался

слушателям. Но в конце беседы было 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 надеялся, что кто-нибудь из участников разрабатываемых проектов проявит желание применить рефакторинг в своей работе. Если у них и возникло желание, в тот момент они его не продемонстрировали.

Похоже, что люди просто не разобрались, в чем дело.

Ральф Джонсон преподал мне важный урок, касающийся исследований: когда кто-то (рецензент статьи, участник конференции) заявляет, что он не понимает, или просто «не врубается», это наша вина. Наша обязанность постараться развить и донести свои идеи.

В течение последующей пары лет у меня были многочисленные возможности поговорить о рефакторинге на внутренних совещаниях в AT&T BellLabs, на конференциях и семинарах в других местах. По ходу своих бесед с практическими разработчиками я начал понимать, почему мои прежние обращения к слушателям не были ими поняты. Отсутствие контакта было частично вызвано новизной объектно-ориентированной технологии. Немногие из тех, кто работал с ней, продвинулись дальше первоначального релиза и потому еще не столкнулись с трудными проблемами развития продукта, которые помогает решить рефакторинг. Это типичная дилемма исследователя, когда современное состояние науки опережает современное состояние практики. Однако была еще одна тревожная причина отсутствия контакта. По ряду причин разработчики, основываясь на здравом смысле, не желали применять рефакторинг к своим программам, даже признавая выгоды, которые он сулит. Чтобы рефакторинг был принят сообществом разработчиков, необходимо было решить эти проблемы.

Почему разработчики не хотят применять рефакторинг к своим программам?

Допустим, что вы - разработчик программного обеспечения. Если ваш проект начинается с нуля (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. Такой сценарий идеален для применения объектно-ориентированной технологии, но большинство из нас может о нем только мечтать.

Значительно чаще предлагается расширить возможности уже имеющегося программного обеспечения. У вас далеко не полное понимание того, что вы делаете. Вы поставлены в жесткие временные рамки. Что можно предпринять?

Можно переписать программу заново. Вы примените свой опыт проектировщика, исправите все прежние грехи, будете работать творчески и с удовольствием. Но кто оплатит расходы? И можно ли быть уверенным, что новая система сможет делать все то, что делала старая?

Можно скопировать и модифицировать части существующей системы, чтобы расширить ее возможности. Это может показаться оправданным и даже будет рассматриваться как демонстрация повторного использования кода; не надо даже разбираться в том, что вы повторно используете. Однако с течением времени происходит размножение ошибок, программы разбухают, их архитектура разрушается и нарастают дополнительные издержки на модификацию.

Рефакторинг лежит посередине между этими двумя крайностями. Он дает возможность изменить существующее программное обеспечение, сделав понимание конструкции более явным, развить строение и извлечь повторно используемые компоненты, сделать яснее архитектуру программы, а также создать условия, облегчающие ввод дополнительных функций. Рефакторинг может способствовать извлечению выгоды из сделанных ранее инвестиций, уменьшить дублирование и рационализировать программу.

Допустим, что вас как разработчика привлекают эти преимущества. Вы согласны с Фредом Бруксом [2] в том, что модернизация представляет собой «внутренне присущую сложность» разработки программного обеспечения. Вы согласны с тем, что теоретически рефакторинг дает указанные преимущества.

Почему же вы все-таки не применяете рефакторинг к своим программам? Возможны четыре причины:

1. Вы можете не понимать, как выполнять рефакторинг.

2. Если выгоды ощутимы лишь в далекой перспективе, зачем тратить силы сейчас? В долгосрочной перспективе, когда придет время пожинать плоды, вы можете оказаться вне проекта.

3. Рефакторинг кода является непроизводительной деятельностью, а вам платят за новые функции.

4. Рефакторинг может нарушить работу имеющейся программы.

Все это законные причины для беспокойства. Мне доводилось выслушивать их от персонала коммуникационных и высокотехнологических компаний. Одни из них относятся к технологиям, а другие - к администрированию. Все они должны быть рассмотрены, прежде чем разработчики взвесят возможность рефакторинга своего программного обеспечения. Разберемся с каждой из них по очереди.

Как и когда применять рефакторинг

Как можно научиться рефакторингу? Каковы инструменты и технологии? В каких сочетаниях они могут принести какую-нибудь пользу? Когда следует их применять? В этой книге описано несколько десятков рефакторингов, которыми Мартин пользуется в своей работе. Приведены примеры применения рефакторингов для внесения в программы важных изменений.

В проекте Software Refactory, осуществлявшемся в Университете штата Иллинойс, мы выбрали минималистский подход. Мы определили небольшой набор рефакторингов [1,3] и показали, как можно их применять. Отбор рефакторингов был основан на нашем личном опыте программирования. Мы оценили структурную эволюцию нескольких объектно-ориентированных сред, в основном связанных с C++, и разговаривали с несколькими опытными разработчиками Smalltalk, а также читали их обзоры. Большинство наших рефакторингов относится к низкому уровню, например, они создают или удаляют класс, переменную или функцию, изменяют атрибуты переменных и функций, такие как права доступа (например, public или protected) и аргументы функций, либо перемещают переменные и функции между классами. Небольшая группа рефакторингов верхнего уровня применяется в таких операциях, как создание абстрактного родительского класса, упрощение класса посредством создания подклассов и упрощения условных операторов либо отделение части существующего класса с созданием нового класса повторно используемого компонента (часто с взаимным преобразованием наследования и делегирования или агрегирования). Более сложные рефакторинги определяются на языке рефакторингов низкого уровня. Наш подход был оправдан соображениями поддержки автоматизации и надежности, о чем будет сказано ниже.

Если есть готовая программа, какие виды рефакторинга следует к ней применять? Конечно, это зависит от стоящих перед вами задач. Одним из распространенных оснований для проведения рефакторинга, которое находится в центре внимания этой книги, является облегчение ввода (краткосрочная цель) новой функции. Это обсуждается в следующем разделе. Есть, однако, и другие основания для проведения рефакторинга.

Опытные программисты, использующие объектно-ориентированный подход, и те, кто имеет подготовку по паттернам проектирования и правильным технологиям проектирования, знают, что есть несколько желательных структурных характеристик и свойств программ для поддержки расширяемости и повторного использования кода [4-6]. Такие объектно-ориентированные технологии проектирования, как CRC [7], сосредоточены на определении классов и их протоколов. Хотя упор делается на предварительном проектировании, можно и имеющиеся программы оценивать исходя из их соответствия этим руководящим принципам.

Можно использовать автоматизированные средства для выявления структурных слабостей программ, таких как функции с чрезмерно большим количеством аргументов или слишком длинные функции. Такие функции являются кандидатами на проведение рефакторинга. С помощью автоматизированного средства можно также выявлять структурное сходство, служащее признаком избыточности. Например, если есть две почти совпадающие функции (как часто случается, если одна функция создается путем копирования и модификации другой), то может быть обнаружено их сходство и 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 лишь несколько простых примеров. С помощью автоматического средства можно выявить и исправить много других, более сложных случаев. Такие структурные аномалии или структурные сходства не всегда, но очень часто означают, что требуется применить рефакторинг.

Значительная часть работы над паттернами проектирования сосредоточилась на хорошем стиле программирования и полезных паттернах взаимодействия между частями программы, которые могут быть отображены в структурные характеристики и в рефакторинг. Например, раздел применимости паттерна шаблона метода [8] имеет отношение к нашему рефакторингу абстрактного родительского класса [9].

Я перечислил [1] некоторые эвристики, которые могут помочь выделить кандидатов на рефакторинг в программе, написанной на C++. Джон Брант и Дон Роберте [10,11] создали инструмент, применяющий обширный набор эвристик для автоматического анализа программ Smalltalk. Они предлагают рефакторинги, способные улучшить архитектуру программы, и указывают, где применять их.

Применение такого инструмента для анализа программы отчасти аналогично применению lint к программе C или C++. Это средство не настолько интеллектуально, чтобы понять смысл программы. Лишь немногие из предложений, которые оно выдвигает на основе структурного анализа программы, действительно следует выполнять. Как программист вы сохраняете за собой право выбора. Вам решать, какие рекомендации применить к своей программе. Эти изменения должны улучшить структуру программы и лучше поддерживать предстоящие модификации.

Прежде чем программисты убедятся в том, что они должны подвергать свой код рефакторингу, они должны понять, как и где применять рефакторинг. Ничто не может заменить опыт. Мы применяли в своем исследовании знания опытных программистов, применяющих объектно-ориентированный подход, чтобы получить набор полезных рефакторингов и понимание того, где они должны быть применены. Автоматизированные средства 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 их применять. По мере того как программисты применяют рефакторинг к своему коду, растет их понимание.

Рефакторинг программ C++

Билл Апдайк

Когда мы с Ральфом Джонсоном начали исследование рефакторинга в 1989 году, язык программирования C++ развивался и становился очень популярным среди объектно-ориентированных разработчиков. Значение рефакторинга впервые было понято в сообществе Smalltalk. Мы почувствовали, что демонстрация применимости рефакторинга к программам C++ должна заинтересовать более широкое сообщество объектно-ориентированных разработчиков.

В C++ есть такие свойства, особенно статическая проверка типов, которые упрощают некоторые задачи анализа программы и рефакторинга. С другой стороны, язык C++ сложен, в значительной мере из-за своей истории и происхождения от языка программирования С. Некоторые стили программирования, допускаемые в C++, затрудняют рефакторинг и развитие программ, написанных на нем.

Особенности языков и стилей программирования, способствующие рефакторингу

Имеющиеся в C++ функции статических типов позволяют относительно легко сузить возможные ссылки на ту 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 ссылки на нее. Поиск и замена ссылок могут быть осложнены, если программа большая.

В сравнении со Smalltalk в C++ есть такие особенности наследования классов и режима управления защитой-доступом (public, protected, private), которые облегчают локализацию обращений к переименовываемой функции. Если переименовываемая функция объявлена закрытой в классе, то ссылки на нее могут производиться только в самом этом классе или классах, объявленных дружественными этому классу. Если функция объявлена защищенной, то ссылки на нее можно найти только в самом классе, его подклассах (и их потомках) и среди дружественных классов. Если функция объявлена открытой (наиболее слабый режим защиты), то анализ все равно ограничен списком классов для защищенных функций и операций в экземплярах класса, который содержит функцию, его подклассами и потомками.

В некоторых очень больших программах в разных местах могут быть объявлены функции с одинаковыми именами. В некоторых случаях две или более одноименные функции лучше заменить одной функцией; есть рефакторинги, часто применяемые для таких изменений. С другой стороны, иногда нужно переименовать одну из функций, а другую сохранить. В проекте, над которым работает несколько человек, два или более программистов могут дать одно и то же имя никак не связанным между собой функциям. В C++ при изменении имени одной из таких функций почти всегда легко определить, какие ссылки относятся к переименовываемой функции, а какие - к другой функции. В Smalltalk такой анализ провести сложнее.

Поскольку в C++ для реализации подтипов используются подклассы, область видимости переменной или функции обычно можно расширить или сузить, перемещая их вверх или вниз по иерархии наследования. Действия по анализу программы и рефакторингу довольно просты.

Несколько правильных принципов программирования, примененных в начале разработки и в течение всего процесса, облегчают процедуру рефакторинга и упрощают развитие программного обеспечения. Определение членов-данных и большинства членов-функций как закрытых или защищенных представляет собой прием абстрагирования, который часто упрощает рефакторинг внутреннего устройства класса и минимизирует необходимость изменений в других частях программы. Применение наследования для моделирования иерархий обобщения и специализации (что обычно для C++) позволяет в будущем довольно легко расширять или сужать область действия членов-переменных или функций путем применения рефакторинга, перемещающих эти члены внутри иерархий наследования.

Особенности сред C++ поддерживают рефакторинг. Если во время проведения рефакторинга программы программист вносит ошибку, компилятор C++ часто может сигнализировать об этом. Многие среды разработки программного обеспечения C++ обеспечивают мощные возможности создания перекрестных ссылок и просмотра кода.

Особенности языков и стилей программирования, осложняющие рефакторинг

Совместимость C++ с С является, как хорошо знает большинство из вас, палкой о двух концах. На С написано много программ, и много программистов обучено работе на С, что (по крайней мере, на первый взгляд) делает миграцию на C++ легче, чем на другие объектно-ориентированные языки. Однако C++ поддерживает много стилей программирования, из которых не все соответствуют принципам качественного проектирования.

Программы, использующие такие возможности C++, как указатели, преобразования типов и sizeof(object), трудно поддаются рефакторингу. Указатели и операции преобразования типа создают наложение имен, затрудняющее обнаружение всех ссылок на объект, который требуется подвергнуть рефакторингу. Каждая из этих особенностей показывает внутреннее представление объекта, что нарушает принципы абстрагирования.

Например в C++ используется механизм V-таблиц для представления членов-переменных в выполняемой программе. Сначала появляется группа унаследованных членов-переменных, за которыми идут локально определенные члены-переменные. Один из безопасных в целом рефакторингов состоит в перемещении переменной в родительский класс. Поскольку в результате переменная наследуется, а не определена локально в подклассе, физическое местоположение переменной в исполняемом модуле после рефакторинга должно измениться. Если все обращения к переменным в программе производятся через интерфейсы классов, такое изменение физического местонахождения переменной не должно отразиться на поведении программы.

С другой стороны, 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 родительский класс, вероятно, повлечет изменение поведения программы. Аналогично, если программист написал условный оператор вида if (sizeof(object)=15)) и произвел рефакторинг программы с целью удаления из класса переменных, на которые нет ссылок, то размер экземпляров этого класса уменьшится, и условие, бывшее раньше истинным, теперь окажется ложным. Предположение, что кто-то пишет программы, выполняющие условные действия в зависимости от размера объекта или использующие арифметику указателей, когда C++ предоставляет значительно более понятный интерфейс к переменным класса, может показаться абсурдным. Но я хочу отметить, что такие возможности (и другие, зависящие от физического расположения объектов) в C++ есть, и есть программисты, привыкшие пользоваться ими. Переход с C на C++ не создает объектно-ориентированных программистов или проектировщиков.

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 того, безопасен ли рефакторинг, и если да, то для проведения его.

Поскольку C++ разрешает большинство ссылок на этапе компиляции, рефакторинг программы обычно требует перекомпиляции, по крайней мере, части ее, и компоновки исполняемого модуля перед тестированием результата модификации. Напротив, Smalltalk и CLOS (Common Lisp Object System) предоставляют среду для интерпретации и инкрементной компиляции. В то время как применение (а возможно, 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 выполняют небольшие изменения.

Во многих приложениях используются базы данных. Изменения в структуре объектов программы C++ могут потребовать соответствующих изменений в схеме базы данных. (Многие из идей, применявшихся мной в практическом рефакторинге, появились в результате исследований эволюции объектно-ориентированной схемы базы данных.)

Другое ограничение, которое может больше заинтересовать исследователей, а не большинство 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. Протокол метаобъектов CLOS, к примеру, поддерживает полезный иногда рефакторинг для превращения отдельных экземпляров класса в экземпляры другого класса с автоматическим перенацеливанием всех ссылок со старых объектов на новые. К счастью, те случаи, когда мне были желательны или необходимы такие возможности, были довольно редкими.

Завершающие комментарии

Приемы рефакторинга могут применяться и реально применялись к программам C++ в различных контекстах. Программы на C++ часто предполагается развивать в течение нескольких лет. Как раз в процессе такого развития программного обеспечения легче всего увидеть выгоды рефакторинга. Этот язык предоставляет некоторые возможности, упрощающие рефакторинг, тогда как другие характеристики языка, если они используются, делают задачу рефакторинга более трудной. К счастью, широко распространено мнение, что лучше не пользоваться такими возможностями языка, как арифметика указателей, поэтому большинство хороших объектно-ориентированных программистов избегает их.

Большая благодарность Ральфу Джонсону, Мику Мерфи, Джеймсу Роскинду и другим, кто познакомил меня с некоторыми сильными сторонами и сложностью C++ в отношении рефакторинга.

Рефакторинг как средство получения выгод в короткие сроки

Относительно просто описать средне- и долгосрочные выгоды рефакторинга. Многие организации, однако, все чаще оцениваются инвесторами и не только по краткосрочным результатам. Может ли рефакторинг принести выгоду в короткие сроки?

Рефакторинг успешно применяется уже более десяти лет опытными объектно-ориентированными разработчиками. Многие из этих программистов приобрели опыт в культуре Smalltalk, где ценятся ясность и простота кода и принято его повторное использование. В такой культуре программисты тратят время на рефакторинг, потому что это «правильное дело». 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, PARC либо небольшие группы программистов в технически передовых компаниях и консультационных фирмах. В таких группах были приняты ценности, несколько отличающиеся от свойственных многим группам промышленной разработки программного обеспечения. Как Мартин, так и я понимаем, что для принятия рефакторинга основной массой сообщества разработчиков программного обеспечения хотя бы некоторые его выгоды должны проявляться достижимы в короткие сроки.

Наша исследовательская группа [3,9,12-15] описала несколько примеров того, как рефакторинги могут перемежаться с развитием программы, так чтобы достигались как краткосрочные, так и долговременные выгоды. Одним из примеров служит среда файловой системы Choices. Первоначально эта среда реализовывала формат файловой системы BSD Unix (Berkeley Software Distribution). Позднее она была расширена 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 среды, заключался в том, чтобы сначала клонировать части реализации BSD Unix, а затем модифицировать этот клон для поддержки System V. Полученная реализация была работоспособной, но в ней было много дублирующегося кода. После добавления нового кода разработчик среды произвел рефакторинг кода, создав абстрактный родительский класс, содержащий общее поведение для двух реализаций файловой системы Unix. Общие переменные и функции были перемещены в родительский класс. В тех случаях, когда соответствующие функции для двух реализаций файловой системы были почти идентичными, но не совпадали полностью, в каждом из подклассов были определены новые функции, содержащие различия. В исходных функциях эти участки кода были заменены вызовами новых функций. Код в двух подклассах пошагово был приведен к большему сходству. Когда функции оказывались идентичными, они перемещались в общий родительский класс.

Эти рефакторинги предоставляют несколько преимуществ, достижимых в короткие и средние сроки. Через короткий срок достигнуто то преимущество, что ошибки, найденные при тестировании в общем коде, надо исправлять только в одном месте. Общий объем кода уменьшился. Поведение, специфическое для формата конкретной файловой системы, четко отделялось от кода, общего для форматов обеих файловых систем. Это облегчило нахождение и исправление ошибок, специфических для каждого формата. Выгодой, достигнутой в результате рефакторинга в течение среднего срока, была возможность применения полученных абстракций для определения новых файловых систем. Естественно, общее для двух имеющихся форматов файловых систем поведение могло не полностью подходить для третьего формата, но служило удобной отправной точкой. Можно было применить дополнительные рефакторинги, чтобы выявить действительно общее. Группа разработчиков среды обнаружила, что с течением времени требовалось меньше дополнительных усилий для добавления нового формата файловой системы. Несмотря на то что новые форматы были сложнее, разработку могли вести менее опытные программисты.

Можно было бы привести и другие примеры преимуществ, приносимых рефакторингом за короткие и средние сроки, но Мартин уже это сделал. Вместо того чтобы расширять его список, я хочу провести аналогию с тем, что близко и дорого многим из нас - с физическим здоровьем.

Во многих отношениях рефакторинг подобен физическим упражнениям и соблюдению правильной диеты. Многие знают, что надо больше заниматься физкультурой и есть более сбалансированную пищу. Некоторые из нас живут в культурах, активно поощряющих здоровые привычки. Иногда 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.

Иногда нас подстегивают перспективы ближайших выгод, приносимых упражнениями и соблюдением диеты, например повышенная бодрость, гибкость, более высокая самооценка и др. Почти каждому известно, что такие краткосрочные выгоды вполне реальны. Многие, хотя и не все, предпринимают, по крайней мере, спорадические усилия в этих областях. Другие, однако, не чувствуют достаточной мотивации что-то делать, пока не достигнут критической точки.

Конечно, необходимо соблюдать некоторые меры осторожности. По поводу физических упражнений и диеты следует проконсультироваться со своим врачом. В отношении рефакторинга надо обратиться к имеющимся ресурсам, таким, как эта книга, и статьям, цитируемым в разных местах этой главы. Те, кто имеет опыт проведения рефакторинга, могут предоставить более целенаправленную помощь.

Я встречал людей, которые могут служить образцами для подражания в отношении физической формы и рефакторинга. Я восхищаюсь их энергией и продуктивностью. Отрицательные примеры наглядно демонстрируют результаты небрежности. Их собственное будущее и будущее создаваемых ими программных систем не кажется безоблачным.

Рефакторинг может приносить выгоды в короткие сроки и облегчать модификацию и сопровождение программного обеспечения. Рефакторинг представляет собой средство, а не конечную цель. Это часть более широкого контекста разработки и сопровождения своих программ отдельными программистами или группами программистов [3].

Уменьшение стоимости рефакторинга

«Рефакторинг кода является непроизводительной деятельностью. Мне платят за создание новых функций, приносящих доход». Мой ответ на это вкратце таков:

Существуют инструменты и технологии, с помощью которых рефакторинг можно осуществлять быстро и сравнительно легко.

По опыту некоторых объектно-ориентированных программистов затраты на рефакторинг более чем компенсируются сокращением затрат и сроков в других фазах разработки программ.

Хотя поначалу рефакторинг может показаться несколько неудобным и требующим лишних затрат занятием, постепенно становясь частью режима разработки программного обеспечения, он перестает восприниматься как дополнительные затраты и становится неотъемлемой частью процесса.

Наверное, наиболее зрелый инструмент для автоматического рефакторинга был разработан для Smalltalk в группе Software Refactory Университета штата Иллинойс (см. главу 14). Его можно бесплатно получить на веб-сайте http://www.cs.uiuc.edu. Хотя средства проведения рефакторинга для других языков не столь общедоступны, многие из приемов, описанных в наших работах и этой книге, относительно просто применять, используя простой текстовый редактор, а еще лучше броузер. Среды разработки программного обеспечения и броузеры получили за последние годы существенное развитие. Мы надеемся на появление в будущем ширящегося множества средств проведения рефакторинга.

Кент Бек и Уорд Каннингем, опытные программисты на Smalltalk, сообщали на конференциях OOPSLA и других форумах, что рефакторинг позволил им быстро разрабатывать программы для таких областей, как торговля ценными бумагами. Аналогичные свидетельства мне приходилось слышать от разработчиков программ на C++ и CLOS. В данной книге Мартин описывает выгоды рефакторинга на примере программ Java. Мы надеемся услышать новые свидетельства от тех, кто прочтет эту книгу и применит изложенные принципы.

Мой опыт свидетельствует, что по мере того, как рефакторинг становится частью стандартного процесса, он перестает восприниматься как дополнительные расходы. Конечно, такие заявления легко делать и трудно обосновывать. Обращаясь к скептикам среди читателей, я советую им просто попробовать, а потом принять для себя решение. Дайте только время рефакторингу показать себя.

Безопасность рефакторинга

Надежность является важным требованием, особенно в организациях, разрабатывающих и развивающих крупные системы. Для многих приложений существуют 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, чтобы гарантировать надежность создаваемых продуктов.

Однако многие программисты зачастую не слишком озабочены проблемами надежности. Есть некоторая ирония в том, что многие из нас проповедуют безопасность как главное в отношении своих детей и родственников и требуют свободы для программиста как некоего гибрида вооруженного бандита с Дикого Запада и подростка за рулем. Дайте нам свободу, дайте нам средства и отпустите на волю. Что же, лишить нашу организацию плодов наших творческих возможностей ради какой-то однотипности и следования нормам?

В данном разделе я обсуждаю подходы к безопасному выполнению рефакторинга. Я сосредоточу внимание на таком подходе, который по сравнению с описываемым Мартином в этой книге более структурирован, но позволяет избежать многих ошибок, которые могут появиться в результате рефакторинга.

Безопасность - трудно определимое понятие. Интуитивно можно определить, что безопасным является такой рефакторинг, который не нарушает работу программы. Поскольку рефакторинг имеет целью реорганизовать программу, не меняя ее поведения, после рефакторинга программа должна выполняться так же, как и до него.

Как обеспечить безопасность рефакторинга? Есть несколько вариантов:

Надеяться на свое умение писать код.

Надеяться на свой компилятор в поиске не замеченных вами ошибок.

Надеяться на свой комплект тестов в поиске ошибок, не замеченных вами и вашим компилятором.

Надеяться на разбор кода, который выявит ошибки, не замеченные вами, вашим компилятором и вашим набором тестов.

Мартин при проведении рефакторинга сосредоточивается на первых трех вариантах. В средних и крупных организациях эти шаги часто дополняются разбором кода.

Несмотря на ценность компиляторов, комплектов тестов и дисциплинированного стиля кодирования, все эти подходы имеют определенные ограничения:

Программисты, даже такие как вы, могут делать ошибки (не говоря обо мне).

Существуют скрытые и не очень скрытые ошибки, которые компиляторы не умеют обнаруживать, особенно в областях действия, связанных с наследованием [1].

Перри и Кайзер [16], а также другие показали, что хотя общепринято считать (или, по крайней мере, так было), что тестирование облегчается, когда в качестве техники реализации используется наследование, в действительности часто требуется обширный набор тестов, чтобы охватить все случаи, когда операции, ранее запрашивавшиеся над экземплярами класса, запрашиваются над экземплярами его подклассов. Если только ваш 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. Иными словами, комплект тестов не может гарантировать охват всех случаев.

Рецензирующие код, 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 и создании прототипа инструмента проведения рефакторинга для проверки возможности безопасного рефакторинга программы и в случае положительного ответа его проведения. В результате удается избежать многих ошибок, появляющихся в результате действия человеческого фактора.

При сем прилагаю описание на высоком уровне моего подхода к безопасному проведению рефакторинга. Возможно, это самое ценное в данной главе. Более подробные сведения можно получить из моей диссертации [1] и других работ, указанных в конце главы; см. также главу 14. Если этот раздел покажется вам слишком специальным, можно пропустить его, за исключением нескольких последних параграфов.

В состав моего инструмента рефакторинга входит анализатор программ, представляющий собой программу, анализирующую другую программу (в данном случае программу C++, к которой желательно применить рефакторинг). Этот инструмент может ответить на ряд вопросов, касающихся областей видимости, типов и семантики программы (смысла или замысла действия программы). Проблемы областей видимости, связанные с наследованием, делают такой анализ сложнее, чем для многих не объектно-ориентированных программ, но в C++ есть такие особенности языка, как статическая типизация, которые делают анализ проще, чем, скажем, для Smalltalk.

Рассмотрим, например, рефакторинг, удаляющий из программы переменную. Данный инструмент может определить, в каких других частях программы происходит обращение к этой переменной (если они есть). Если обращения к переменной есть, то после ее удаления останутся повисшие ссылки, и рефакторинг, таким образом, не будет безопасным. Пользователь, попросивший инструмент произвести рефакторинг программы, получит сообщение об ошибке. Тогда пользователь может решить, что проведение данного рефакторинга вообще было плохой идеей, либо изменит те части программы, которые ссылаются на переменную, и выполнит рефакторинг, удаляющий переменную. Есть много других проверок, в большинстве своем таких же простых, но часто и более сложных.

В моем исследовании я определил безопасность на языке свойств программы (относящихся к таким действиям, как определение области видимости и типизация), которые должны сохраниться после рефакторинга. Многие из этих свойств программы аналогичны ограничениям целостности, которые должны сохраниться при изменении схемы базы данных [17]. С каждым рефакторингом связан набор необходимых предварительных условий, при выполнении которых гарантируется сохранение свойств программы. Данный инструмент выполняет рефакторинг, только когда убедится, что все безопасно.

К счастью, безопасность рефакторинга часто устанавливается тривиальным образом, особенно для рефакторингов нижнего уровня, составляющих на практике большинство. Чтобы обеспечить безопасность более сложных рефакторингов высокого уровня, мы определили их на языке рефакторингов низкого уровня. Например, рефакторинг, создающий абстрактный родительский класс, определен через шаги, каждый из которых является более простым рефакторингом, таким как создание и перемещение переменных и методов. Показав, что каждый шаг сложного рефакторинга безопасен, мы по построению знаем, что безопасен весь рефакторинг.

Встречаются (относительно редко) случаи, когда рефакторинг в действительности можно безопасно провести, а инструмент в этом не уверен. В таких случаях инструмент перестраховывается и запрещает проведение рефакторинга. Рассмотрим, например, снова случай, когда вы хотите удалить из программы переменную, но в каком-то месте программы есть ссылка на нее. Возможно, ссылка присутствует в фрагменте кода, который никогда не выполняется. Например, ссылка находится в условном операторе, скажем, if-then, условие которого никогда не может быть истинным. Если есть уверенность, что условие никогда не окажется истинным, можно убрать проверку условия и код, ссылающийся на переменную или функцию, которые вы хотите удалить. После этого можно благополучно удалить переменную или функцию. Вообще говоря, нельзя наверное знать, что некоторое условие всегда будет ложным. (Допустим, вы наследуете код, разработанный кем-то другим. Можно ли быть уверенным, удаляя этот код?)

Инструмент проведения рефакторинга может сообщить о ссылке и предупредить пользователя. Пользователь может решить оставить код в покое. Если когда-нибудь у пользователя возникнет уверенность в том, что код со ссылкой никогда не будет выполняться, он может удалить код и применить рефакторинг. Этот инструмент ставит пользователя в известность о возможных последствиях остающихся ссылок, а не осуществляет слепо модификацию.

Может показаться, что это сложно. Для диссертации это может быть хорошо (главная ее аудитория - оценивающая диссертацию комиссия, желает, чтобы было уделено внимание теоретическим проблемам), но практично ли это в реальном рефакторинге?

Вся проверка безопасности может выполняться невидимо для программиста инструментом, поддерживающим проведение рефакторин-га. Программист, которому требуется провести рефакторинг программы, просто просит инструмент проверить код и, если это безопасно, выполнить рефакторинг. Мой инструмент был прототипом, предназначенным для исследований. Дон Роберте, Джон Брант, Ральф Джонсон и я [10] реализовали гораздо более надежный и богатый функциями инструмент (см. главу 14) в ходе нашего исследования рефакторинга программ Smalltalk.

К рефакторингу можно применить много уровней безопасности. Некоторые из них легко применимы, но не гарантируют высокого уровня безопасности. Применение инструмента проведения рефакторинга имеет многие преимущества. Инструмент может произвести много простых, но утомительных проверок и заранее сигнализировать о проблемах, без решения которых работа программы будет нарушена в результате рефакторинга.

Хотя применение инструмента рефакторинга позволяет избежать появления многих из тех ошибок, обнаружить которые можно было бы во время компиляции, тестирования и разбора кода, указанные приемы сохраняют свою ценность, особенно при разработке или развитии систем реального времени. Часто 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 скорость выполнения программы. Повышение скорости работы одной программы может привести к возникновению узких мест в других частях системы. Совершенно аналогично после замены микропроцессора более новым, что ускоряет работу отдельных частей системы, требуется настройка и тестирование работы всей системы. И наоборот, некоторые рефакторинги могут несколько снизить общую производительность, но в целом такие воздействия на производительность минимальны.

Безопасные подходы имеют целью гарантировать, что рефакторинг не внесет в программу новых ошибок. Они не обнаруживают и не исправляют те ошибки, которые присутствовали в программе до ее рефакторинга. Однако рефакторинг может облегчить обнаружение таких ошибок и исправление их.

Возращаясь к проверке в реальных условиях

Проведение рефакторинга на практике требует обращения к тому, что реально беспокоит профессиональных разработчиков программного обеспечения. Часто выдвигаются следующие четыре возражения:

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 кода является непроизводительной деятельностью, а программистам платят за новые функции.

Рефакторинг может нарушить работу имеющейся программы.

В этой главе я кратко остановлюсь на каждом из этих возражений и дам ссылки для тех, кто хочет глубже изучить эти проблемы.

В некоторых проектах возникают следующие проблемы:

Как быть, если подлежащий рефакторингу код находится в коллективной собственности нескольких программистов? К некоторым случаям относятся традиционные механизмы смены управления. В других случаях, когда программное обеспечение правильно спроектировано и структурировано, подсистемы оказываются достаточно разъединенными, чтобы многие рефакторинги воздействовали лишь на небольшое подмножество кода.

Как быть, если есть несколько версий исходного кода? В некоторых случаях уместно проведение рефакторингов для всех версий кода, и тогда все они должны проверяться на безопасность перед применением рефакторинга. В других случаях рефакторинг может касаться лишь некоторых версий, что упрощает проверку и рефакторинг кода. Управление модификацией нескольких версий часто требует применения многих приемов традиционного управления версиями. Рефакторинг может оказаться полезным при объединении вариантов или версий в модифицированный основной код, что может упростить управление версиями.

Подводя итоги, отмечу, что убеждать профессиональных разработчиков программного обеспечения в практической ценности рефакторинга надо совсем не так, как убеждают совет по присуждению ученых степеней в том, что исследование заслуживает присуждения степени Ph.D. Чтобы вполне оценить разницу, мне потребовалось определенное время после завершения моего исследования.

Ресурсы и ссылки, относящиеся к рефакторингу

Я надеюсь, что, добравшись в книге до этого места, вы уже собираетесь применять технологию рефакторинга в своей работе и поощряете своих коллег к тому же. Тем, кто все еще пребывает в нерешительности, возможно, следует воспользоваться приводимой мной библиографией или связаться с Мартином (Fowler@acm.org), со мной или с другими, у кого есть опыт применения рефакторинга.

Для тех, кто хочет глубже изучить рефакторинг, приведу список книг, которые, может быть, следует заказать. Как отмечено Мартином, эта книга - не первый печатный труд по рефакторингу, но (я надеюсь) она откроет идеи и преимущества рефакторинга для более широкой аудитории. Хотя моя докторская диссертация была первой крупной печатной работой по этой теме, большинству читателей, интересующихся ранними основополагающими работами по рефакторингу, следует сначала просмотреть несколько статей [3,9,12,13]. Рефакторинг был темой консультаций на OOPSLA 95 и OOPSLA 96 [14,15]. Тем, кого интересуют как паттерны проектирования, так и рефакторинг, адресована работа «Lifecycle and Refactoring Patterns That Support Evolution and Reuse» [3] (Жизненный цикл и паттерны, поддерживающие развитие и повторное использование), которую Брайан Фут (Brian Foote) и я представили на PLoP '94 и появившаяся в первом томе серии Addison-Wesley «Pattern Languages of Program Design», будет хорошей отправной точкой. Мое исследование рефакторинга было в большой степени основано на работе Ральфа Джонсона и Брайана, касающейся сред объектно-ориентированных приложений и проектирования повторно используемых классов [4]. Последующие исследования рефакторинга, которые проводили Джон Брант, Дон Роберте и Ральф Джонсон в Университете штата Иллинойс, было посвящено рефакторингу программ Smalltalk [10,11]. На их веб-сайте (http://www.cs.uiuc.edu) выложены некоторые из их последних работ. Интерес к рефакторингу вырос в сообществе исследователей объектно-ориентированного программирования. Несколько работ на эту тему было представлено на OOPSLA 96 на секции «Refactoring and Reuse» (Рефакторинг и повторное использование кода) [18].

Последствия повторного использования программного обеспечения и передачи технологий

Практические проблемы, к которым мы обратились выше, касаются не только рефакторинга. Они применимы в более широком плане к развитию и повторному использованию программного обеспечения.

В течение нескольких последних лет я много занимался проблемами, связанными с повторным использованием программного обеспечения, платформами, средами, паттернами и эволюцией унаследованных систем, в которых часто есть программное обеспечение, не использующее 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 тем, которые относятся к рефакторингу.

Технический персонал может не понимать, что и как повторно использовать.

Технический персонал может оказаться незаинтересованным в применении подхода с повторным использованием, если при этом нельзя быстро добиться выгоды.

Чтобы повторное использование было принято, надо решить проблемы стоимости дополнительной работы, обучения и развертывания.

Принятие подхода с повторным использованием кода не должно привести к срыву проекта; может оказываться сильный нажим в пользу применения имеющихся средств или реализации, хотя и с действующими

ограничениями. Новые реализации должны взаимодействовать или быть обратно совместимыми с существующими системами.

Джеффри Мур [23] описал процесс принятия технологии в виде колоколообразной кривой, на переднем фронте которой находятся новаторы и те, кто рано принял технологию, большой средний горб содержит образовавшееся в начале и запоздалое большинство, а в хвостовой части плетутся нерасторопные. Иными словами, многие идеи, притягательные для новаторов, в итоге оказываются неудачными, потому что никогда не дойдут до большинства. Отсутствие контакта обусловлено, в основном, 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, стоимостью, наличием поддержки и посматривает, насколько успешно новая идея или продукт применяются другими, у кого существуют аналогичные потребности.

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 и особенно менеджеры часто входят в раннее и позднее большинство. Распознать эти различия важно, чтобы достичь каждой из этих групп. Для повторного использования программного обеспечения, как и для рефакторинга, важно предлагать профессиональным разработчикам то, что их интересует.

В Lucent/Bell Labs я обнаружил, что для поощрения платформ и повторного использования кода необходимо установить контакт с рядом заинтересованных кругов. Это потребовало формулировки стратегии для администраторов, организации совещаний руководителей среднего звена, консультаций с разрабатываемыми проектами и оповещения о преимуществах этих технологий широких аудиторий исследователей и разработчиков посредством семинаров и публикаций. И всюду было важно обучать персонал принципам, отмечать выгоды, достигаемые в короткие сроки, обеспечивать сокращение издержек и заботиться о безопасном внедрении этих технологий. Понимание этого я приобрел из опыта исследований рефакторинга.

Как отметил, просматривая черновик этой главы, Ральф Джонсон, 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 убеждать других людей в достоинствах рефакторинга (или принятия какой-либо другой технологии или практики), сосредоточьтесь на отмеченных проблемах и убеждайте людей, учитывая их интересы. Передавать технологию трудно, но можно.

Завершающее замечание

Спасибо, что нашли время прочесть эту главу. Я попытался коснуться многих опасений, которые могли возникнуть у вас в отношении рефакторинга, и показать, что многие из встречающихся в реальной жизни опасений, касающихся рефакторинга, имеют более широкое отношение к развитию и повторному использованию программного обеспечения. Я надеюсь, что эта глава придаст вам энтузиазма в применении этих идей в своей работе. Желаю вам прогресса в решении 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 IF-IP 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 In ternational 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 OOPSLA96: 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 Op-dyke. «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 PloP97: 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.

14 ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ПРОВЕДЕНИЯ РЕФАКТОРИНГА

Дон Роберте и Джон Брант

Одним из крупнейших препятствий на пути к проведению рефакторинга кода была печальная недостаточность инструментальных средств, поддерживающих его. В языках, для которых рефакторинг составляет часть культуры, таких как Smalltalk, обычно имеются мощные среды, поддерживающие многие из функций, необходимых для рефакторинга кода. Но даже в них до недавнего времени поддержка этой процедуры была лишь частичной, и большая часть работы по-прежнему осуществляется вручную.

Рефакторинг с использованием инструментальных средств

Рефакторинг с поддержкой автоматизированных средств выглядит иначе, чем ручной. Даже при наличии страховки в виде комплекта тестов рефакторинг, производимый вручную, требует больших затрат времени. Это простое обстоятельство удерживает программистов от рефакторинга, необходимость которого им понятна, лишь потому, что обходится он слишком дорого. Если добиться того, чтобы рефакторинг оказывался не дороже, чем настройка формата кода, привести код в порядок можно аналогично тому, как приводится в порядок его внешний вид. Однако такая разновидность приведения кода в порядок может оказать глубокий положительный эффект на сопровождение, повторное использование и легкость понимания кода. Кент Бек говорит так:

> *Кент Бек*
>
> Броузер рефакторинга [The Restructurisation Browser] полностью изменяет представления о программировании. Всякие мелкие соображения типа «конечно, следовало бы поменять это имя, но...» уходят в прошлое, вы просто изменяете это имя, поскольку для этого нужно лишь выбрать один 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 пяти минут, которые мне понадобились бы, чтобы выполнить этот рефакторинг вручную, затем выполнял другой и снова смотрел в пространство. Наконец, я поймал себя на этом и понял, что необходимо научиться думать более крупными рефакторингами и делать это быстрее. Сейчас я делю время примерно пополам между рефакторингом и вводом нового кода и делаю это с одинаковой скоростью.

При таком уровне инструментальной поддержки рефакторинг все менее отделяется от программирования. Очень редко можно сказать: «Сейчас я программирую» или «Сейчас я занимаюсь рефакторингом». Скорее всего, мы скажем: «Выделяю эту часть метода, поднимаю ее в родительский класс, затем добавляю вызов нового метода в новый подкласс, над которым я работаю». Поскольку после автоматизированного рефакторинга тестирование не требуется, один вид деятельности переходит в другой, и процесс переключения между ними становится малозаметным, хотя по-прежнему происходит.

Рассмотрим важный вид рефакторинга - «Выделение метода» (Extract Method). Действуя вручную, нужно выполнить много проверок. С помощью «броузера рефакторинга» вы просто отмечаете текст, который нужно выделить, и выбираете в меню пункт Extract Method. Инструмент определит, можно ли выделить в метод помеченный текст. Есть несколько причин, по которым это может оказаться невозможным. В этом отрывке может оказаться помеченной лишь часть идентификатора, могут быть присваивания переменной, но содержаться не все ссылки на нее. Беспокоиться обо всем этом не надо, потому что проверкой занимается броузер. Затем он определит, сколько параметров должно быть передано в новый метод. Потом предложит ввести имя нового метода и позволит задать порядок параметров в вызове новой функции. Когда это будет сделано, инструмент извлечет код из исходного метода и заменит его вызовом функции. После этого он создаст новый метод в том же классе, что и исходный, с именем, заданным пользователем. Весь процесс занимает 15 секунд. Сравните это с временем, необходимым для выполнения шагов, указанных в «Выделении метода» (Extract Method).

По мере удешевления рефакторинга менее дорого обходятся ошибки проектирования. Поскольку дешевле обходится исправление ошибок проектирования, предварительное проектирование требует меньших усилий. Предварительное проектирование использует прогнозирование, поскольку технические требования не бывают полными. Поскольку кода нет, не ясно, как проектировать, чтобы упростить код. В прошлом приходилось держаться изначально созданного проекта, каким бы он ни был, потому что стоимость изменения проекта была слишком высока. Благодаря средствам автоматизации рефакторинга можно позволить себе работу с более подвижными проектами, поскольку изменение проекта стало обходиться значительно дешевле. Исходя из новой расстановки цен можно выполнять проектирование на уровне текущей задачи, зная, что в будущем можно будет без особых затрат расширить проект, введя в него дополнительную гибкость. Не надо больше пытаться предсказать все направления, по которым станет развиваться система. Если обнаружится, что текущий проект служит причиной неуклюжести кода и «душков», описанных в главе 3, можно быстро изменить проект, чтобы сделать код понятнее и легче в сопровождении.

Применение инструментальных средств рефакторинга оказывает воздействие на тестирование. Оно требуется в значительно меньшем объеме, поскольку многие рефакторинги выполняются автоматически. Всегда найдутся рефакторинги, которые нельзя автоматизировать, поэтому исключить этап тестирования не удастся никогда. Опыт показывает, что в течение дня выполняется столько же тестов, как и в средах без средств автоматизации рефакторинга, но рефакторингов выполняется больше.

Как отмечал Мартин, такие вспомогательные средства нужны для программистов на Java. Мы хотим выделить некоторые критерии, которым должен удовлетворять такой инструмент. Мы включили в их состав и технические критерии, но полагаем, что гораздо большее значение имеют критерии практические.

Технические критерии для инструментов проведения рефакторинга

Основная задача инструмента рефакторинга - позволить программисту производить рефакторинг кода без обязательного повторного 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 проведения рефакторинга, необходимые для преобразования программы с сохранением ее поведения.

База данных программы

Одним из первых требований, которые были осознаны, является возможность поиска различных программных объектов по всей программе: например, для некоторого метода - нахождения всех вызовов, потенциально реализующих обращения к рассматриваемому методу, или для некоторой переменной экземпляра - нахождения всех методов, выполняющих ее чтение или запись. В таких сильно интегрированных средах, как Smalltalk, эта информация постоянно поддерживается в форме, пригодной для поиска. Это не база данных в традиционном понимании, но это хранилище с возможностями поиска. Программист может выполнить поиск и найти перекрестные ссылки на любой программный элемент, главным образом благодаря динамической компиляции кода. Как только в какой-либо класс вносится изменение, оно немедленно компилируется в байт-коды, и «база данных» обновляется. В более статичных средах, таких как Java, программисты вводят код в текстовые файлы. Обновление базы данных должно происходить путем выполнения некой программы, обрабатывающей эти файлы и извлекающей нужную информацию. Такие обновления аналогичны компиляции самого кода Java. В некоторых более современных средах, таких как IBM VisualAge for Java, скопировано из Smalltalk динамическое обновление базы данных программы.

Простой подход заключается в использовании для поиска текстовых средств, таких как grep, но он быстро приводит к ошибкам, потому что не делает разницы между переменной с именем foo и функцией с именем foo. Создание базы данных требует применения семантического анализа (парсинга) для идентификации каждой лексема в программе как «части речи». Это надо делать как на уровне определения класса, чтобы выявить определения переменных экземпляра и методов, так и на уровне метода, чтобы выявить ссылки на переменные экземпляра и методы.

Деревья синтаксического разбора

В большинстве рефакторингов надо обрабатывать части системы, находящиеся ниже уровня метода. Обычно это ссылки на изменяемые программные элементы. Например, если переименовывается переменная экземпляра (просто изменяется определение), то должны быть обновлены все ссылки внутри методов этого класса и его подклассов. Другие рефакторинги целиком осуществляются ниже уровня метода, например, выделение части метода в отдельный самостоятельный метод. Любое обновление метода должно иметь возможность манипулировать структурой метода. Для этого необходимы деревья 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.pnntln(“Hello World”);
}
```

Соответствующее дерево синтаксического разбора показано на рисунке 14.1.

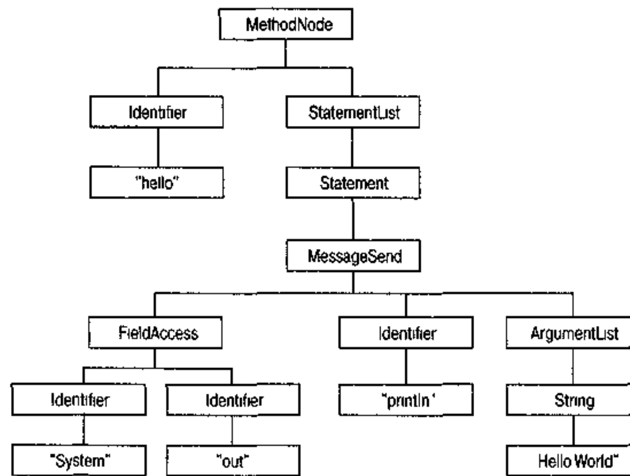


Рисунок 14.1 - Дерево синтаксического разбора для hello world

Точность

Выполняемые инструментальным средством рефакторинги должны в достаточной мере сохранять поведение программ. Полного сохранения поведения достичь невозможно. Что, например, 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 выполнению в реальном времени, она может начать работать некорректно.

Можно нарушить работу даже довольно обычных программ. Например, если программа строит строку и использует API reflection Java для выполнения метода с именем String, то в результате переименования метода программа сгенерирует исключительную ситуацию, которой не возникало в исходной программе.

Однако для большинства программ можно выполнять рефакторинги достаточно корректно. Если идентифицировать ситуации, вступающие в конфликт с рефакторингом, то программист может отказаться от проведения рефакторинга или вручную исправить те части программы, которые не может скорректировать инструментальное средство проведения рефакторинга.

Практические критерии для инструментов рефакторинга

Инструменты создаются для того, чтобы помочь человеку в решении конкретной задачи. Если инструмент не согласуется с тем, как работает человек, тот не станет им пользоваться. Самые важные критерии относятся к интеграции рефакторинга с другими инструментами.

Скорость

Анализ и преобразования, необходимые для проведения рефакторинга, могут потребовать больших затрат времени, если их сложность очень высока. Всегда следует учитывать относительную стоимость времени и точности. Если рефакторинг выполняется слишком долго, программист не станет прибегать к его автоматизированному выполнению, а просто произведет его вручную со всеми возможными последствиями. Скорость выполнения всегда должна приниматься во внимание. В процессе разработки броузера рефакторинга мы отказались от реализации нескольких рефакторингов только потому, что не могли надежно выполнять их за разумное время. Однако мы неплохо справились со своей задачей, и большинство наших рефакторингов выполняется очень быстро и точно. Теоретики склонны обращать внимание на все предельные случаи, с которыми не справляется определенный подход. На самом деле, большинство программ не представляют собой предельные случаи, и упрощенные быстрые подходы действуют удивительно хорошо.

Возможный подход в случае, когда анализ производится слишком медленно, состоит просто в том, чтобы запросить информацию у программиста. Тем самым ответственность за точность перекладывается обратно на программиста, позволяя в то же время быстро выполнить анализ. Весьма часто программист располагает необходимой информацией. Конечно, при таком подходе нельзя быть уверенным в безопасности, поскольку программист может ошибиться, но ответственность за ошибки ложится на него. Как ни странно, в результате программисты более склонны пользоваться инструментом, потому что им не приходится полагаться при поиске данных 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 дизайна. Поскольку

рефакторинг должен сохранять поведение, обратный рефакторинг, возвращающий к оригиналу, тоже является рефакторингом и должен сохранять поведение. В ранние версии броузера рефакторинга не была включена функция undo. Из-за этого рефакторинг оказывался несколько более экспериментальным, потому что отмена некоторых примененных рефакторингов, несмотря на сохранение поведения, была затруднена. Довольно часто приходилось искать старую версию программы и начинать сначала. Это досаждало. С вводом undo еще одно ограничение было снято. Теперь можно безнаказанно заниматься исследованиями, зная, что можно вернуться к любому предшествующему варианту. Можно очень быстро создавать классы, перемещать в них методы и смотреть, что за код получается, а затем изменять решение и двигаться совсем в другом направлении.

Интеграция с другими инструментами

В последнее десятилетие интегрированная среда разработки (IDE) была основой большинства разрабатываемых проектов. В IDE объединены редактор, компилятор, компоновщик, отладчик и любые другие инструменты, необходимые для разработки программ. Ранняя реализация броузера рефакторинга для Smalltalk представляла собой инструмент, отдельный от стандартных средств разработки для Smalltalk. Оказалось, что никто с ним не работает. Даже мы сами не применяли его. Как только мы встроили рефакторинги непосредственно в Smalltalk Browser, мы стали широко пользоваться ими. То, что теперь они всегда были под рукой, оказалось существенным отличием.

Краткое заключение

Мы занимались разработкой и применением броузера рефакторинга в течение нескольких лет. Нередко с его помощью мы производим рефакторинг его собственного кода. Одна из причин его успеха в том, что мы сами программисты и постоянно стремимся приспособить его к своему способу работы. Если мы сталкивались с рефакторингом, который приходилось выполнять вручную, и чувствовали его универсальность, то реализовывали его и добавляли в броузер. Если какой-то рефакторинг выполнялся слишком долго, мы добивались увеличения скорости его выполнения. Если что-то делалось не слишком точно, мы исправляли положение.

Мы уверены, что средства автоматизации рефакторинга предоставляют лучший способ справиться со сложностью, растущей по мере развития программного проекта. Без инструментов, позволяющих бороться с этой сложностью, программное обеспечение становится раздутым, насыщенным ошибками и хрупким. Поскольку Java гораздо проще того языка, с которым имеет общий синтаксис, значительно легче разрабатывать для него средства автоматизации рефакторинга. Мы надеемся, что это произойдет, и мы сможем избежать недостатков C++.

15 СКЛАДЫВАЯ ВСЕ ВМЕСТЕ

Кент Бек

Теперь все части головоломки перед вами. Вы познакомились с рефакторингом, изучили каталог и освоились со всеми списками шагов. Вы умеете тестировать и поэтому ничего не боитесь. Вам может теперь показаться, что вы умеете производить рефакторинг программы. Пока это не совсем так.

Перечень приемов составляет только начало. Это ворота, в которые надо войти. Без этой техники нельзя изменять конструкцию работающих программ. С ней тоже нельзя, но, по крайней мере, можно приступить к этому.

Почему же все эти замечательные приемы составляют только начало? Потому что вы пока не знаете, когда их нужно применять, а когда - нет, когда начать, а когда остановиться, когда двигаться вперед, а когда ждать. Рефакторинг это ритм, а не отдельные ноты.

Как узнать, что вы действительно начали его понимать? Вы узнаете об этом, когда на вас снизойдет спокойствие. Когда вы почувствуете абсолютную уверенность, что как бы ни был закручен код, который вам достался, вы можете сделать его лучше, настолько лучше, чтобы развивать его дальше.

Но главным признаком того, что вы освоили рефакторинг, будет уверенная остановка. Умение остановиться - самый сильный ход из имеющихся в запасе у применяющего рефакторинг. Вы видите крупную цель - можно убрать тьму подклассов. Вы начинаете двигаться к этой цели маленькими, но твердыми шагами, подкрепляя каждый шаг успешным выполнением всех тестов. Вы близки к цели. Осталось объединить два метода в каждом из подклассов, и можно про них забыть.

Тут-то оно и происходит. Мотор заглох. Может быть, уже поздно, и вы устали. Может быть, вы ошиблись изначально, и на самом деле избавиться от всех этих подклассов нельзя. Может быть, чувствуется недостаток тестов. Что бы ни было причиной, вы потеряли уверенность. Вы не можете сделать следующий шаг без сомнений. Вряд ли что-нибудь произойдет, но полной уверенности нет.

Вот тут надо остановиться. Если код уже стал лучше, соедините все сделанное вами вместе и выпускайте готовую версию. Если он не стал лучше, отойдите в сторону. Выбросьте и забудьте. Приятно получить урок, жаль, что не получилось. Что у нас по плану на завтра?

Завтра или через день, или через месяц, или через год (мой личный рекорд - девять лет перед тем, как приступить ко второй части рефакторинга) приходит озарение. Либо вы поймете, почему были неправы, либо поймете, почему были правы. В том и другом случае следующий шаг ясен. Вы сделаете его с той же уверенностью, с какой начинали. Может быть, вам будет даже несколько неловко оттого, что не сообразили раньше. Не надо стыдиться. С каждым бывает.

Это как идти 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.

Это может показаться мистическим и туманным. Отчасти так оно и есть, потому что вы вступили со своей программой в новые отношения. Если вы действительно понимаете рефакторинг, то архитектура системы так же подвижна, пластична и формируема для вас, как отдельные символы в файле с исходным кодом. Вы ощущаете сразу всю конструкцию. Вы видите, как она может изгибаться и изменяться - чуть-чуть в одну сторону, чуть-чуть в другую.

В ином смысле, однако, здесь нет никакой мистики и туманности. Рефакторинг - это искусство, которому можно научиться, о его составляющих вы прочли в этой книге и начали их изучать. Вы соединяете вместе мелкие приемы и шлифуете их. После этого разработка представляется вам в новом свете.

Я сказал, что это искусство, которому можно научиться. Как это сделать?

Привыкайте намечать цель. Местами у вашего кода есть «душок». Решитесь избавиться от этой проблемы. После этого двигайтесь к выбранной цели. Вы занимаетесь рефакторингом не в поисках истины и красоты (во всяком случае, это не единственная ваша задача). Вы стараетесь, чтобы ваш мир было проще понять, восстановить контроль над программой, которая живет своей жизнью.

Остановитесь, почувствовав неуверенность. По мере продвижения к цели может наступить момент, когда вы не можете уверенно доказать 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 и легко потерять из виду, даже если это происходит на мгновение. Я по-прежнему теряю перспективу чаще, чем хотелось бы. Я выполняю три-четыре рефакторинга подряд, не повторяя тестирования. Конечно, все обойдется. Я чувствую себя уверенно. У меня есть опыт. Оп! Тест провалился, и я не знаю, которое из моих изменений стало источником проблемы.

В этот момент может возникнуть сильный соблазн выбраться просто путем отладки. В конце концов, есть тесты, которые надо выполнить в первую очередь. Трудно ли будет добиться, чтобы они снова стали выполняться? Остановитесь. Вы потеряли управление, и у вас нет представления о том, чего будет стоить восстановление контроля при движении вперед. Вернитесь к своей последней успешной конфигурации. Воспроизведите модификации поочередно. Выполняйте тесты после каждой из них.

Это может казаться очевидным, когда вы удобно сидите в своем кресле. Когда вы работаете с кодом и чувствуете, что до существенного упрощения рукой подать, сложнее всего остановиться и возвратиться назад. Но думайте об этом сейчас, пока голова еще ясная. Если вы занимались рефакторингом час, то воспроизвести сделанное сможете за десять минут. Поэтому вы гарантированно сможете вновь выйти на свой путь в течение десяти минут. Если, однако, вы попытаетесь двигаться вперед, отладка может занять пять секунд, а может - два часа.

Объяснять, что сейчас надо делать, просто. Крайне тяжело действительно делать это. То, что я не следовал своему собственному совету, обошлось мне, пожалуй, в четыре часа потерянного времени в трех отдельных попытках. Я потерял контроль, вернулся назад, пошел вперед сначала медленно, снова потерял контроль, и еще раз - в течение четырех мучительных часов. Это не шутки. Вот почему вам нужна помощь.

Дуэты. Ради Бога, производите рефакторинг с кем-нибудь на пару. Есть много преимуществ работы парами во всех разновидностях разработки. При проведении рефакторинга преимущества оказываются чрезвычайными. Рефакторинг вознаграждает тщательную и методичную работу. Ваш напарник заставит двигаться шаг за шагом вас, а вы - его. При проведении рефакторинга вознаграждается способность увидеть возможно 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 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.

Другой стороной работы с партнером является возможность обсуждения. Вам надо поговорить о том, что, по вашему мнению, должно произойти, чтобы оба вы смотрели в одном направлении. О том, что, по вашему мнению, должно произойти, чтобы увидеть опасность как можно раньше. О том, что только что произошло, чтобы в следующий раз этого не случилось. Все эти разговоры точно закрепляют в вашем уме, как отдельные рефакторинги вписываются в ритм проведения рефакторинга.

Вполне можно открыть в своем коде новые возможности, даже проработав с ним годы, если знать, какие бывают «душки» и как их нейтрализовать. Может даже возникнуть желание взять и привести в порядок все недостатки, которые вам видны. Не делайте этого. Ни один руководитель не обрадуется, услышав, что команда должна прерваться на три месяца и убрать весь беспорядок, который она натворила. Да это и не нужно. Крупный рефакторинг - это способ навлечь на себя катастрофу.

Каким бы ужасным ни выглядел беспорядок, приучите себя ограничиваться краями проблемы. Если надо добавить некоторую новую функцию в одной области, потратьте несколько минут и приведите ее в порядок. Если требуется добавить некоторые тесты, чтобы быть уверенным в программе после приборки, добавьте их. Вы будете рады, что сделали это. После предварительного рефакторинга добавление нового кода не так опасно. Поработав с кодом, вы вспомните, как он действует. Вы быстрее закончите работу и получите удовлетворение, понимая, что когда вы обратитесь к этому коду в следующий раз, он будет выглядеть лучше, чем сейчас.

Не забывайте пересаживаться с одного коня на другого. Во время проведения рефакторинга вы неизбежно обнаружите случаи неправильной работы кода. Вы будете в этом абсолютно уверены. Не поддавайтесь соблазну. При проведении рефакторинга задача состоит в том, чтобы получившийся код выдавал точно те же ответы, что и перед тем, как вы за него взялись. Не больше и не меньше. Ведите список (у меня всегда рядом с компьютером лежит учетная карточка) того, что надо сделать позднее - добавить или изменить контрольные примеры, провести не относящиеся к делу рефакторинги, документы, которые необходимо составить, графики, которые надо нарисовать. Благодаря этому вы не потеряете свои мысли, но и не позволите им вмешиваться в то, чем вы занимаетесь в данный момент.

БИБЛИОГРАФИЯ

[Auer]

Auer, Ken. «Reusability through Self-Encapsulation.» In Pattern Languages of Program Design 1, edited by J.0. Coplien and D.C. Schmidt. Reading, Mass.: Addison- Wesley, 1995.

Работа по паттернам, связанным с идеей самоинкапсуляции.

[Blummer and Riehle]

Blummer, 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.

Паттерн гибкого создания объектов без знания того, в каком классе они должны быть.

[Beck]

Beck, Kent. Smalltalk Best Practice Patterns. Upper Saddle River, N.J.: Prentice Hall, 1997a.

Необходимая книга для всех, кто работает на Smalltalk, и чрезвычайно полезная для всех занимающихся объектно-ориентированными разработками. Ходят слухи о выпуске версии для Java.

[Beck, hanoi]

Beck, Kent. «Make it Run, Make it Right: Design Through Refactoring.» The Smalltalk Report, 6: (1997b): 19-24.

Первая печатная работа, действительно дающая ощущение того, как действует процесс рефакторинга. Источник ряда идей для главы 1.

[Beck, XP]

Beck, Kent, extreme Programming explained: Embrace Change. Reading, Mass.: Addison-Wesley, 2000.

[Fowler, UML]

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.

[Fowler, AP]

Fowler, M. Analysis Patterns: Reusable Object Models. Reading, Mass.: Addison- Wesley, 1997.

Книга о паттернах моделей предметных областей. Содержит описание паттерна диапазона.

[Gang of Four]

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object Oriented Software. Reading, Mass.: Addison-Wesley, 1995.

Вероятно, самая ценная книга по объектно-ориентированному проектированию. Сейчас уже невозможно делать вид, будто понимаешь что-то в объектах, если не умеешь с умным видом говорить о стратегиях, одиночках и цепочке ответственности.

[Jackson, 1993]

Jackson, Michael. Michael Jackson's Beer Guide, Mitchell Beazley, 1993.

Полезное руководство по предмету, который вознаграждает за практическое изучение в большом количестве.

[Java Spec]

Gosling, James, Bill Joy, and Guy Steele. The Java Language Specification. Reading, Mass.: Addison-Wesley, 2000.

Авторитетные ответы на вопросы о Java, хотя было бы желательно обновленное издание.

[JUnit]

Beck, Kent, and Erich Gamma. JUnit Open-Source Testing Framework. Доступна в Интернете на странице автора (http://ourworld.compuser ve.com.homepages/'Martin _Fowler).

Важный инструмент для работы с Java. Простая среда, помогающая писать, организовывать и выполнять тесты модулей. Аналогичные среды есть для Smalltalk и C++.

[Lea]

Lea, Doug. Concurrent Programming in Java: Design Principles and Patterns, Reading, Mass.: Addison-Wesley, 1997.

Компилятор должен останавливать каждого, кто реализует Runnable, если он не прочел эту книгу.

[McConnell]

McConnell, Steve. Code Complete: A Practical Handbook of Software Construction. Redmond, Wash.: Microsoft Press, 1993.

Отличное руководство по стилям программирования и построению программного обеспечения. Написано до появления Java, но почти все советы сохранили ценность.

[Meyer]

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 программированию. Содержит доскональное изложение проектирования по контракту.

[Opdyke]

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. Первый труд приличного объема по рефакторингу. Рассматривает его с несколько академической и ориентированной на инструментальные средства точки зрения (все-таки это диссертация), но вполне достоен прочтения теми, кто интересуется теорией рефакторинга.

[Refactoring Browser]

Brant, John, and Don Roberts. Refactoring Browser Tool,

http://st-www.cs.uiuc.edu/~brant/RefactoringBrowser. Будущее инструментальных средств разработки программного обеспечения.

[Woolf]

Woolf, Bobby. «Null Object.» In Pattern Languages of Program Design 3, edited by R. Martin, F. Buschmann, and D. Riehle. Reading, Mass.: Addi-son-Wesley, 1998.

Обсуждение паттерна нулевого объекта.