Refactoring

Improving the Design of Existing Code

Martin Fowler

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

ADDISON-WESLEY

Рефакторинг

Улучшение существующего кода

Мартин Фаулер

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

Санкт-Петербург
2003

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». 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 70x100 1/16 . Печать офсетная.
Объем 27 печ. л. Тираж 3000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Список рефакторингов

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 (Martin Fowler) и другие авторы, принявшие участие в написании этой книги, внесли большой вклад в разработку объектно-

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

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.

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 Робертс (Don Roberts)** пошли дальше слов и создали инструмент – *Refactoring 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 Апдайк и Дон Робертс 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 Газули (Hisham Ghazouli), Тони Гоулд (Tony Gould), Джон Айзнер (John Isner), Брайен Марик (Brian Marick), Ральф Рейссинг (Ralf Reissing), Джон Солт (John Salt), Марк Свонсон (Mark Swanson), Дейв Томас (Dave Thomas) и Дон Уэллс (Don Wells). 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-Антон Манолеску (Dragos-Anton Manolescu), Хироки Накамура (Hiroaki Nakamura), Джеймс Овертерф (James Overturf), Дон Робертс (Don Roberts), Чико Шираи (Chieko Shirai), Лес Тайрел (Les Tyrell) и Джо Йодер (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).

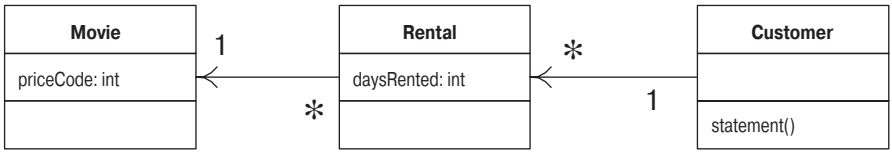

Рис. 1.1. *Диаграмма классов для исходной программы. Нотация соответствует унифицированному языку моделирования [Fowler, UML]*

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

```
 public int getPriceCode() {
     return _priceCode;
 }

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

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

Rental

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) * 1.5;
                    break;
                case Movie.NEW_RELEASE:
                    thisAmount += each.getDaysRented() * 3;
                    break;
                case Movie.CHILDRENS:
                    thisAmount += 1.5;
                    if (each.getDaysRented() > 3)
                        thisAmount += (each.getDaysRented() - 3) * 1.5;
                    break;

            }

            // 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;

    }
```

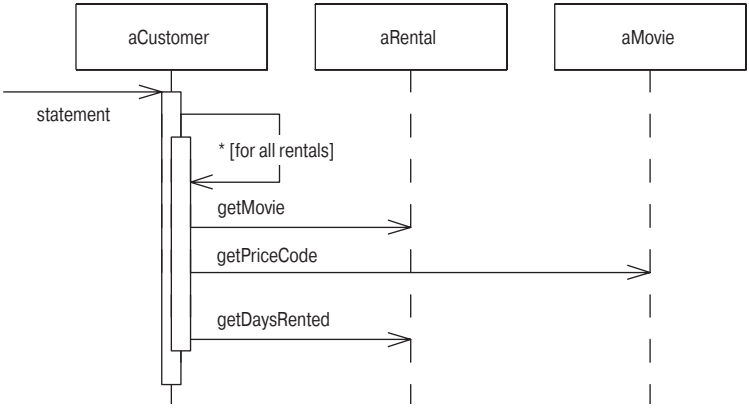

Рис. 1.2. *Диаграмма взаимодействий для метода statement*

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 контрольны-

ми, 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 «OK», 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, 124*). 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, а `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.getDaysRented() > 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().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;
}
  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;

  }
  private int amountFor(Rental each) {
      int thisAmount = 0;
      switch (each.getMovie().getPriceCode()) {
          case Movie.REGULAR:
              thisAmount += 2;
              if (each.getDaysRented() > 2)
                  thisAmount += (each.getDaysRented() - 2) * 1.5;
              break;
          case Movie.NEW_RELEASE:
              thisAmount += each.getDaysRented() * 3;
              break;
          case Movie.CHILDRENS:
              thisAmount += 1.5;
              if (each.getDaysRented() > 3)
                  thisAmount += (each.getDaysRented() - 3) * 1.5;
```

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

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

Такие глупые ошибки 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) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented() - 3) * 1.5;
            break;
    }
    return thisAmount;
}
```

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

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

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, 154*). 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) * 1.5;
              break;
          case Movie.NEW_RELEASE:
              result += getDaysRented() * 3;
              break;
          case Movie.CHILDRENS:
              result += 1.5;
              if (getDaysRented() > 3)
                  result += (getDaysRented() - 3) * 1.5;
              break;
      }
      return result;
  }
```

В 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.valueOf(frequentRenterPoints) +
             " очков за активность";
        return result;

    }
```

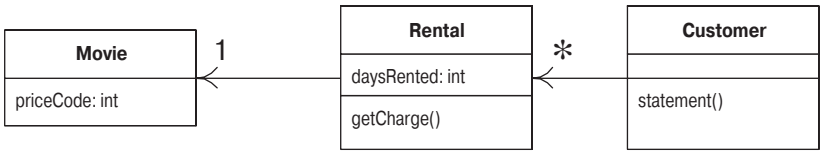

***Рис. 1.3.** Диаграмма классов после перемещения метода, вычисляющего сумму оплаты*

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.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;

    }
```

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, 133*):

```
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» на стр. 80.

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, 124*) 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» (рис. 1.4–1.7). Как обычно, слева находятся диаграммы, отражающие ситуацию до внесения изменений, а справа – после внесения изменений.

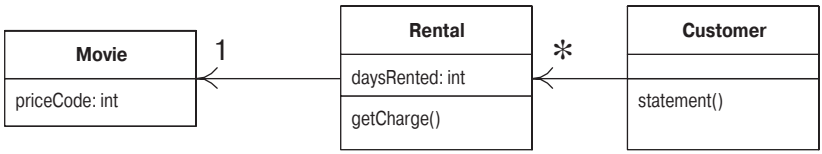

Рис. 1.4. *Диаграмма классов до выделения кода начисления бонусов и его перемещения*

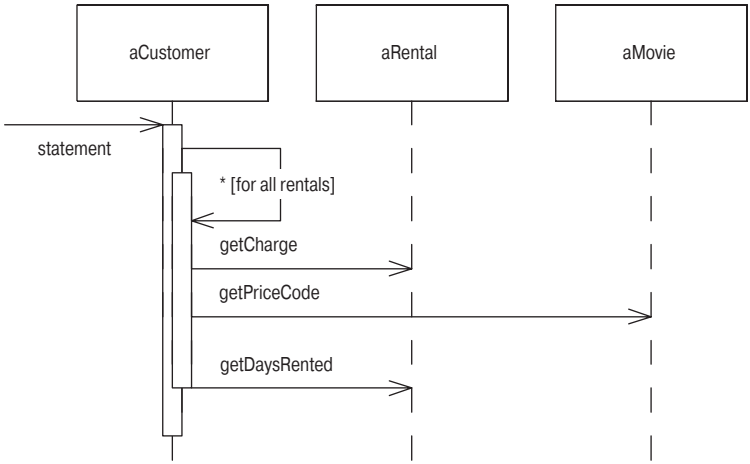

Рис. 1.5. *Диаграмма последовательности до выделения кода начисления бонусов и его перемещения*

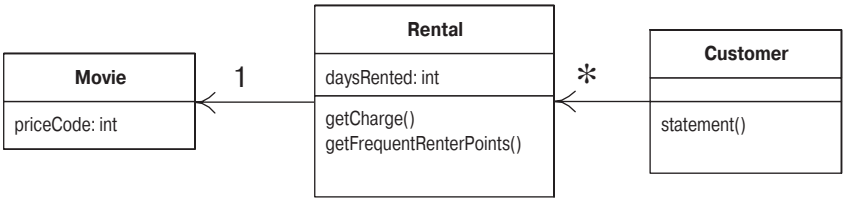

Рис. 1.6. *Диаграмма классов после выделения кода начисления бонусов и его перемещения*

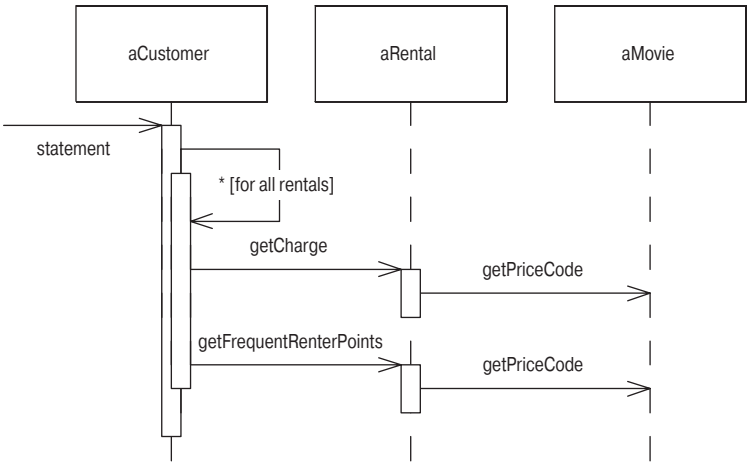

Рис. 1.7. *Диаграмма последовательности после выделения кода начисления бонусов и его перемещения*

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, 133*), чтобы заменить `totalAmount` и `frequentRentalPoints` 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, 133*): 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() {
    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;
}

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

На рис. 1.8–1.11 показаны изменения, внесенные в процессе рефакторинга в диаграммах классов и диаграмме взаимодействия для метода `statement`.

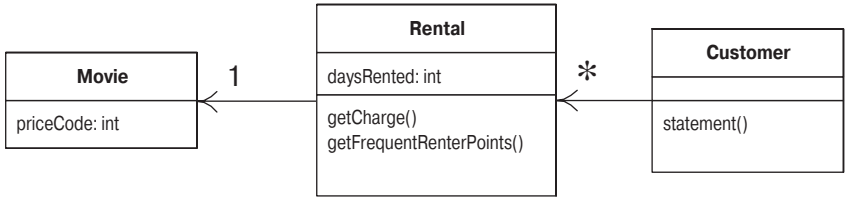

Рис. 1.8. *Диаграмма классов до выделения подсчета итоговых сумм*

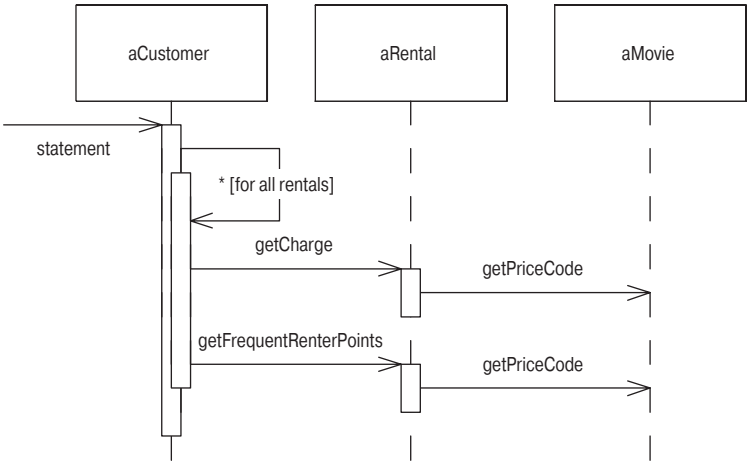

Рис. 1.9. *Диаграмма последовательности до выделения подсчета итоговых сумм*

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

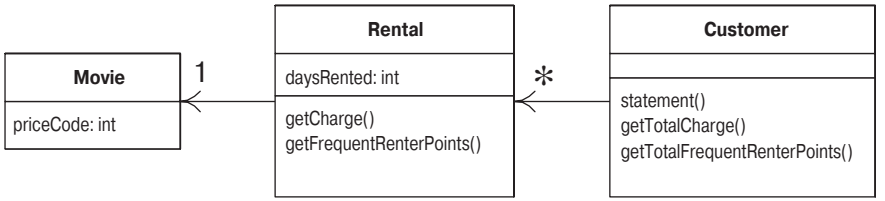

Рис. 1.10. *Диаграмма классов после выделения подсчета итоговых сумм*

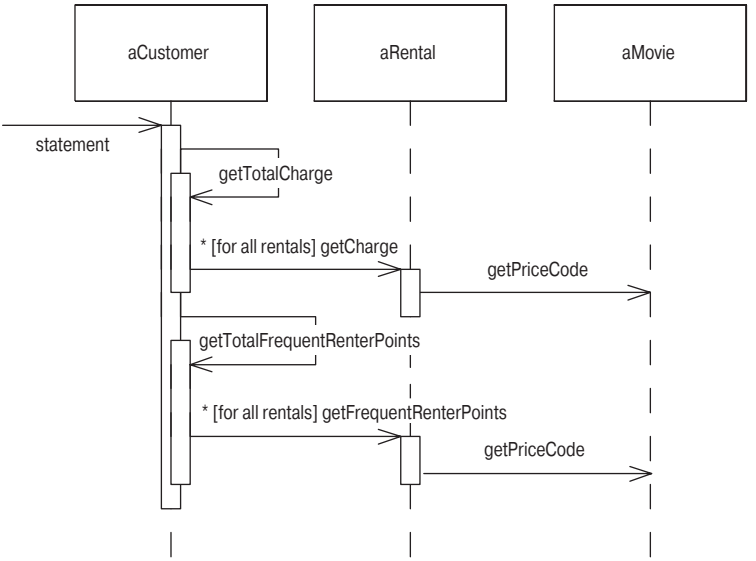

Рис. 1.11. *Диаграмма последовательности после выделения подсчета итоговых сумм*

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 (см. обсуждение на стр. 80).

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, 344*). 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) * 1.5;
              break;
          case Movie.NEW_RELEASE:
              result += getDaysRented() * 3;
              break;
          case Movie.CHILDRENS:
              result += 1.5;
              if (getDaysRented() > 3)
                  result += (getDaysRented() - 3) * 1.5;
              break;
      }
      return result;
  }
```

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

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 (рис. 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;
  }
```

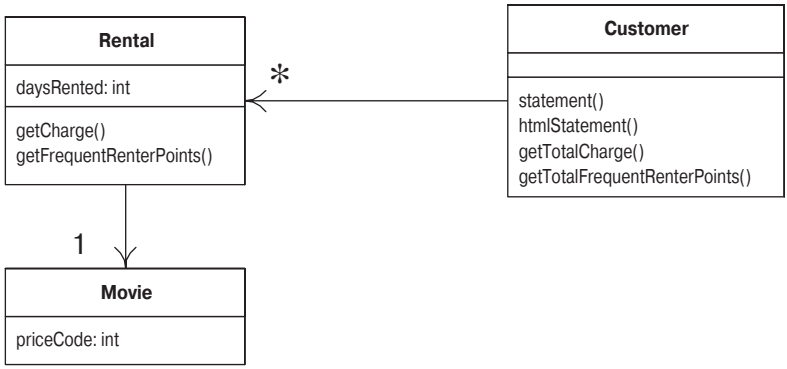

***Рис. 1.12.** Диаграмма классов до перемещения методов в Movie*

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

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

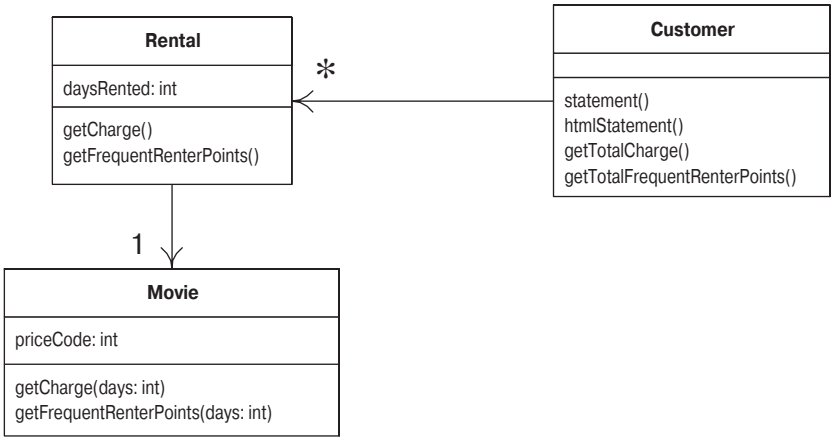

Рис. 1.13. *Диаграмма классов после перемещения методов в Movie*

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 (рис. 1.14).

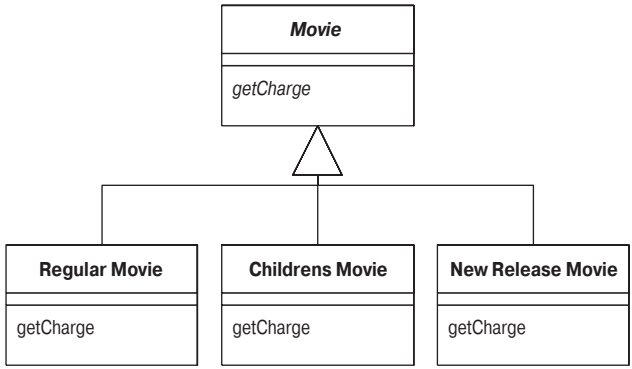

Рис. 1.14. *Применение наследования для Movie*

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 (рис. 1.15):

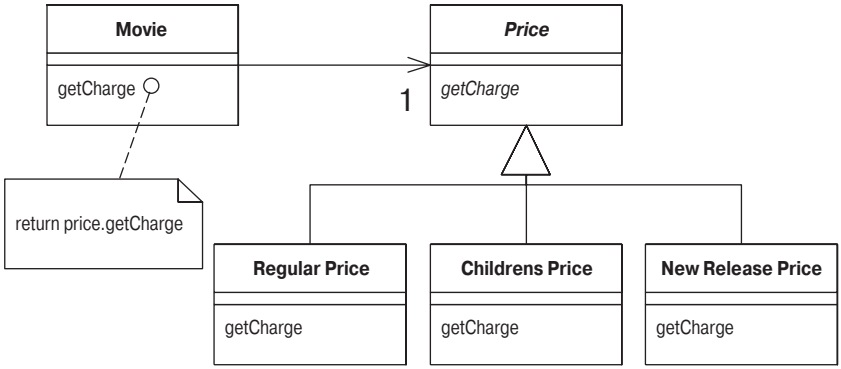

Рис. 1.15. *Использование паттерна «состояние» с movie*

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, 231*). Затем можно применить «Перемещение метода» (*Move Method, 154*), чтобы перенести оператор `switch` в класс `Price`. Наконец, с помощью «Замены условного оператора полиморфизмом» (*Replace Conditional with Polymorphism, 258*) я исключу оператор switch.

Начну с «Замены кода типа состоянием/стратегией» (*Replace Type Code with State/Strategy, 231*). На этом первом шаге к коду типа следует применить «Самоинкапсуляцию поля» (*Self Encapsulate Field, 181*), 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, 154*):

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

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

После перемещения можно приступить к «Замене условного оператора полиморфизмом» (*Replace Conditional with Polymorphism, 258*):

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

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) * 1.5;
        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 = 1.5;
        if (daysRented > 3)
            result += (daysRented - 3) * 1.5;
        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 рис. 1.16 и 1.17 показано, как паттерн «Состояние», который я применил, работает с информацией по ценам.

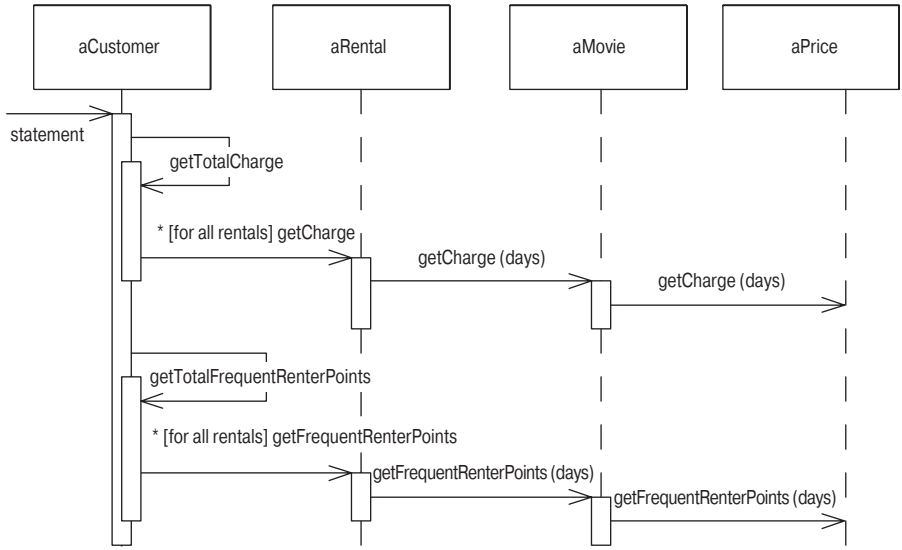

Рис. 1.16. *Диаграмма взаимодействия с применением паттерна «Состояние»*

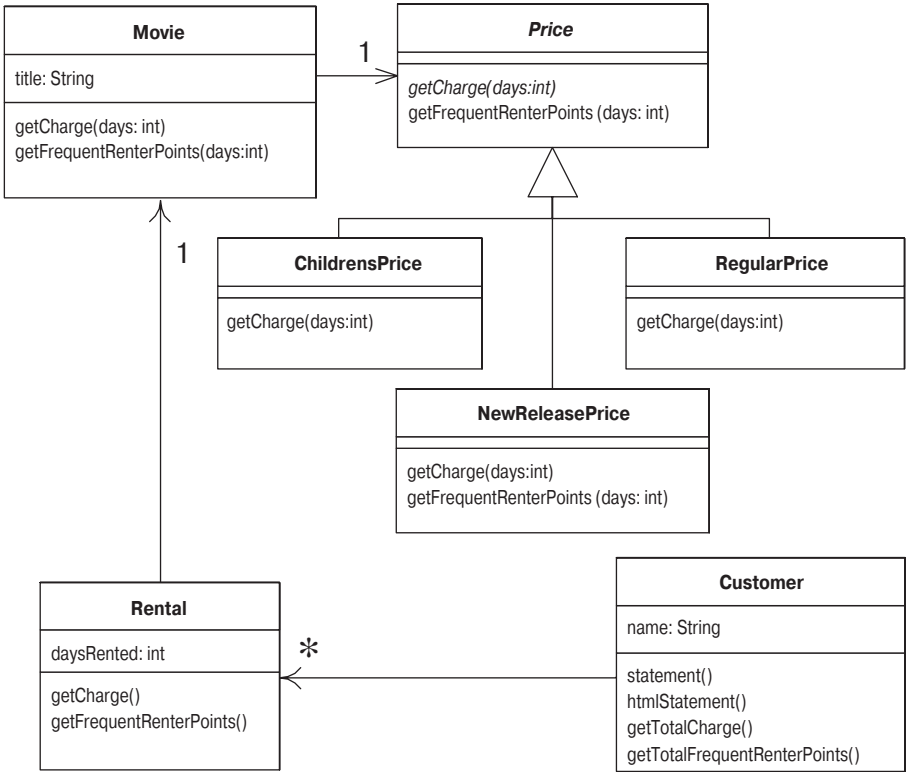

Рис. 1.17. *Диаграмма классов после добавления паттерна «Состояние»*

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, 124*), «Перемещение метода» (*Move Method, 154*) и «Замена условного оператора полиморфизмом» (*Replace Conditional with Polymorphism, 258*). 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).