Эрик Фримен, Элизабет Робсон
при участии Кэти Сьерра, Берта Бейтса

Обновленное юбилейное издание

Head First

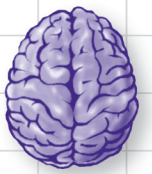
ПАТТЕРНЫ проектирования

Избегайте нелепых ошибок наследования

Узнайте секреты гуру проектирования паттернов

Изучите сотни примеров и практических упражнений

Узнайте, почему все, что вы знали о паттернах, возможно, ошибочно



Усвойте теорию паттернов проектирования максимально эффективно

Посмотрите, как улучшается жизнь программиста благодаря паттернам

O'REILLY®

ПИТЕР®

Отзывы о книге

«Я получил книгу вчера, начал читать ее по дороге домой... и не мог остановиться. Я взял ее в тренажерный зал, и окружающие, вероятно, удивлялись, когда я читал во время тренировки. Круто в высшей степени. Книга отлично читается, но в ней рассматривается вполне серьезный материал, и все по делу. Весьма впечатляюще».

— Эрик Гамма, заслуженный специалист IBM,
соавтор книги «Приемы объектно-ориентированного проектирования», один из участников
«Банды Четырех» наряду с Ричардом Хелмом, Ральфом Джонсоном и Джоном Влиссидесом

«Книга умудряется смешать юмор, техническую глубину и полезнейшие практические советы; получается занимательное чтиво, располагающее к размышлениям. И новички в области паттернов, и опытные разработчики, применявшие их годами, наверняка вынесут что-то полезное из посещения Объективиля».

— Ричард Хелм, соавтор книги «Приемы объектно-ориентированного проектирования»,
вместе с остальными участниками «Банды Четырех» — Эриком Гаммой,
Ральфом Джонсоном и Джоном Влиссидесом

«У меня такое чувство, словно я прочитал сразу полтонны книг».

— Уорд Каннингем, изобретатель Wiki и основатель Hillside Group

«Книга близка к идеалу благодаря сочетанию удобочитаемости и практического опыта. Авторы излагают материал на достойном уровне и делают это изящно. Это одна из немногих книг по программированию, которую я считаю незаменимой (а я к этой категории причисляю книг десять, не более)».

— Дэвид Гелентер, профессор информационных технологий,
Йельский университет, автор книг «Mirror Worlds» и «Machine Beauty»

«Погружение в мир паттернов — в страну, в которой сложное становится простым, но и простое может оказаться сложным. Не представляю себе лучшего вводного руководства, чем книга Фрименов».

— Мико Мацумура, отраслевой аналитик, Middleware Company,
бывший ведущий специалист по Java, Sun Microsystems

«Я смеялся, я плакал, книга тронула меня».

— Дэниел Стейнберг, старший редактор java.net

«Сначала мне захотелось упасть на пол от смеха. Но потом я собрался и понял, что эта книга не только содержит технически точную информацию, но и является самым доступным введением в паттерны проектирования, которое я когда-либо встречал».

— Доктор Тимоти Бадд, адъюнкт-профессор в области информационных технологий
Орегонского государственного университета, автор более дюжины книг,
в том числе «C++ for Java Programmers»

Отзывы о книге

«Джерри Райс обращается с паттернами лучше любого принимающего в NFL, но Фримены превзошли его. Серьезно... Это одна из самых забавных и умных книг в области проектирования ПО, которые я когда-либо читал».

— Аарон Лаберг, старший вице-президент по технологиям и разработке продуктов, ESPN

«Хорошая архитектура программы прежде всего определяется хорошей информационной архитектурой. Проектировщик учит компьютер, как выполнить ту или иную операцию, и не приходится удивляться тому, что хороший учитель компьютеров оказывается хорошим учителем программистов. Благодаря ее доступности, юмору и уму авторов даже не-программист хорошо воспримет эту книгу».

— Кори Доктороу, один из редакторов Boing Boing, автор книг «Down and Out in the Magic Kingdom» и «Someone Comes to Town, Someone Leaves Town»

«Эрик и Элизабет Фримен в своей книге бесстрашно вызвались заглянуть за занавес программного кода. Они излагают основные концепции проектирования на таком честном уровне, на который не решаются многие писатели, думающие только об укреплении своего замечательного эго, — на уровне, на котором открываются столь поразительные истины. Софистам и цирковым зазывалам здесь делать нечего. Образованные люди следующего поколения — не забудьте взять в руки карандаш».

— Кен Голдстейн, исполнительный вице-президент и директор-распорядитель, Disney Online

«Правильно выбранный тон для внутреннего раскрепощенного эксперта-программиста, скрывающегося в каждом из нас. Отличный справочник по практическим стратегиям разработки — мой мозг работает, не отвлекаясь на надоедливый, устаревший академический жаргон».

— Трэвис Каланик, генеральный директор и соучредитель Uber, член MIT TR100

«Благодаря сочетанию юмора, отличных примеров и глубокого знания паттернов проектирования обучение по этой книге становится таким увлекательным занятием. Например, меня как активного участника индустрии развлечений сразу заинтриговал Голливудский принцип и паттерн Фасад для домашнего кинотеатра. Понимание паттернов проектирования не только помогает нам создавать качественные программы, пригодные для повторного использования, но и совершенствует наши навыки решения задач во всех предметных областях. Эта книга рекомендуется всем профессионалам и студентам в области компьютерных технологий».

— Ньютон Ли, основатель и старший редактор сайта acmcie.org (Association for Computing Machinery / Computers in Entertainment)

Отзывы о книге

«Если и есть тема, преподавание которой определенно требует большей занимательности, то это паттерны проектирования. К счастью, у нас теперь есть эта книга.

Великолепные авторы «Head First Java» используют все мыслимые приемы, чтобы помочь — вам понять и запомнить материал. Здесь вы найдете не только множество изображений людей, которые привлекают внимание других людей. Сюрпризы повсюду! Многочисленные истории (например, о пицце и шоколаде. Стоит ли говорить еще?). Вдобавок книга невероятно смешная.

В ней представлены множество концепций и приемов, а также почти все паттерны, которые чаще всего используются на практике: Наблюдатель, Декоратор, Фабрика, Одиночка, Команда, Адаптер, Фасад, Шаблонный Метод, Итератор, Компоновщик, Состояние, Заместитель. Прочитайте, и все они перестанут быть «просто словами», превратившись в воспоминания, которые задевают вас за живое, и инструменты, применяемые в повседневной работе».

— Билл Камарда, READ ONLY

«После использования «Head First Java» для обучения азам программирования я с нетерпением ждал следующего издания из этой серии. Я уверен, что данная книга быстро станет первой книгой, с которой следует начинать знакомство с паттернами, — и она уже стала книгой, которую я рекомендую своим студентам».

— Бен Бедерсон, адъюнкт-профессор в области информационных технологий, директор лаборатории взаимодействий «человек–компьютер» в Мэрилендском университете

«Обычно во время чтения книги или статьи, посвященных паттернам программирования, мне приходится время от времени щипать себя, чтобы убедиться в том, что я еще не заснул. С этой книгой все совершенно иначе. Как ни странно, она делает изучение паттернов легким и веселым занятием».

— Эрик Вулер

«Я буквально влюблен в эту книгу. Я даже поцеловал ее на глазах у жены».

— Сатиш Кумар

Отзывы о технологии «*Head First*»

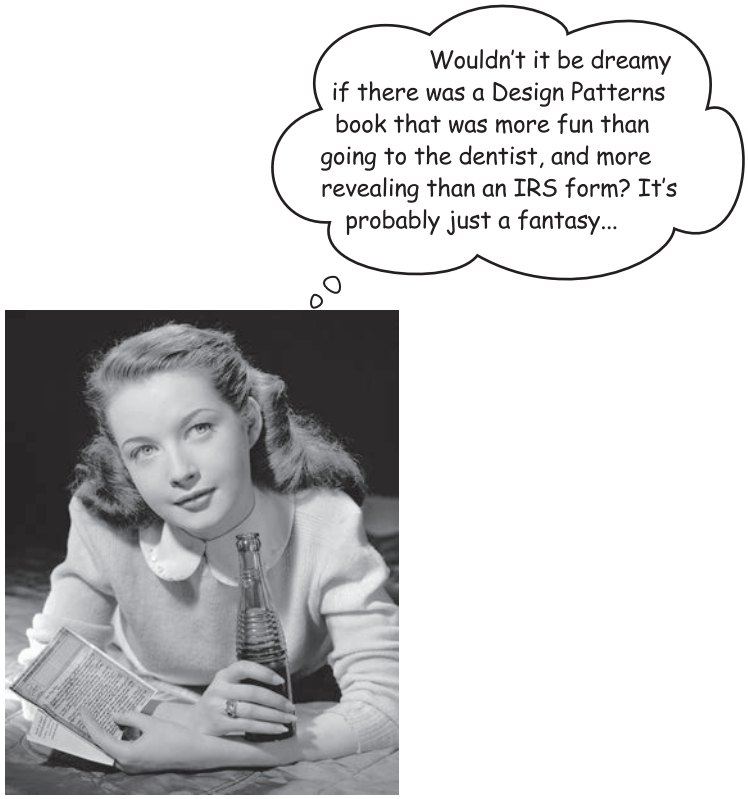
«Технология Java повсюду: в мобильных телефонах, в машинах, фотоаппаратах, принтерах, играх, КПК, банкоматах, смарт-картах, бензонасосах, на стадионах, в медицинском оборудовании, веб-камерах, серверах... Если вы занимаетесь программированием, но еще не изучили Java, вам определенно стоит сделать это с книгой Head First».

— Скотт Макнили, председатель совета директоров Sun Microsystems, президент и исполнительный директор

«Книга читается быстро, она несерьезная, веселая и увлекательная. Будьте внимательны — из нее легко что-нибудь узнать!»

— Кен Арнольд, бывший старший специалист в Sun Microsystems, соавтор книги «Язык программирования Java» (написанной вместе с Джеймсом Гослингом, создателем Java).

Head First
Design Patterns



Eric Freeman
Elisabeth Robson

O'REILLY®

Beijing • Cambridge • Köln • Sebastopol • Tokyo

Head First
Паттерны проектирования
Обновленное юбилейное издание

Эрик Фримен,
Элизабет Робсон

при участии
Кэтти Сьерра
и Берта Бейтса

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2018

ББК 32.973.2-018-02
УДК 004.42
Х99

Фримен Э., Робсон Э., Сьерра К., Бейтс Б.
Х99 Head First. Паттерны проектирования. Обновленное юбилейное издание. — СПб.: Питер, 2018. — 656 с.: ил. — (Серия «Head First O'Reilly»).

ISBN 978-5-496-03210-0

В мире постоянно кто-то сталкивается с такими же проблемами программирования, которые возникают и у вас. Многие разработчики решают совершенно идентичные задачи и находят похожие решения. Если вы не хотите изобретать велосипед, используйте готовые шаблоны (паттерны) проектирования, работе с которыми посвящена эта книга.

Паттерны появились, потому что многие разработчики искали пути повышения гибкости и степени повторного использования своих программ. Найденные решения воплощены в краткой и легко применимой на практике форме.

Особенностью данного издания является уникальный способ подачи материала, выделяющий серию «Head First» издательства O'Reilly в ряду множества скучных книг, посвященных программированию. Книга будет интересна широкому кругу веб-разработчиков, от начинающих до профессионалов, желающих освоить работу с паттернами проектирования.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018-02
УДК 004.42

ISBN 978-0596007126 англ. Authorized Russian translation of the English edition of Head First Design Patterns, ISBN 9780596007126

ISBN 978-5-496-03210-0

Посвящается «Банде Четырех»; их прозорливость и мастерство в формулировке и описании паттернов проектирования навсегда изменили область проектирования программных архитектур и улучшили жизнь разработчиков во всем мире.

Ну сколько можно ждать, когда выйдет второе издание? В конце концов, прошло уже ~~*десять*~~ *двадцать лет*!

Авторы/разработчики книги

Эрик Фримен

Элизабет Робсон

Эрик, по словам Кэти Сьерра, соавтора серии Head First, — «один из редких людей, хорошо разбирающихся в языке, практике применения и культуре из самых разных областей — технохипстер, вице-президент, инженер, аналитик».

Эрик отработал почти десять лет на должности технического директора Disney Online & Disney.com в компании The Walt Disney Company. Сейчас Эрик отдает свое время WickedlySmart — молодой компании, которую он создал совместно с Элизабет.

По образованию Эрик является специалистом по компьютерным технологиям; он занимался исследованиями вместе с корифеем отрасли Дэвидом Гелернтером во время работы над диссертацией в Йельском университете. Его диссертация стала одним из фундаментальных трудов в области альтернатив для интерфейсов, реализующих метафору рабочего стола, а также первой реализацией потоков активности — концепции, разработанной им совместно с доктором Гелернтером.

В свободное время Эрик серьезно занимается музыкой; последний проект Эрика, созданный вместе с одним из пионеров направления «амбиент» Стивом Роучем, доступен в магазине iPhone App Store под названием Immersion Station.

Эрик живет с женой и маленькой дочкой в Остине (штат Техас). Его дочь часто наведывается в студию Эрика; ей нравится крутить рукоятки синтезаторов.

Ему можно написать по адресу eric@wickedlysmart.com или посетить его сайт ericfreeman.com.

Элизабет — программист, писатель и преподаватель. Она влюблена в свою работу еще с времен учебы в Йельском университете, где получила степень магистра в области компьютерных технологий, а также создала параллельный визуальный язык программирования и программную архитектуру.

Элизабет участвовала в создании популярного сайта Ada Project — одного из первых, помогающих найти женщинам информацию о работе и образовании в области компьютерных технологий.

Она стала одним из учредителей WickedlySmart — компании, работающей в области интернет-образования на базе веб-технологий. Здесь она пишет книги, статьи, создает видеокурсы и т. д. Ранее Элизабет занимала должность директора по специальным проектам в O'Reilly Media и разрабатывала семинары и курсы дистанционного обучения по разным техническим темам, помогающие людям разобраться в новых технологиях. До прихода в O'Reilly Элизабет работала в The Walt Disney Company, где она руководила исследованиями и разработками в сфере цифровых мультимедийных технологий.

Когда Элизабет не сидит за компьютером, она занимается велоспортом и греблей, фотографирует или готовит вегетарианские блюда.

С ней можно связаться по адресу beth@wickedlysmart.com или посетить ее блог http://elisabethrobson.com.

Создатели серии Head First (и соавторы книги)

Кэтти интересовалась теорией обучения еще в те времена, когда она создавала игры для Virgin, MGM и Amblin'). Большая часть формата Head First была разработана ею во время ведения курса New Media Authoring для программы UCLA Extension's Entertainment Studies. В последнее время она готовит специалистов для Sun Microsystems, учит инструкторов Java искусству преподавания новейших Java-технологий; участвовала в разработке нескольких сертификационных экзаменов Sun. Вместе с Бертом Бейтсом использовала концепции Head First для обучения тысяч разработчиков. Кэтти является учредителем сайта javaranch.com, который в 2003 и 2004 годах завоевал награды журнала Software Development. Иногда преподает Java на курсах Java Jam Geek Cruise (geekcruises.com).

Недавно Кэтти переехала из Калифорнии в Колорадо. Здесь ей пришлось учить новые слова: «заморозки», «плед» и другие, но зато пейзажи просто потрясающие.

Любит: бег, лыжи, скейтборд, свою исландскую лошадку и науку. Не любит: энтропию.

Часто бывает на javaranch, иногда ведет блог на seriouspony.com. Пишите ей по адресу kathy@wickedlysmart.com.

Берт — опытный программист и проектировщик, но его десятилетние изыскания в области искусственного интеллекта вызвали интерес к теории и технологии обучения. С тех пор он занимается повышением квалификации программистов. В последнее время возглавлял группу разработки сертификационных экзаменов по языку Java для корпорации Sun.

Первое десятилетие своей карьеры программиста Берт путешествовал по всему миру, помогая СМИ — таким, как Radio New Zealand, Weather Channel и Arts & Entertainment Network (A & E). Одним из его любимых проектов того времени стало построение полного имитатора железнодорожной сети для Union Pacific Railroad.

Берт — заядлый поклонник игры *го* и давно работает над ее программированием. Неплохо играет на гитаре, также пробует свои силы с банджо.

Его можно встретить на javaranch или на го-сервере IGS, можно написать ему по адресу terrapin@wickedlysmart.com.

Содержание (сводка)

Содержание (настоящее)

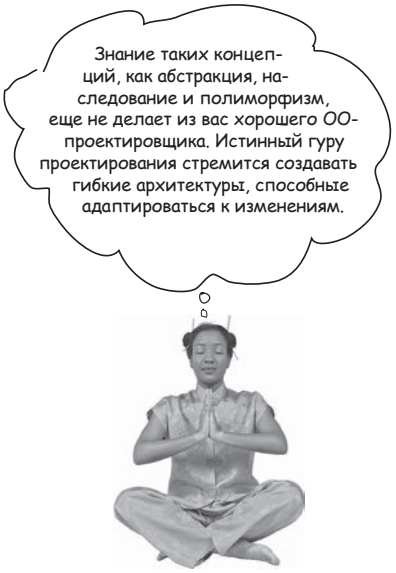
Введение

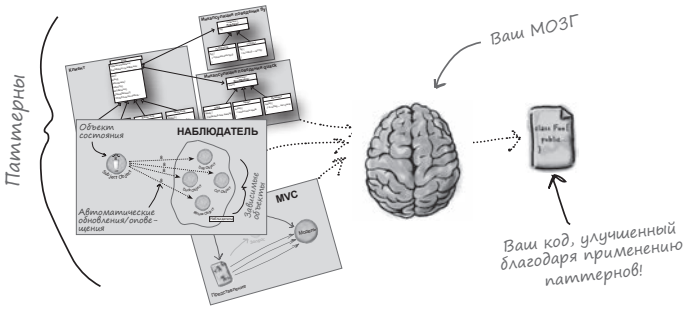
Настройте свой мозг на дизайн паттернов. Вот что вам понадобится, когда вы пытаетесь что-то выучить, в то время как ваш мозг не хочет воспринимать информацию. Ваш мозг считает: «Лучше уж я подумаю о более важных вещах, например об опасных диких животных или почему нельзя голышом прокатиться на сноуборде». Как же заставить свой мозг думать, что ваша жизнь зависит от овладения дизайном паттернов?

1 Знакомство с паттернами

Добро пожаловать в мир паттернов

Наверняка вашу задачу кто-то уже решал. В этой главе вы узнаете, почему (и как) следует использовать опыт других разработчиков, которые уже сталкивались с аналогичной задачей и успешно решили ее. Заодно мы поговорим об использовании и преимуществах паттернов проектирования, познакомимся с ключевыми принципами ООП и разберем пример одного из паттернов. Лучший способ использовать паттерны — *запомнить их*, а затем научиться *распознавать* те места ваших архитектур и существующих приложений, где их уместно *применить*. Таким образом, вместо программного кода вы повторно используете чужой *опыт*.

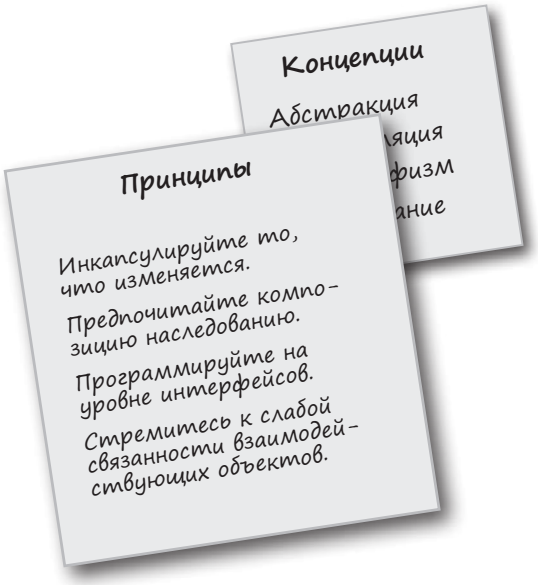


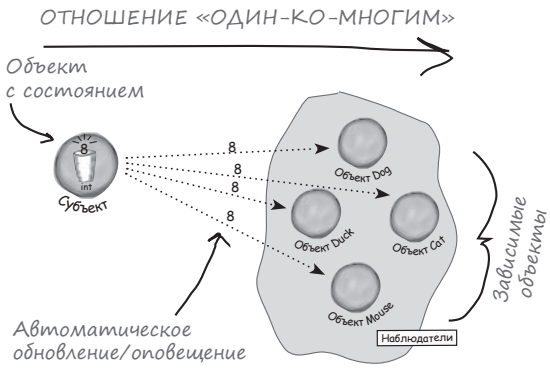


2 Паттерн Наблюдатель

Объекты в курсе событий

Не упустите, когда происходит что-то интересное! Наш следующий паттерн оповещает объекты о наступлении неких событий, которые могут представлять для них интерес, — причем объекты даже могут решать во время выполнения, желают ли они и дальше получать информацию. Паттерн Наблюдатель чрезвычайно полезен и принадлежит к числу наиболее часто используемых паттернов JDK. Также в этой главе будут рассмотрены связи типа «один-ко-многим» и слабые связи. С паттерном Наблюдатель вы станете душой Общества Паттернов.

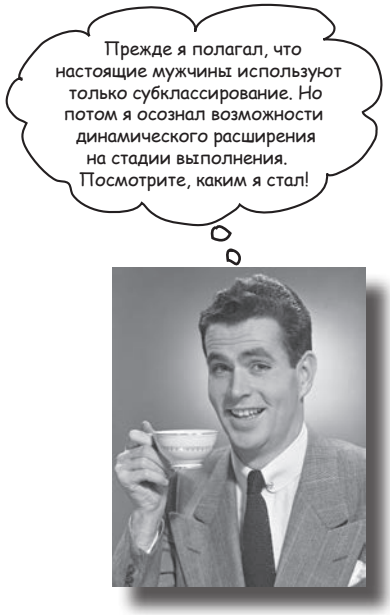




3 Паттерн Декоратор

Украшение объектов

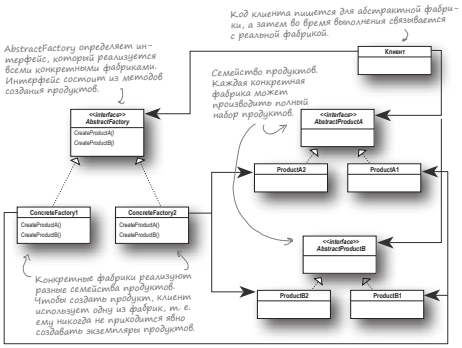
Эту главу можно назвать «Взгляд на архитектуру для любителей наследования». Мы проанализируем типичные злоупотребления из области наследования, и вы научитесь декорировать свои классы во время выполнения с использованием разновидности композиции. Зачем? Затем, что этот прием позволяет вам наделить свои (или чужие) объекты новыми возможностями *без модификации кода классов*.



4 Паттерн Фабрика

Домашняя ОО-выпечка

Приготовьтесь заняться выпечкой объектов в слабосвязанных ОО-архитектурах. Создание объектов отнюдь не сводится к простому вызову оператора *new*. Оказывается, создание экземпляров не всегда должно осуществляться открыто; оно часто создает проблемы *сильного связывания*. А ведь вы *этого* не хотите, верно? Паттерн Фабрика спасет вас от неприятных зависимостей.



5 Паттерн Одиночка

Уникальные объекты

Паттерн Одиночка направлен на создание уникальных объектов, существующих только в одном экземпляре. Из всех паттернов Одиночка имеет самую простую диаграмму классов; собственно, вся диаграмма состоит из одного-единственного класса! Однако не стоит расслабляться; несмотря на всю его простоту с точки зрения архитектуры классов, в его реализации кроется немало ловушек. Так что пристегните ремни!

6 Паттерн Команда

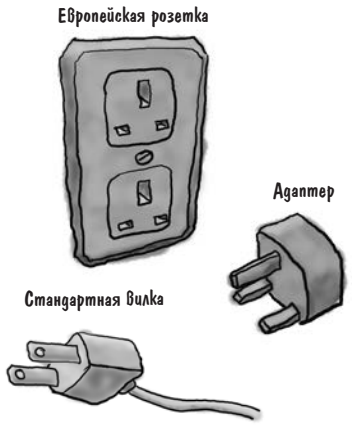
Инкапсуляция вызова

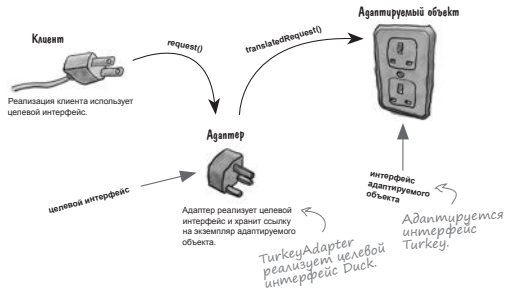
В этой главе мы выходим на новый уровень инкапсуляции — на этот раз будут инкапсулироваться вызовы методов. Да, все верно — вызывающему объекту не нужно беспокоиться о том, как будут выполняться его запросы. Он просто использует инкапсулированный метод для решения своей задачи. Инкапсуляция позволяет решать и такие нетривиальные задачи, как регистрация или отмена вызовов.

7 Паттерны Адаптер и Фасад

Умение приспосабливаться

В этой главе мы займемся всякими невозможными трюками — будем затыкать круглые дырки квадратными пробками. Невозможно, скажете вы? Только не с паттернами проектирования. Помните паттерн Декоратор? Мы «упаковывали» объекты, чтобы расширить их возможности. А в этой главе мы займемся упаковкой объектов с другой целью: чтобы имитировать интерфейс, которым они в действительности не обладают. Для чего? Чтобы адаптировать архитектуру, рассчитанную на один интерфейс, для класса, реализующего другой интерфейс. Но и это еще не все; попутно будет описан другой паттерн, в котором объекты упаковываются для упрощения их интерфейса.





8 Паттерн Шаблонный Метод

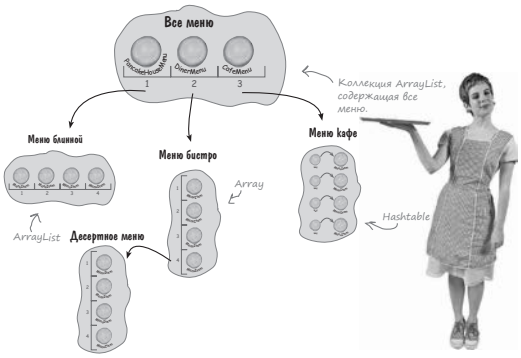
Инкапсуляция алгоритмов

Мы уже «набили руку» на инкапсуляции; мы инкапсулировали *создание объектов*, *вызовы методов*, *сложные интерфейсы*, *уток*, *пиццу*... Что дальше? Следующим шагом будет инкапсуляция алгоритмических блоков, чтобы субклассы могли в любой момент связаться с нужным алгоритмом обработки. В этой главе даже будет описан принцип проектирования, вдохновленный Голливудской практикой.

9 Паттерны Итератор и Компоновщик

Управляемые коллекции

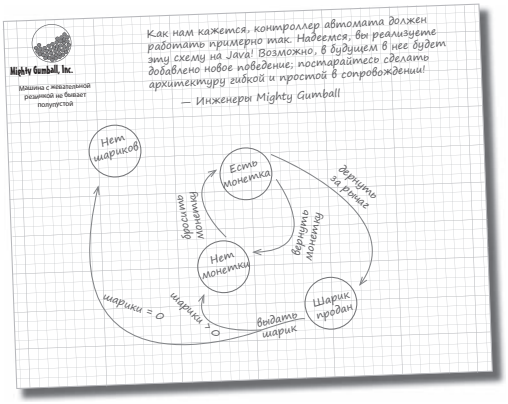
Существует много способов создания коллекций. Объекты можно разместить в контейнере Array, Stack, List, Hashtable — выбирайте сами. Каждый способ обладает своими достоинствами и недостатками. Но в какой-то момент клиенту потребуется перебрать все эти объекты, и, когда это произойдет, собираетесь ли вы раскрывать реализацию коллекции? Надеемся, нет! Это было бы крайне непрофессионально. В этой главе вы узнаете, как предоставить клиенту механизм перебора объектов без раскрытия информации о способе их хранения. Также в ней будут описаны способы создания суперколлекций. А если этого недостаточно, вы узнаете кое-что новое относительно обязанностей объектов.



10 Паттерн Состояние

Состояние дел

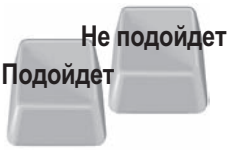
Малоизвестный факт: паттерны Стратегия и Состояние — близнецы, разлученные при рождении. Как известно, паттерн Стратегия организовал успешный бизнес в области взаимозаменяемых алгоритмов. Паттерн Состояние выбрал, пожалуй, более благородный путь: он помогает объектам управлять своим поведением посредством изменения внутреннего состояния.

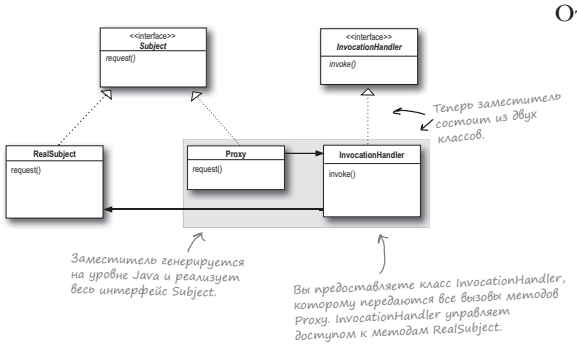


11 Паттерн Заместитель

Управление доступом к объектам

Когда-нибудь разыгрывали сценку «хороший полицейский, плохой полицейский»? Вы — «хороший полицейский», вы общаетесь со всеми любезно и по-дружески, но не хотите, чтобы все обращались к вам за каждым пустяком. Поэтому вы обзаводитесь «плохим полицейским», который управляет доступом к вам. Именно этим и занимаются заместители: они управляют доступом. Как вы вскоре увидите, способы взаимодействия заместителей с обслуживаемыми объектами. Иногда заместители пересылают по Интернету целые вызовы методов, а иногда просто терпеливо стоят на месте, изображая временно отсутствующие объекты.

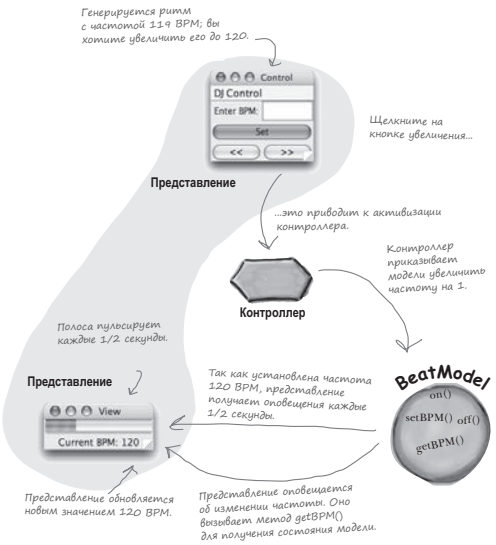




12 Составные паттерны

Паттерны паттернов

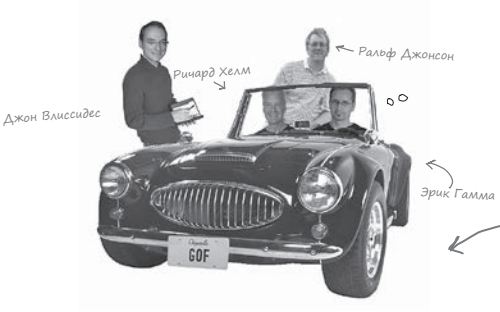
Кто бы мог предположить, что паттерны порой работают рука об руку? Вы уже были свидетелями ожесточенных перепалок в «Беседах у камина» (причем вы не видели «Смертельные поединки» паттернов — редактор заставил нас исключить их из книги!). И после этого оказывается, что мирное сосуществование все же возможно. Хотите верьте, хотите нет, но некоторые из самых мощных ОО-архитектур строятся на основе комбинаций нескольких паттернов.



13 Паттерны для лучшей жизни

Паттерны в реальном мире

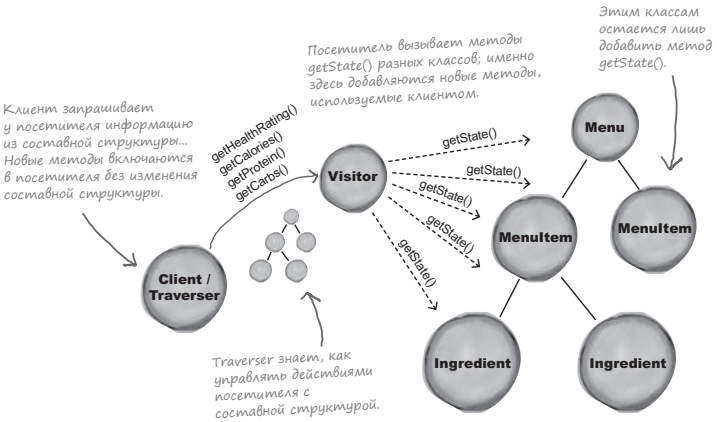
Вы стоите на пороге дивного нового мира, населенного паттернами проектирования. Но прежде чем открывать дверь, желательно изучить некоторые технические тонкости, с которыми вы можете столкнуться — в реальном мире жизнь немного сложнее, чем здесь, в Объективиле. К счастью, у вас имеется хороший путеводитель, который упростит ваши первые шаги...



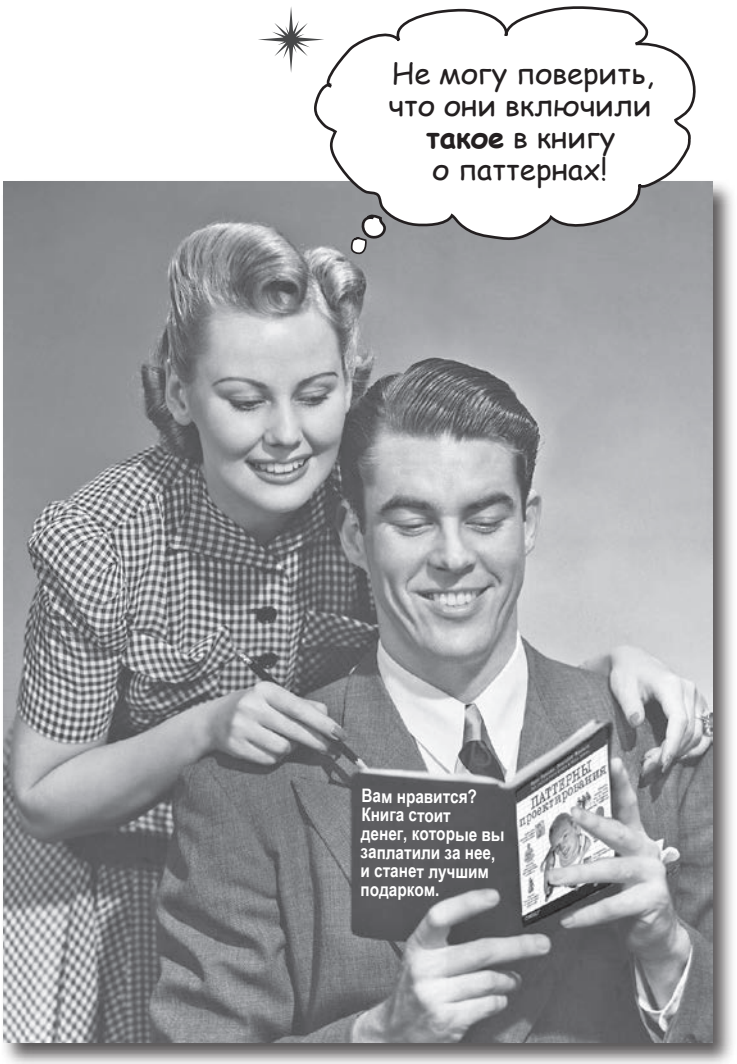
«Банда Четырех»

14 Приложение: Другие паттерны

Не каждому суждено оставаться на пике популярности. За последние 10 лет многое изменилось. С момента выхода первого издания книги «Банды Четырех» разработчики тысячи раз применяли эти паттерны в своих проектах. В этом приложении представлены полноценные, первосортные паттерны от «Банды Четырех» — они используются реже других паттернов, которые рассматривались ранее. Однако эти паттерны ничем не плохи, и если они уместны в вашей ситуации — применяйте их без малейших сомнений. В этом приложении мы постараемся дать общее представление о сути этих паттернов.



Введение



В этом разделе мы ответим на насущный вопрос: «Так почему они включили ТАКОЕ в книгу о паттернах?»

Для кого написана эта книга?

Если вы ответите «да» на все следующие вопросы...

1. **Вы знаете Java? (Быть знатоком не обязательно.)**

Вероятно, вместо Java подойдет и C#.

2. Вы хотите **изучить**, **понять**, **запомнить** и ***применять*** паттерны, а также принципы ОО-проектирования, на которых основаны паттерны?

3. **Вы предпочитаете оживленную беседу сухим, скучным академическим лекциям?**

...то эта книга для вас.

Кому эта книга не подойдет?

Если вы ответите «да» на любой из следующих вопросов...

1. **Вы абсолютно не знаете Java?**

   (Быть знатоком не обязательно. Причем даже если вы *не знаете* Java, но владеете C#, вероятно, вы поймете не менее 80% примеров. *Возможно*, подойдет и опыт программирования на C++.)

2. Вы — крутой ОО-проектировщик/разработчик, которому нужен ***справочник***?

3. Вы занимаетесь архитектурным проектированием и ищете информацию о паттернах ***корпоративного*** уровня?

4. Вы **боитесь попробовать что-нибудь новое**? Скорее пойдете к зубному врачу, чем наденете полосатое с клетчатым? Считаете, что техническая книга, в которой компоненты Java изображены в виде человечков, серьезной быть не может?

...эта книга не для вас.

[Заметка от отдела продаж: вообще-то эта книга для любого, у кого есть деньги.]

Мы знаем, о чем вы думаете

«Разве серьезные книги по программированию такие?»

«И почему здесь столько рисунков?»

«Можно ли так чему-нибудь научиться?»

И мы знаем, о чем думает ваш *мозг*

Мозг жаждет новых впечатлений. Он постоянно ищет, анализирует, *ожидает* чего-то необычного. Он так устроен, и это помогает нам выжить.

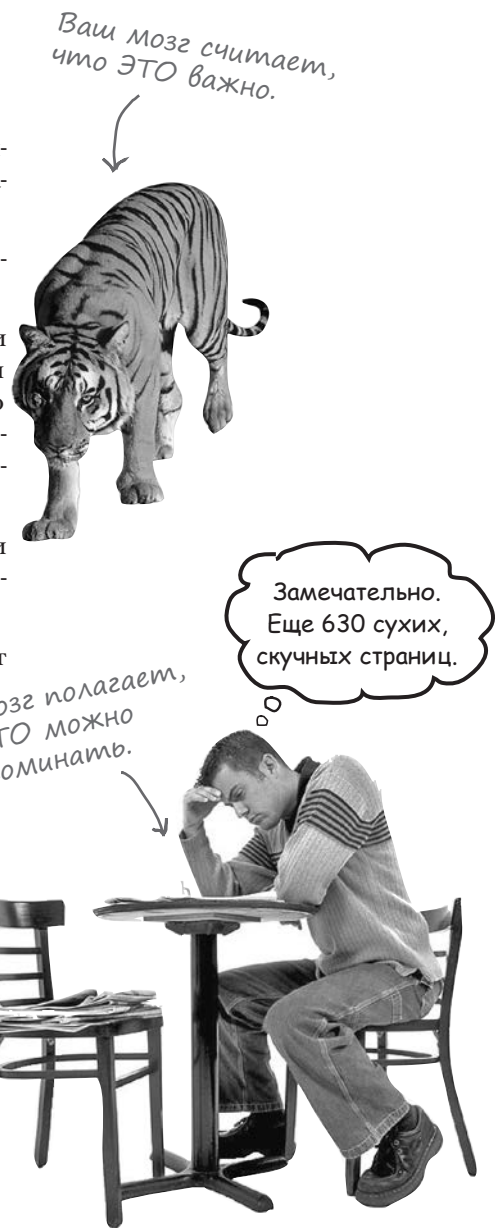
В наши дни вы вряд ли попадете на обед к тигру. Но наш мозг постоянно остается настороже. Просто мы об этом не знаем.

Как же наш мозг поступает со всеми обычными, повседневными вещами? Он всеми силами пытается оградиться от них, чтобы они не мешали его *настоящей* работе — сохранению того, что действительно *важно*. Мозг не считает нужным сохранять скучную информацию. Она не проходит фильтр, отсекающий «очевидно несущественное».

Но как же мозг *узнает*, что важно? Представьте, что вы выехали на прогулку, и вдруг прямо перед вами появляется тигр. Что происходит в вашей голове и в теле?

Активизируются нейроны. Вспыхивают эмоции. Происходят химические реакции.

И тогда ваш мозг понимает...



Конечно, это важно! Не забывать!

А теперь представьте, что вы находитесь дома или в библиотеке, в теплом, уютном месте, где тигры не водятся. Вы учитесь — готовитесь к экзамену. Или пытаетесь освоить сложную техническую тему, на которую вам выделили неделю... максимум десять дней.

И тут возникает проблема: ваш мозг пытается оказать вам услугу. Он старается сделать так, чтобы на эту *очевидно* несущественную информацию не тратились драгоценные ресурсы. Их лучше потратить на что-нибудь важное. На тигров, например. Или на то, что к огню лучше не прикасаться. Или что на лыжах не стоит кататься в футболке и шортах.

Нет простого способа сказать своему мозгу: «Послушай, мозг, я тебе, конечно, благодарен, но какой бы скучной ни была эта книга, и пусть мой датчик эмоций сейчас на нуле, я *хочу* запомнить то, что здесь написано».

Эта книга для тех, кто хочет учиться.

Как мы что-то *узнаем*? Сначала нужно это «что-то» *понять*, а потом *не забыть*. Затолкать в голову побольше фактов недостаточно. Согласно новейшим исследованиям в области когнитивистики, нейробиологии и психологии обучения, для *усвоения материала* требуется что-то большее, чем простой текст на странице. Мы знаем, как заставить ваш мозг работать.

Основные принципы серии «Head First»:

Наглядность. Графика запоминается гораздо лучше, чем обычный текст, и значительно повышает эффективность восприятия информации (до 89% по данным исследований). Кроме того, материал становится более понятным. **Текст размещается на рисунках**, к которым он относится, а не под ними или на соседней странице.

Разговорный стиль изложения. Недавние исследования показали, что при личном разговорном стиле изложения материала (вместо формальных лекций) улучшение результатов на итоговом тестировании составляло до 40%. Рассказывайте историю вместо того, чтобы читать лекцию. Не относитесь к себе слишком серьезно. Что скорее привлечет ваше внимание: занимательная беседа за столом или лекция?

Плохо быть абстрактным методом. Приходится обходиться без тела.

```
abstract void roam();
```

У метода нет тела! Не забудьте поставить точку с запятой.

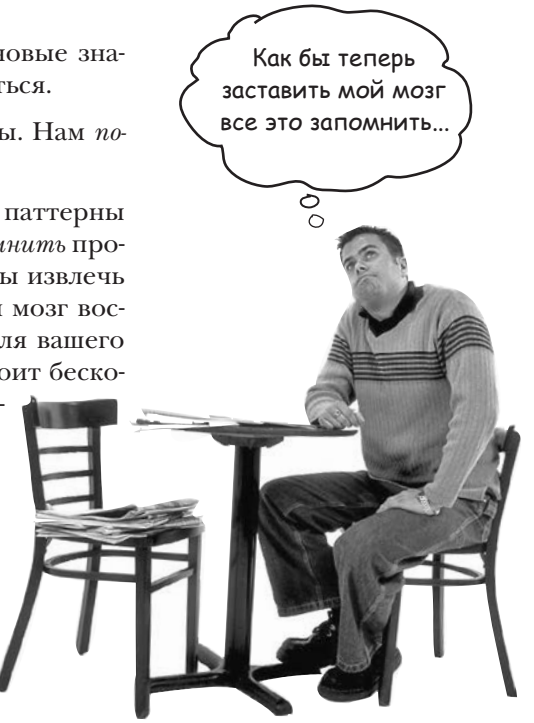
Активное участие читателя. Пока вы не начнете напрягать извилины, в вашей голове ничего не произойдет. Читатель должен быть заинтересован в результате; он должен решать задачи, формулировать выводы и овладевать новыми знаниями. А для этого необходимы упражнения и каверзные вопросы, в решении которых задействованы оба полушария мозга и разные чувства.

Можно ли сказать, что мыло СОДЕРЖИТ пену? Или это отношение типа «является частным случаем»?

Привлечение (и сохранение) внимания читателя. Ситуация, знакомая каждому: «Я очень хочу изучить это, но засыпаю на первой странице». Мозг обращает внимание на интересное, странное, притягательное, неожиданное. Изучение сложной технической темы не обязано быть скучным. Интересное узнается намного быстрее.

Обращение к эмоциям. Известно, что наша способность запоминать в значительной мере зависит от эмоционального сопереживания. Мы запоминаем то, что нам небезразлично. Мы запоминаем, когда что-то чувствуем. Нет, сентименты здесь ни при чем: речь идет о таких эмоциях, как удивление, любопытство, интерес и чувство «Да я крут!» при решении задачи, которую окружающие считают сложной, — или когда вы понимаете, что разбираетесь в теме лучше, чем всезнайка Боб из технического отдела.

Метапознание: наука о мышлении



Если вы действительно хотите быстрее и глубже усваивать новые знания — задумайтесь над тем, как вы задумываетесь. Учитесь учиться.

Мало кто из нас изучает теорию метапознания во время учебы. Нам *положено* учиться, но нас редко этому *учат*.

Но раз вы читаете эту книгу, то, вероятно, вы хотите изучить паттерны проектирования, и по возможности быстрее. Вы хотите *запомнить* прочитанное и *применять* новую информацию на практике. Чтобы извлечь максимум пользы из учебного процесса, нужно заставить ваш мозг воспринимать новый материал как Нечто Важное. Критичное для вашего существования. Такое же важное, как тигр. Иначе вам предстоит бесконечная борьба с вашим мозгом, который всеми силами уклоняется от запоминания новой информации.

Как же УБЕДИТЬ мозг, что паттерны проектирования так же важны, как и тигр?

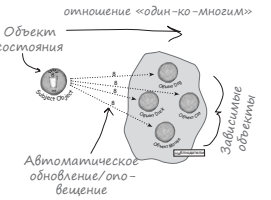
Есть способ медленный и скучный, а есть быстрый и эффективный. Первый основан на тупом повторении. Всем известно, что даже самую скучную информацию *можно* запомнить, если повторять ее снова и снова. При достаточном количестве повторений ваш мозг прикидывает: «*Вроде бы* несущественно, но раз одно и то же повторяется *столько раз*... Ладно, уговорил».

Быстрый способ основан на **повышении активности мозга**, и особенно на сочетании разных ее *видов*. Доказано, что все факторы, перечисленные на предыдущей странице, помогают вашему мозгу работать на вас. Например, исследования показали, что размещение слов *внутри* рисунков (а не в подписях, в основном тексте и т. д.) заставляет мозг анализировать связи между текстом и графикой, а это приводит к активизации большего количества нейронов. Больше нейронов = выше вероятность того, что информация будет сочтена важной и достойной запоминания.

Разговорный стиль тоже важен: обычно люди проявляют больше внимания, когда они участвуют в разговоре, так как им приходится следить за ходом беседы и высказывать свое мнение. Причем мозг совершенно не интересует, что вы «разговариваете» с книгой! С другой стороны, если текст сух и формален, то мозг чувствует то же, что чувствуете вы на скучной лекции в роли пассивного участника. Его клонит в сон.

Но рисунки и разговорный стиль — это только начало.

Вот что сделали МЫ



Мы использовали **рисунки**, потому что мозг лучше приспособлен для восприятия графики, чем текста. С точки зрения мозга рисунок стоит 1024 слов. А когда текст комбинируется с графикой, мы внедряем текст прямо в рисунки, потому что мозг при этом работает эффективнее.

Мы используем **избыточность**: повторяем одно и то же несколько раз, применяя *разные* средства передачи информации, обращаемся к разным чувствам — и все для повышения вероятности того, что материал будет закодирован в нескольких областях вашего мозга.

Гуру Паттернов

Мы используем концепции и рисунки несколько **неожиданным** образом, потому что мозг лучше воспринимает новую информацию. Кроме того, рисунки и идеи обычно имеют ***эмоциональное*** *содержание*, потому что мозг обращает внимание на биохимию эмоций. То, что заставляет нас *чувствовать*, лучше запоминается — будь то ***шутка***, ***удивление*** или ***интерес***.

Мы используем ***разговорный стиль***, потому что мозг лучше воспринимает информацию, когда вы участвуете в разговоре, а не пассивно слушаете лекцию. Это происходит и при *чтении*.

В книгу включены многочисленные упражнения, потому что мозг лучше запоминает, когда вы работаете самостоятельно. Мы постарались сделать их непростыми, но интересными — то, что предпочитает большинство читателей.

КЛЮЧЕВЫЕ МОМЕНТЫ

Мы совместили ***несколько стилей обучения***, потому что одни читатели любят пошаговые описания, другие стремятся сначала представить «общую картину», а третьим хватает фрагмента кода. Независимо от ваших личных предпочтений полезно видеть несколько вариантов представления одного материала.

Мы постарались задействовать ***оба полушария вашего мозга***; это повышает вероятность усвоения материала. Пока одна сторона мозга работает, другая часто имеет возможность отдохнуть; это повышает эффективность обучения в течение продолжительного времени.

Головоломки

А еще в книгу включены ***истории*** и упражнения, отражающие другие точки зрения. Мозг качественнее усваивает информацию, когда ему приходится оценивать и выносить суждения.

В книге часто встречаются ***вопросы***, на которые не всегда можно дать простой ответ, потому что мозг быстрее учится и запоминает, когда ему приходится что-то делать. Невозможно накачать *мышцы*, наблюдая за тем, как занимаются *другие*. Однако мы позаботились о том, чтобы усилия читателей были приложены в *верном* направлении. Вам не придется ломать голову над невразумительными примерами или разбираться в сложном, перенасыщенном техническим жаргоном или слишком лаконичном тексте.

Мы руководствовались принципом ***80/20***. Для серьезного изучения одной книги все равно недостаточно, поэтому мы не пытались рассказать обо всем. Речь пойдет только о самом необходимом.

Вырежьте и прикрепите на холодильник.

Что можете сделать ВЫ, чтобы заставить свой мозг повиноваться

Мы свое дело сделали. Остальное за вами. Эти советы станут отправной точкой; прислушайтесь к своему мозгу и определите, что вам подходит, а что не подходит. Пробуйте новое.

1 Не торопитесь. Чем больше вы поймете, тем меньше придется запоминать.

Просто читать недостаточно. Когда книга задает вам вопрос, не переходите к ответу. Представьте, что кто-то *действительно* задает вам вопрос. Чем глубже ваш мозг будет мыслить, тем скорее вы поймете и запомните материал.

2 Выполняйте упражнения, делайте заметки.

Мы включили упражнения в книгу, но выполнять их за вас не собираемся. И не *разглядывайте* упражнения. **Берите карандаш и пишите.** Физические действия *во время* учения повышают его эффективность.

3 Читайте врезки.

Это значит: читайте всё. ***Врезки – часть основного материала!*** Не пропускайте их.

4 Не читайте другие книги после этой перед сном.

Часть обучения (особенно перенос информации в долгосрочную память) происходит *после* того, как вы откладываете книгу. Ваш мозг не сразу усваивает информацию. Если во время обработки поступит новая информация, часть того, что вы узнали ранее, может быть потеряна.

5 Пейте воду. И побольше.

Мозг лучше всего работает в условиях высокой влажности. Дегидратация (которая может наступить еще до того, как вы почувствуете жажду) снижает когнитивные функции.

6 Говорите вслух.

Речь активизирует другие участки мозга. Если вы пытаетесь что-то понять или получше запомнить, произнесите вслух. А еще лучше – попробуйте объяснить кому-нибудь другому. Вы будете быстрее усваивать материал и, возможно, откроете для себя что-то новое.

7 Прислушивайтесь к своему мозгу.

Следите за тем, когда ваш мозг начинает уставать. Если вы начинаете поверхностно воспринимать материал или забываете только что прочитанное – пора сделать перерыв.

8 *Чувствуйте*!

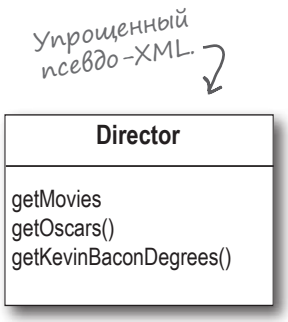
Ваш мозг должен знать, что материал книги действительно *важен*. Переживайте за героев наших историй. Придумывайте собственные подписи к фотографиям. Поморщиться над неудачной шуткой все равно лучше, чем не почувствовать ничего.

9 *Проектируйте*!

Примените новые знания к проекту, над которым вы работаете, или переделайте старый проект. Просто сделайте *хоть что-нибудь*, чтобы приобрести практический опыт за рамками упражнений. Все, что для этого нужно – это карандаш и подходящая задача.

Примите к сведению

Это учебник, а не справочник. Мы намеренно убрали из книги все, что могло бы помешать изучению материала, над которым вы работаете. И при первом чтении книги начинать следует с самого начала, потому что книга предполагает наличие у читателя определенных знаний и опыта.



Мы используем простые UML-подобные диаграммы.

Скорее всего, вы уже сталкивались с UML, но знание UML не является строго обязательным для восприятия материала. Если вы никогда не имели дела с UML — не огорчайтесь, в книге попутно приводятся краткие пояснения. Иначе говоря, вам не придется одновременно думать о паттернах проектирования *и* о UML. Наши диаграммы правильнее назвать «UML-подобными»; мы пытаемся соблюдать правила UML, но время от времени слегка нарушаем их (обычно из эстетических соображений).

Мы не описываем все когда-либо созданные паттерны.

Существует великое множество паттернов проектирования: исходные базовые паттерны (называемые паттернами «Банды Четырех»), паттерны J2EE, паттерны JSP, архитектурные паттерны, паттерны игрового программирования и т. д. Мы постарались глубоко и содержательно представить важнейшие паттерны из числа базовых. Другие паттерны (значительно реже применяемые на практике) описаны в приложении. Впрочем, после прочтения книги вы сможете взять любой справочник паттернов и легко освоить незнакомые темы.

Упражнения ОБЯЗАТЕЛЬНЫ.

Упражнения являются частью основного материала книги. Одни упражнения способствуют запоминанию материала, другие помогают лучше понять его, третьи ориентированы на его практическое применение. **Не пропускайте упражнения.**

Повторение применяется намеренно.

У книг этой серии есть одна принципиальная особенность: мы хотим, чтобы вы *действительно хорошо* усвоили материал. И чтобы вы запомнили все, что узнали. Большинство справочников не ставит своей целью успешное запоминание, но это не справочник, а учебник, поэтому некоторые концепции излагаются в книге по несколько раз.

Примеры кода были сделаны по возможности компактными.

Наши читатели не любят просматривать по 200 строк кода, чтобы найти две нужные строки. Большинство примеров книги приводится в минимальном контексте, чтобы та часть, которую вы непосредственно изучаете, была понятной и простой. Не ждите, что весь код будет стопроцентно устойчивым или даже просто завершенным — примеры написаны в учебных целях, и не всегда являются полнофункциональными.

Иногда в примеры не включаются все необходимые директивы импорта; любой Java-программист должен знать, что ArrayList находится в пакете java.util. Если директивы импорта не являются частью базового JSE API, мы упоминаем об этом в тексте. Кроме того, весь исходный код примеров размещен на веб-сайте. Вы можете загрузить его по адресу `http://wickedlysmart.com/head-first-design-patterns/`

Чтобы сосредоточиться на учебной стороне кода, мы не стали размещать наши классы в пакетах (иначе говоря, все они находятся в Java-пакете по умолчанию). В реальных приложениях так поступать не рекомендуется. Загрузив примеры кода с сайта, вы увидите, что в них все классы *размещены* в пакетах.

Упражнения «Мозговой штурм» не имеют ответов.

В некоторых из них правильного ответа вообще нет, в других вы должны сами решить, насколько правильны ваши ответы (это является частью процесса обучения). В некоторых упражнениях «Мозговой штурм» приводятся подсказки, которые помогут вам найти нужное направление.

Технические рецензенты

Джеф Кампс

Валентин Кретмас

Барни Мариспини

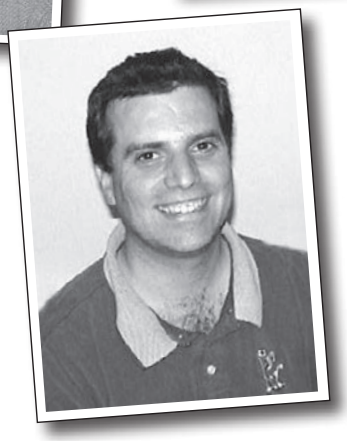
Айк Ван Атта

Бесстрашный предводитель группы HFDP Extreme Review Team

Йоханнес де Йонг

Джейсон Менар

Марк Спритцлер



Дирк Шрекман

Филипп Маке

Посвящается Филиппу Маке

1960–2004

Твои выдающиеся технические знания, энтузиазм и забота об учениках всегда будут вдохновлять нас.

Мы никогда не забудем тебя.

Благодарности

Сотрудникам издательства O'Reilly

Прежде всего мы благодарны **Майку Лукидесу** из O'Reilly за то, что он предложил исходную идею и помог преобразовать концепцию книг «Head First» в серию. Большое спасибо **Тиму О'Рейли**, вдохновителю серии «Head First». Спасибо «покровительнице» серии **Кайл Харт**, звезде рок-н-ролла **Элли Фолькхаузен** за творческий дизайн обложки и **Колин Горман** за бескомпромиссное редактирование. Также мы благодарим **Майка Хендриксона** за предложение написать книгу о паттернах проектирования и за подбор команды.

Нашим бесстрашным рецензентам

Мы чрезвычайно благодарны руководителю группы технического рецензирования **Йоханнесу де Йонгу**. Йоханнес, ты — наш герой. Также мы глубоко ценим вклад соуправляющего группы рецензирования Javaranch, покойного **Филиппа Маке**. Своими усилиями он облегчил жизнь тысячам разработчиков.

Джеф Кампс превосходно справляется с поиском недостатков в черновых вариантах глав. И на этот раз его замечания снова заметно повлияли на книгу. Спасибо, Джеф! **Валентин Кретас** (специалист по АОП), работавший с нами с самой первой книги из серии «Head First», доказал, насколько нам необходим его технический опыт и понимание сути вещей. Два новых участника группы рецензирования — Барни Мариспини и Айк Ван Атта — отлично справились со своей работой.

Кроме того, мы получили замечательную техническую поддержку от модераторов/гуру Javaranch **Марка Спритцлера**, **Джейсона Менара**, **Дирка Шрекмана**, **Томаса Пола** и **Маргариты Исаевой**. И как всегда, спасибо «начальнику» javanranch.com **Полу Уитону**.

Спасибо всем участникам конкурса на оформление обложки книги. Победитель конкурса Си Брюстер написал очерк, который убедил нас выбрать женский портрет, изображенный на обложке.

В подготовке обновленной версии книги за 2014 год нам помогали научные редакторы: Джордж Хоффер (George Hoffer), Тед Хилл (Ted Hill), Тодд Бартошкевич (Todd Bartoszkiewicz), Сильвен Тенье (Sylvain Tenier), Скотт Дэвидсон (Scott Davidson), Кевин Райан (Kevin Ryan), Рич Уорд (Rich Ward), Марк Фрэнсис Ягер (Mark Francis Jaeger), Марк Масс (Mark Masse), Гленн Рэй (Glenn Ray), Баярд Фетлер (Bayard Fetler), Пол Хиггинс (Paul Higgins), Мэтт Карпентер (Matt Carpenter), Джулия Уильямс (Julia Williams), Мэтт Маккалох (Matt McCullough) и Мэри Энн Белармино (Mary Ann Belarmino).

...И другие*

От Эрика и Элизабет

Работа над книгой из серии «Head First» напоминала поездку в незнакомую местность в компании двух великолепных проводников: **Кэтти Сьерра** и **Берта Бейтса**. С ними мы отбросили все традиционные правила написания книг и вошли в замечательный мир теории обучения, научных принципов мышления и поп-культуры, в центре которого всегда находится читатель. Спасибо вам за то, что помогли нам выйти на правильный путь, заставляли нас двигаться вперед, и прежде всего — за то, что доверили нам свое создание. Безусловно, вы оба умны до неприличия, а еще вы самые классные 29-летние ребята, которых мы знаем. Что там дальше в ваших творческих планах?

Майк Лукидес, Майк Хендриксон и **Мэган Бланшетт**, огромное вам спасибо. Майк Л. сопровождал нас на каждом шагу этого пути. Его проницательные комментарии способствовали формированию концепции книги, а поддержка помогала нам двигаться вперед. Майк Х. целых пять лет уговаривал нас написать книгу о паттернах; мы рады тому, что книга была опубликована именно в серии «Head First». И Мэг, спасибо тебе за то, что взялась за это обновление вместе с нами; без тебя мы бы не справились.

Хотим особо поблагодарить **Эрика Гамму** — его труды по рецензированию книги вышли далеко за рамки служебных обязанностей (он даже взял черновик в отпуск). Эрик, твой интерес к книге вдохновлял нас, а доскональный технический анализ принес неизмеримую пользу. Спасибо всей «**Банде Четырех**» за поддержку и интерес, а также за их появление в Объективиле. Мы также многим обязаны **Уорду Каннингему** и участникам сообщества паттернов, создателям Портлендского хранилища паттернов — бесценного источника информации для написания книги.

В завершение хотим особо поблагодарить всю **группу рецензирования Javaranch** за прекрасную работу и поддержку. Эта книга обязана вам бóльшим, чем вы думаете.

От Кэтти и Берта

Теперь мы (к своему ужасу) знаем, что мы *не единственные*, кто может написать книгу из серии «Head First» ;) Но если читатели *вдруг* решат, что самым лучшим эта книга обязана Кэтти и Берту... кто *мы* такие, чтобы их переубеждать?

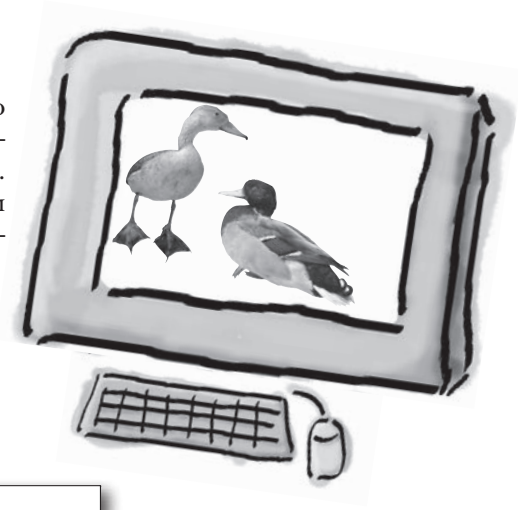
*Большое количество благодарностей объясняется тем, что мы проверяем свою теорию. Согласно этой теории, каждый, о ком упоминается в разделе благодарностей, купит хотя бы один экземпляр книги — а может, и больше (для родственников и т. д.). Если вы хотите, чтобы мы поблагодарили вас в *следующей* книге, и у вас большая семья — напишите нам.

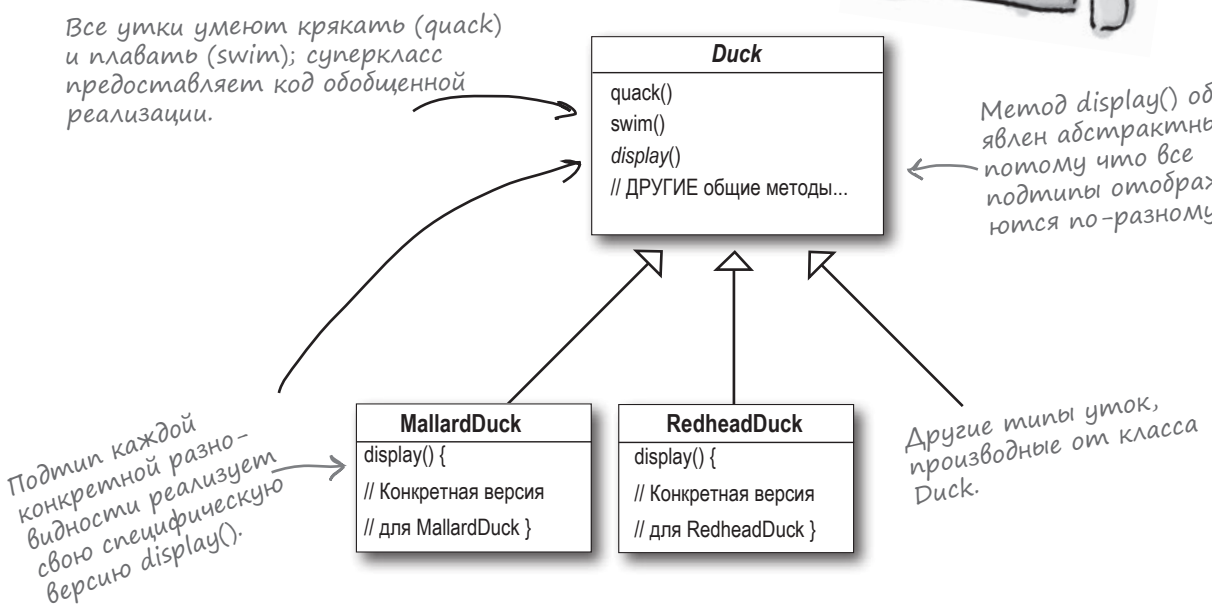
Добро пожаловать в мир паттернов

Наверняка вашу задачу кто-то уже решал. В этой главе вы узнаете, почему (и как) следует использовать опыт других разработчиков, которые уже сталкивались с аналогичной задачей и успешно решили ее. Заодно мы поговорим об использовании и преимуществах паттернов проектирования, познакомимся с ключевыми принципами ООП и разберем пример одного из паттернов. Лучший способ использовать паттерны — *запомнить их*, а затем научиться *распознавать* те места ваших архитектур и существующих приложений, где их уместно *применить*. Таким образом, вместо программного кода вы повторно используете чужой *опыт*.

Все началось с простого приложения SimUDuck

Джо работает на компанию, выпустившую чрезвычайно успешный имитатор утиного пруда. В этой игре представлен пруд, в котором плавают и крякают утки разных видов. Проектировщики системы воспользовались стандартным приемом ООП и определили суперкласс Duck, на основе которого объявляются типы конкретных видов уток.





За последний год компания испытывала жесткое давление со стороны конкурентов. Через неделю долгих выездных совещаний за игрой в гольф руководство компании решило, что пришло время серьезных изменений. Нужно сделать что-то *действительно* впечатляющее, что можно было бы продемонстрировать на предстоящем собрании акционеров *на следующей неделе*.

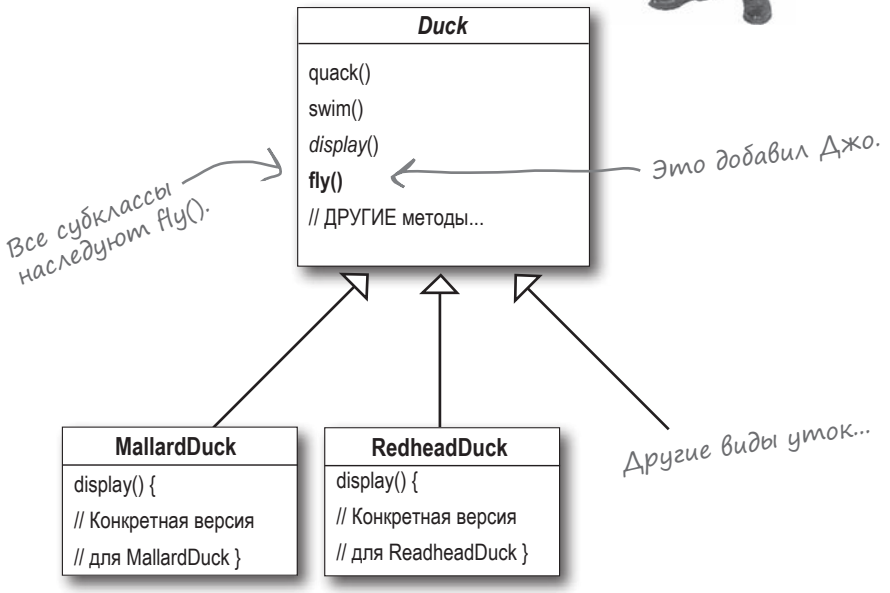
Теперь утки будут ЛЕТАТЬ

Начальство решило, что летающие утки — именно та «изюминка», которая сокрушит всех конкурентов. И конечно, начальство Джо пообещало, что Джо легко соорудит что-нибудь этакое в течение недели. «В конце концов, он ООП-программист... *Какие могут быть трудности?»*

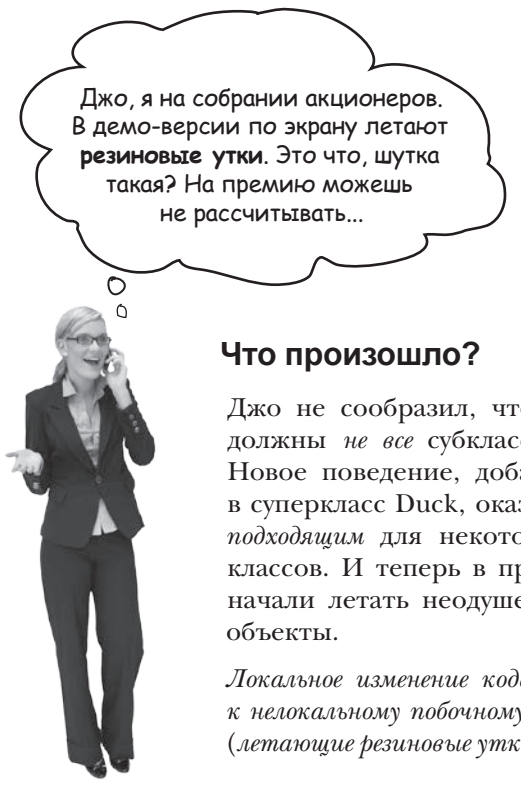
Чего мы добиваемся.

Джо

Я добавлю метод fly() в класс Duck, и он будет унаследован всеми производными классами. Пора продемонстрировать мои таланты в области ООП.



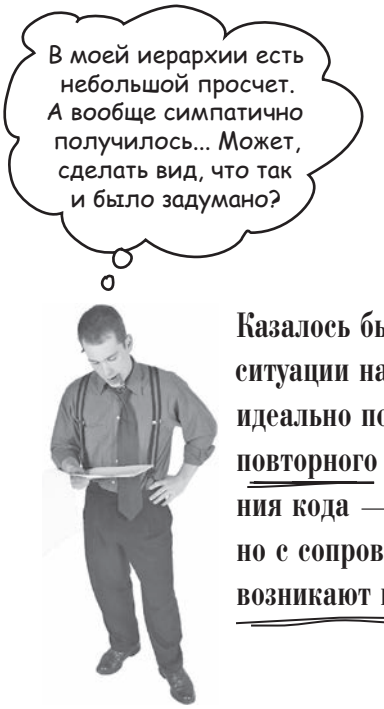
Но тут все пошло наперекосяк ...



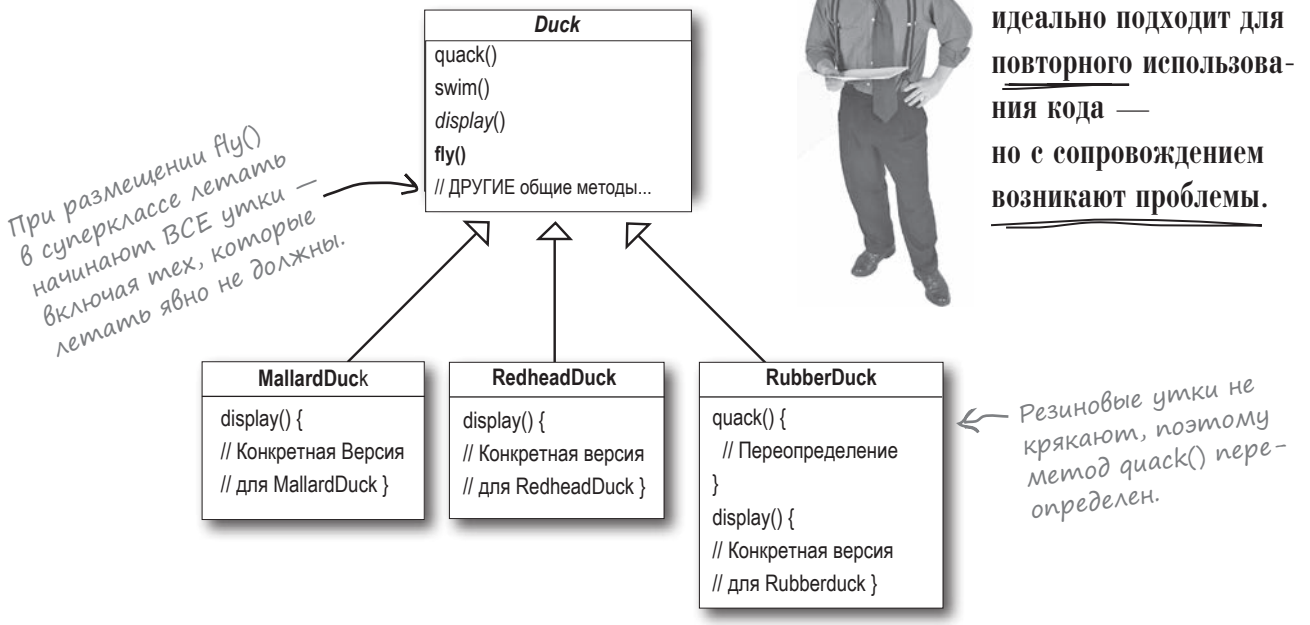
Что произошло?

Джо не сообразил, что *летать* должны *не все* субклассы Duck. Новое поведение, добавленное в суперкласс Duck, оказалось *неподходящим* для некоторых субклассов. И теперь в программе начали летать неодушевленные объекты.

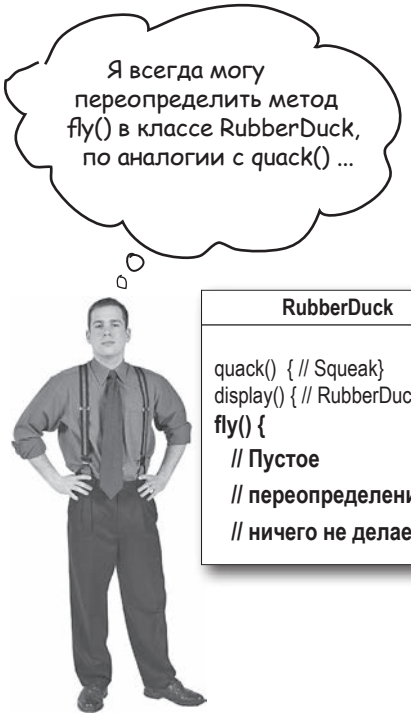
Локальное изменение кода привело к нелокальному побочному эффекту (летающие резиновые утки!)



Казалось бы, в этой ситуации наследование идеально подходит для повторного использования кода — но с сопровождением возникают проблемы.



Джо думает о наследовании...



RubberDuck

```
quack()  { // Squeak}
display() { // RubberDuck }
fly() {
  // Пустое
  // переопределение
  // ничего не делает }
```

Еще один класс в иерархии; деревянные утки не летают и не крякают.

DecoyDuck

```
quack() {
  // Пустое переопределение
}

display() { // DecoyDuck }

fly() {
  //Пустое переопределение}
}
```

Возьми в руку карандаш

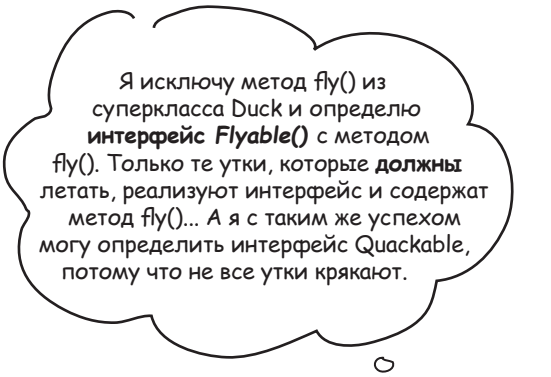
Какие из перечисленных недостатков относятся к применению *наследования* для реализации Duck? (Укажите все варианты.)

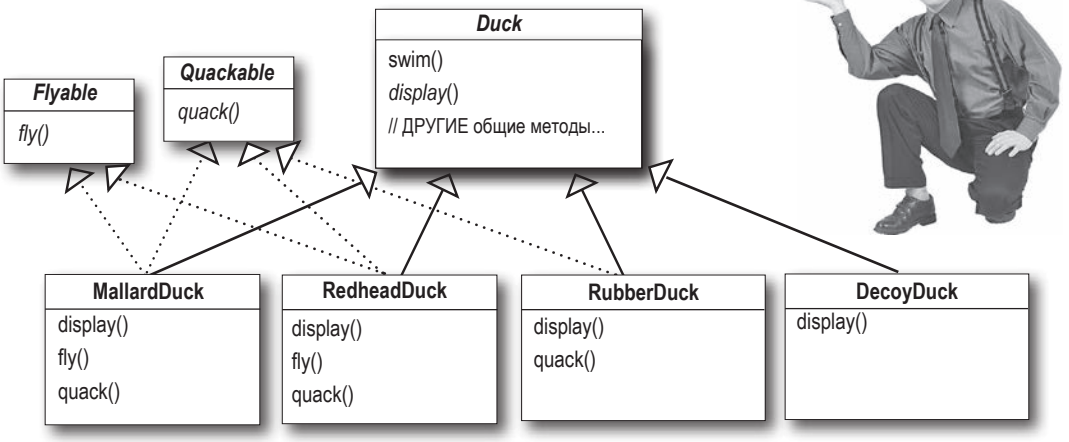
- ❑ A. Дублирование кода в субклассах.
- ❑ B. Трудности с изменением поведения на стадии выполнения.
- ❑ C. Уток нельзя научить танцевать.
- ❑ D. Трудности с получением информации обо всех аспектах поведения уток.
- ❑ E. Утки не могут летать и крякать одновременно.
- ❑ F. Изменения могут оказать непредвиденное влияние на другие классы.

Как насчет интерфейса?

Джо понял, что наследование не решит проблему – он только что получил служебную записку, в которой говорится, что продукт должен обновляться каждые 6 месяцев (причем начальство еще не знает, как именно). Джо знает, что спецификация будет изменяться, а ему придется искать (и, возможно, переопределять) методы fly() и quack() для каждого нового субкласса, включаемого в программу... *вечно*.

Итак, ему нужен более простой способ заставить летать или крякать только *некоторых* (но не всех!) уток.





А что ВЫ думаете об этой архитектуре?

По-моему, это самая дурацкая из твоих идей. **Как насчет дублирования кода?** Тебе не хочется переопределять несколько методов, но как тебе понравится вносить маленькое изменение в поведении fly()... во всех 48 «летающих» субклассах Duck?!

А как бы ВЫ поступили на месте Джо?

Мы знаем, что *не все* субклассы должны реализовывать методы fly() или quack(), так что наследование не является оптимальным решением. С другой стороны, реализация интерфейсов Flyable и (или) Quackable решает проблему *частично* (резиновые утки перестают летать), но полностью исключает возможность повторного использования кода этих аспектов поведения — а следовательно, создает *другой* кошмар из области сопровождения. Не говоря уже о том, что даже *летающие* утки могут летать по-разному...

Вероятно, вы ждете, что сейчас паттерн проектирования явится на белом коне и всех спасет. Но какой интерес в готовом рецепте? Нет, мы самостоятельно вычислим решение, *руководствуясь канонами ОО-проектирования.*

А как было бы хорошо, если бы программу можно было написать так, чтобы вносимые изменения оказывали минимальное влияние на существующий код... Мы тратили бы меньше времени на **переработку** и больше — на всякие интересные вещи...

Единственная константа в программировании

На что всегда можно рассчитывать в ходе работы над проектом?

В какой бы среде, над каким бы проектом, на каком угодно языке вы ни работали — что всегда будет неизменно присутствовать в вашей программе?

RNHƎHƎMƐN

(ответ можно прочитать в зеркале)

Как бы вы ни спроектировали свое приложение, со временем оно должно развиваться и изменяться — иначе оно *умрет*.

Возьми в руку карандаш

Изменения могут быть обусловлены многими факторами. Укажите некоторые причины для изменения кода в приложениях (чтобы вам было проще, мы привели пару примеров).

Клиенты или пользователи требуют реализации новой или расширенной функциональности.

Компания переходит на другую СУБД, а данные будут приобретаться у другого поставщика в новом формате. Ужас!

Захожу на цель...

Итак, наследование нам не подошло, потому что утиное поведение изменяется в субклассах, а некоторые аспекты поведения присутствуют *не во всех* субклассах. Идея с интерфейсами Flyable и Quackable на первый взгляд выглядит заманчиво — но интерфейсы Java не имеют реализации, что исключает повторное использование кода. И если когда-нибудь потребуется изменить аспект поведения, вам придется искать и изменять его во всех субклассах, где он определяется, — скорее всего, с внесением *новых* ошибок!

К счастью, для подобных ситуаций существует полезный принцип проектирования.

Принцип проектирования

Выделите аспекты приложения, которые могут изменяться, и отделите их от тех, которые всегда остаются постоянными.

Первый из многих принципов проектирования, которые встречаются в этой книге.

Выделите то, что изменяется, и «инкапсулируйте» эти аспекты, чтобы они не влияли на работу остального кода.

Результат? Меньше непредвиденных последствий от изменения кода, бóльшая гибкость ваших систем!

Иначе говоря, если некий аспект кода изменяется (допустим, с введением новых требований), то его необходимо отделить от тех аспектов, которые остаются неизменным.

Другая формулировка того же принципа: ***выделите переменные составляющие и инкапсулируйте их, чтобы позднее их можно было изменять или расширять без воздействия на постоянные составляющие.***

При всей своей простоте эта концепция лежит в основе почти всех паттернов проектирования. *Все паттерны обеспечивают возможность изменения некоторой части системы независимо от других частей.*

Итак, пришло время вывести утиное поведение за пределы классов Duck!

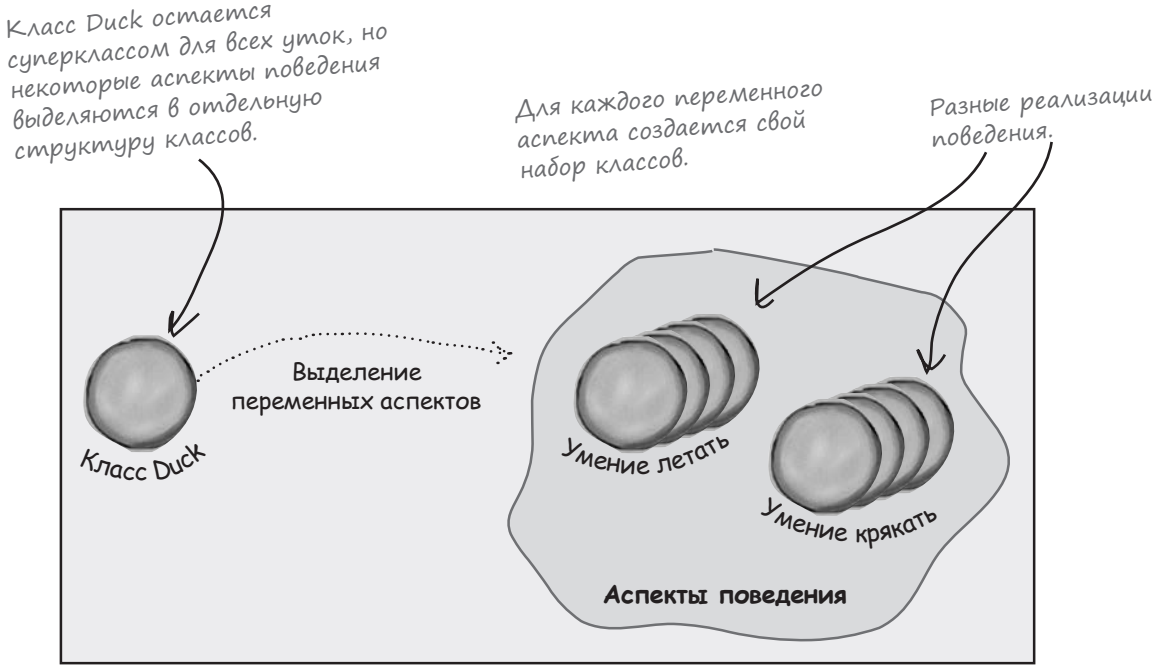
Отделяем переменное от постоянного

С чего начать? Если не считать проблем с fly() и quack(), класс Duck работает хорошо, и другие его аспекты вряд ли будут часто изменяться. Таким образом, если не считать нескольких второстепенных модификаций, класс Duck в целом остается неизменным.

Чтобы отделить «переменное от постоянного», мы создадим два *набора* классов (совершенно независимых от Duck): один для *fly*, другой для *quack*. Каждый набор классов содержит реализацию соответствующего поведения.

Мы знаем, что fly() и quack() — части класса Duck, изменяющиеся в зависимости от субкласса.

Чтобы отделить эти аспекты поведения от класса Duck, мы выносим оба метода *за пределы* класса Duck и создаем новый набор классов для представления каждого аспекта.



Проектирование переменного поведения

Как же спроектировать набор классов, реализующих переменные аспекты поведения ?

Нам хотелось бы сохранить максимальную гибкость; в конце концов, все неприятности возникли именно из-за отсутствия гибкости в поведении Duck. Например, желательно иметь возможность создать новый экземпляр MallardDuck и инициализировать его с конкретным *типом* поведения fly(). И раз уж на то пошло, почему бы не предусмотреть возможность динамического изменения поведения? Иначе говоря, в классы Duck следует включить методы выбора поведения, чтобы способ полета MallardDuck можно было *изменить во время выполнения*.

Так мы переходим ко второму принципу проектирования.

Принцип проектирования

Программируйте на уровне интерфейса, а не на уровне реализации.

Для представления каждого аспекта поведения (например, FlyBehavior или QuackBehavior) будет использоваться интерфейс, а каждая реализация аспекта поведения будет представлена реализацией этого интерфейса.

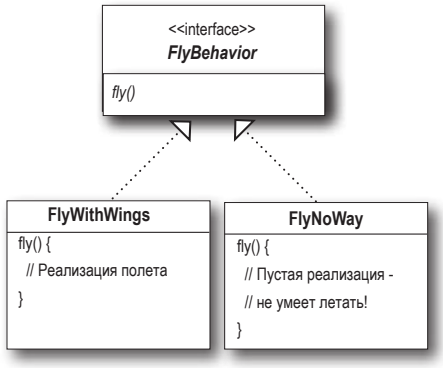
Итак, на этот раз интерфейсы реализуются не классами *Duck*. Вместо этого мы создаем набор классов, единственным смыслом которых является представление некоторого поведения. И теперь интерфейс поведения реализуется *классом поведения*, а не классом Duck.

Такой подход отличается от того, что делалось прежде, когда поведение предоставлялось либо конкретной реализацией в суперклассе Duck, либо специализированной реализацией в самом субклассе. В обоих случаях возникала зависимость от *реализации*. Мы были вынуждены использовать именно эту реализацию, и изменить поведение было невозможно (без написания дополнительного кода).

В новом варианте архитектуры субклассы Duck используют поведение, представленное *интерфейсом* (FlyBehavior или QuackBehavior), поэтому фактическая *реализация* этого поведения (то есть конкретное поведение, запрограммированное в классе, реализующем FlyBehavior или QuackBehavior) не привязывается к субклассу Duck.

Отныне аспекты поведения Duck будут находиться в отдельных классах, реализующих интерфейс конкретного аспекта.

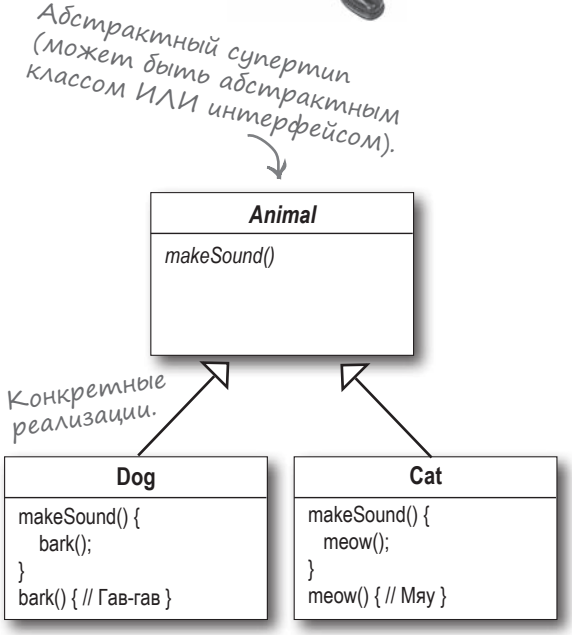
В этом случае классам Duck не нужно знать подробности реализации своих аспектов поведения.



«Интерфейс» в данном случае означает «супертип».

Слово *интерфейс* имеет несколько смыслов. Наряду с *концепцией* интерфейса существует конструкция Java **`interface`**. Программирование на уровне интерфейса может не использовать Java-конструкцию **`interface`**. Собственно, главной целью применения полиморфизма посредством программирования на уровне супертипа является как раз отсутствие жесткой привязки к конкретному объекту во время выполнения. Или другими словами, «переменные должны объявляться с супертипом (обычно абстрактным классом или интерфейсом), чтобы присваиваемые им объекты могли относиться к любой конкретной реализации супертипа».

Вероятно, вам все это хорошо известно, но просто для того, чтобы убедиться в общности наших представлений, рассмотрим просто пример использования полиморфного типа. Допустим, имеется абстрактный класс Animal с двумя конкретными реализациями: Dog и Cat.



Программирование на уровне реализации выглядит так:

```
Dog d = new Dog();
d.bark();
```

Объявление «d» с типом Dog требует программирования на уровне конкретной реализации Animal.

Программирование на уровне интерфейса/супертипа:

```
Animal animal = new Dog();
animal.makeSound();
```

Полиморфное использование ссылки.

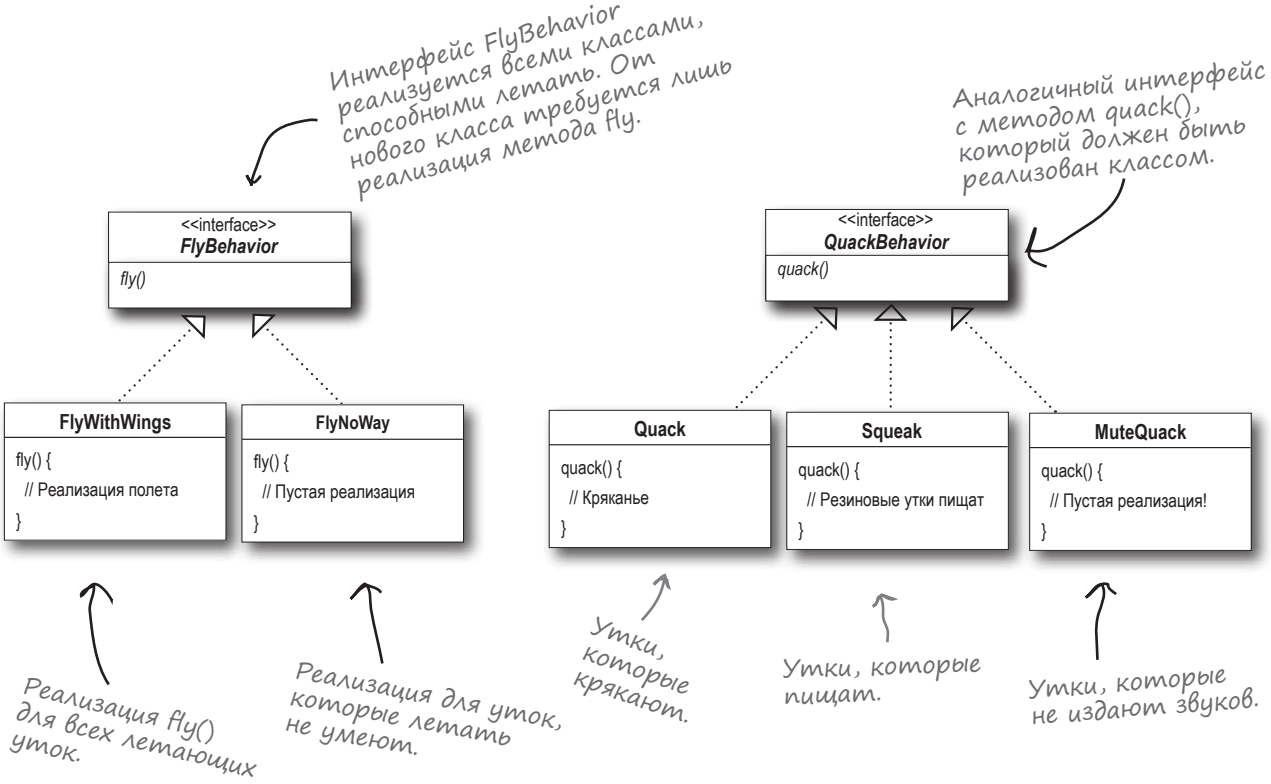
Или еще лучше, вместо жесткой фиксации подтипа в коде (new Dog()), **объект конкретной реализации присваивается во время выполнения**:

```
a = getAnimal();
a.makeSound();
```

Фактический подтип Animal неизвестен... Важно лишь то, что он умеет реагировать на makeSound().

Реализация поведения уток

Интерфейсы FlyBehavior и QuackBehavior вместе с соответствующими классами, реализующими каждое конкретное поведение.



Такая архитектура позволяет использовать поведение fly() и quack() в других типах объектов, потому что это поведение не скрывается в классах Duck!

Кроме того, мы можем добавлять новые аспекты поведения без изменения существующих классов поведения, и без последствий для классов Duck, *использующих* существующее поведение.

Все преимущества ПОВТОРНОГО ИСПОЛЬЗОВАНИЯ без недостатков, присущих наследованию!

Часто Задаваемые Вопросы

В: Так я всегда должен сначала реализовать приложение, посмотреть, что в нем изменяется, а затем вернуться, выделить и инкапсулировать переменные составляющие?

О: Не всегда; в ходе проектирования приложения часто удается заранее выявить изменяющиеся аспекты и включить гибкие средства для работы с ними в программный код. Общие принципы и паттерны применимы на любой стадии жизненного цикла разработки.

В: Может, Duck тоже стоит преобразовать в интерфейс?

О: Не в этом случае. Структура, в которой Duck не является интерфейсом, имеет свои преимущества: она позволяет конкретным подклассам уток (например, MallardDuck) наследовать общие свойства и методы. После исключения переменных аспектов из иерархии Duck мы пользуемся преимуществами этой структуры без всяких проблем.

В: Класс, представляющий поведение, выглядит немного странно. Разве классы не должны представлять *сущности*? И разве классы не должны обладать *состоянием* И *поведением*?

О: Действительно, в ОО-системах классы представляют сущности, которые обычно обладают как состоянием (переменными экземпляров), так и методами. И в данном случае сущностью оказывается поведение. Однако даже поведение может обладать состоянием и методами; скажем, поведение полета может использовать переменные экземпляров, представляющие атрибуты полета (количество взмахов крыльев в минуту, максимальная высота и скорость и т. д.).

Возьми в руку карандаш

1. Как бы вы поступили в новой архитектуре, если бы вам потребовалось включить в приложение SimUDuck полеты на реактивной тяге?

2. Какой класс мог бы повторно использовать поведение quack(), не являясь при этом уткой?

Ответы:

1. Создайте класс FlyRocketPowered, реализующий интерфейс FlyBehavior.

2. Например, утиный манок (охотничье устройство, подражающее кряканью).

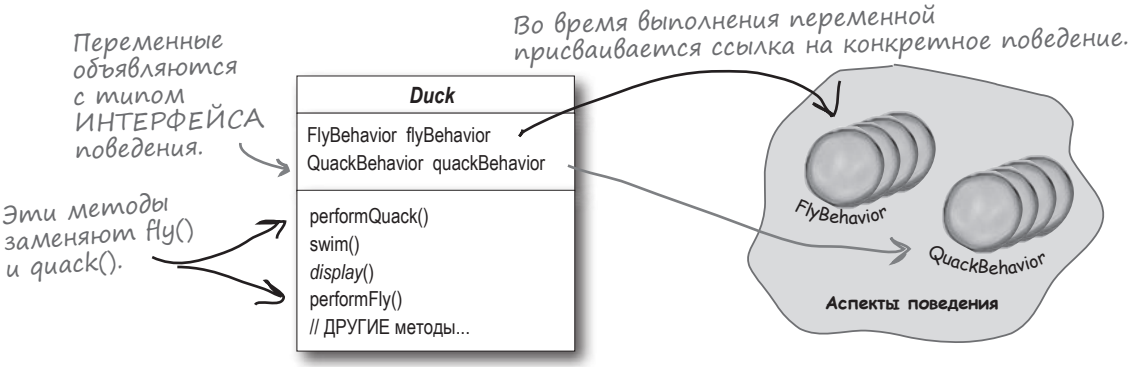
Интеграция поведения с классом Duck

У новой структуры есть одна принципиальная особенность: класс Duck теперь *делегирует* свои аспекты поведения (вместо простого использования методов, определенных в классе Duck или его субклассах). Вот как это делается:

1. **Сначала в класс Duck включаются две переменные экземпляров** *flyBehavior* и *quackBehavior*, объявленные с типом интерфейса (а не с типом конкретного класса реализации). Каждый объект на стадии выполнения присваивает этим переменным полиморфные значения, соответствующие *конкретному* типу поведения (FlyWithWings, Squeak и т. д.).

   Методы fly() и quack() удаляются из класса Duck (и всех субклассов), потому что это поведение перемещается в классы FlyBehavior и QuackBehavior.

   В классе Duck методы fly() и quack() заменяются двумя аналогичными методами: performFly() и performQuack(); вскоре вы увидите, как они работают.



2. **Реализация performQuack()**

```
public class Duck {
   QuackBehavior quackBehavior;
   // ...

   public void performQuack() {
     quackBehavior.quack();
   }
}
```

Каждый объект Duck содержит ссылку на реализацию интерфейса QuackBehavior.

Объект Duck делегирует поведение объекту, на который ссылается quackBehavior.

Все просто, верно? Вместо того, чтобы выполнять действие самостоятельно, объект Duck просто поручает эту работу объекту, на который ссылается quackBehavior. В этой части, когда нас совершенно не интересует, что это за объект, — ***важно лишь, чтобы он умел выполнять quack()***!

Подробнее об интеграции...

❸ Пора разобраться с тем, как присваиваются значения переменных **flyBehavior** и **quackBehavior**. Рассмотрим фрагмент класса MallardDuck:

```
public class MallardDuck extends Duck {

   public MallardDuck() {
      quackBehavior = new Quack();
      flyBehavior = new FlyWithWings();
   }

   public void display() {
      System.out.println("I'm a real Mallard duck");
   }
}
```

MallardDuck использует класс Quack для выполнения действия, так что при вызове performQuack() ответственность за выполнение возлагается на объект Quack.

А в качестве реализации FlyBehavior используется тип FlyWithWings.

Запомните, что MallardDuck наследует переменные quackBehavior и flyBehavior от класса Duck.

При создании экземпляра MallardDuck конструктор инициализирует унаследованную переменную экземпляра quackBehavior новым экземпляром типа Quack (класс конкретной реализации QuackBehavior).

То же самое происходит и с другим аспектом поведения: конструктор MallardDuck инициализирует переменную flyBehavior экземпляром типа FlyWithWings (класс конкретной реализации FlyBehavior).

Все верно, именно так мы и поступаем... *пока*.

Позднее в книге будут описаны другие паттерны, которые помогут решить эту проблему.

А пока стоит заметить, что, хотя аспекты поведения связываются с конкретными реализациями (мы создаем экземпляр класса поведения типа Quack или FlyWithWings, и присваиваем его ссылочной переменной), эти реализации можно *легко* менять во время выполнения.

Таким образом, гибкость инициализации переменных экземпляров оставляет желать лучшего. Но поскольку переменная экземпляра quackBehavior относится к интерфейсному типу, мы можем (благодаря волшебству полиморфизма) динамически присвоить другой класс реализации QuackBehavior во время выполнения.

Остановитесь на минуту и подумайте, как бы *вы* реализовали динамическое изменение поведения. (Пример кода будет приведен через несколько страниц.)

Тестирование кода Duck

1 Введите и откомпилируйте класс Duck (Duck.java, см. ниже) и класс MallardDuck class (MallardDuck.java, приводился две страницы назад).

```
public abstract class Duck {

   FlyBehavior flyBehavior;
   QuackBehavior quackBehavior;

   public Duck() {
   }

   public abstract void display();

   public void performFly() {
      flyBehavior.fly();
   }

   public void performQuack() {
      quackBehavior.quack();
   }

   public void swim() {
      System.out.println("All ducks float, even decoys!");
   }
}
```

Объявляем две ссылочные переменные с типами интерфейсов поведения. Переменные наследуются всеми субклассами Duck (в том же пакете).

Делегирование операции классам поведения.

2 Введите и откомпилируйте интерфейс FlyBehavior (FlyBehavior.java) и два класса реализации поведения (FlyWithWings.java и FlyNoWay.java).

```
public interface FlyBehavior {
   public void fly();
}
```

Интерфейс реализуется всеми классами.

```
public class FlyWithWings implements FlyBehavior {
   public void fly() {
       System.out.println("I'm flying!!");
   }
}
```

Реализация поведения для уток, которые УМЕЮТ летать...

```
public class FlyNoWay implements FlyBehavior {
   public void fly() {
       System.out.println("I can't fly");
   }
}
```

Реализация поведения для уток, которые НЕ ЛЕТАЮТ (например, резиновых).

Тестирование кода Duck продолжается...

3 Введите и откомпилируйте интерфейс QuackBehavior (QuackBehavior.java) и три класса реализации поведения (Quack.java, MuteQuack.java, and Sqeak.java).

```
public interface QuackBehavior {
   public void quack();
}
```

```
public class Quack implements QuackBehavior {
   public void quack() {
      System.out.println("Quack");
   }
}
```

```
public class MuteQuack implements QuackBehavior {
   public void quack() {
       System.out.println("<< Silence >>");
   }
}
```

```
public class Squeak implements QuackBehavior {
   public void quack() {
       System.out.println("Squeak");
   }
}
```

4 Введите и откомпилируйте тестовый класс (MiniDuckSimulator.java).

```
public class MiniDuckSimulator {
   public static void main(String[] args) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();
   }
}
```

Вызов метода performQuack(), унаследованного классом MallardDuck; метод делегирует выполнение операции по ссылке на QuackBehavior (то есть вызывает quack() через унаследованную переменную quackBehavior).

Затем то же самое происходит с методом performFly(), также унаследованным классом MallardDuck.

5 Выполните код!

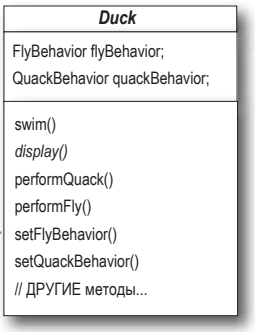
```
File Edit Window Help Yadayadayada
%java MiniDuckSimulator

Quack

I'm flying!!

%
```

Динамическое изменение поведения

Согласитесь, обидно было бы наделить наших уток возможностями динамической смены поведения — и не использовать их! Предположим, вы хотите, чтобы тип поведения задавался set-методом подкласса (вместо создания экземпляра в конструкторе).

❶ Добавьте два новых метода в класс Duck:

```
public void setFlyBehavior(FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```



Вызывая эти методы в любой момент, мы можем изменить поведение утки «на лету».

❷ Создайте новый субкласс Duck (ModelDuck.java).

```
public class ModelDuck extends Duck {
    public ModelDuck() {
        flyBehavior = new FlyNoWay();
        quackBehavior = new Quack();
    }

    public void display() {
        System.out.println("I'm a model duck");
    }
}
```

Утка-приманка изначально летать не умеет...

❸ Определите новый тип FlyBehavior (FlyRocketPowered.java).

Определяем новое поведение — реактивный полет.

```
public class FlyRocketPowered implements FlyBehavior {
    public void fly() {
        System.out.println("I'm flying with a rocket!");
    }
}
```

4 **Внесите изменения в тестовый класс(MiniDuckSimulator.java), добавьте экземпляр ModelDuck и переведите его на реактивную тягу.**

```
public class MiniDuckSimulator {
   public static void main(String[] args) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();

      Duck model = new ModelDuck();
      model.performFly();
      model.setFlyBehavior(new FlyRocketPowered());
      model.performFly();
   }
}
```

до

Первый вызов performFly() передается реализации, заданной в конструкторе ModelDuck — то есть экземпляру FlyNoWay.

Способность утки-приманки к полету переключается динамически! Если бы реализация находилась в иерархии Duck, ТАКОЕ было бы невозможно.

Вызываем set-метод, унаследованный классом ModelDuck, и... утка-приманка вдруг взлетает на реактивном двигателе!

после

5 **Поехали!**

```
File Edit Window Help Yabadabadoo
%java MiniDuckSimulator
Quack
I'm flying!!
I can't fly
I'm flying with a rocket
```

Поведение утки во время выполнения изменяется простым вызовом set-метода.

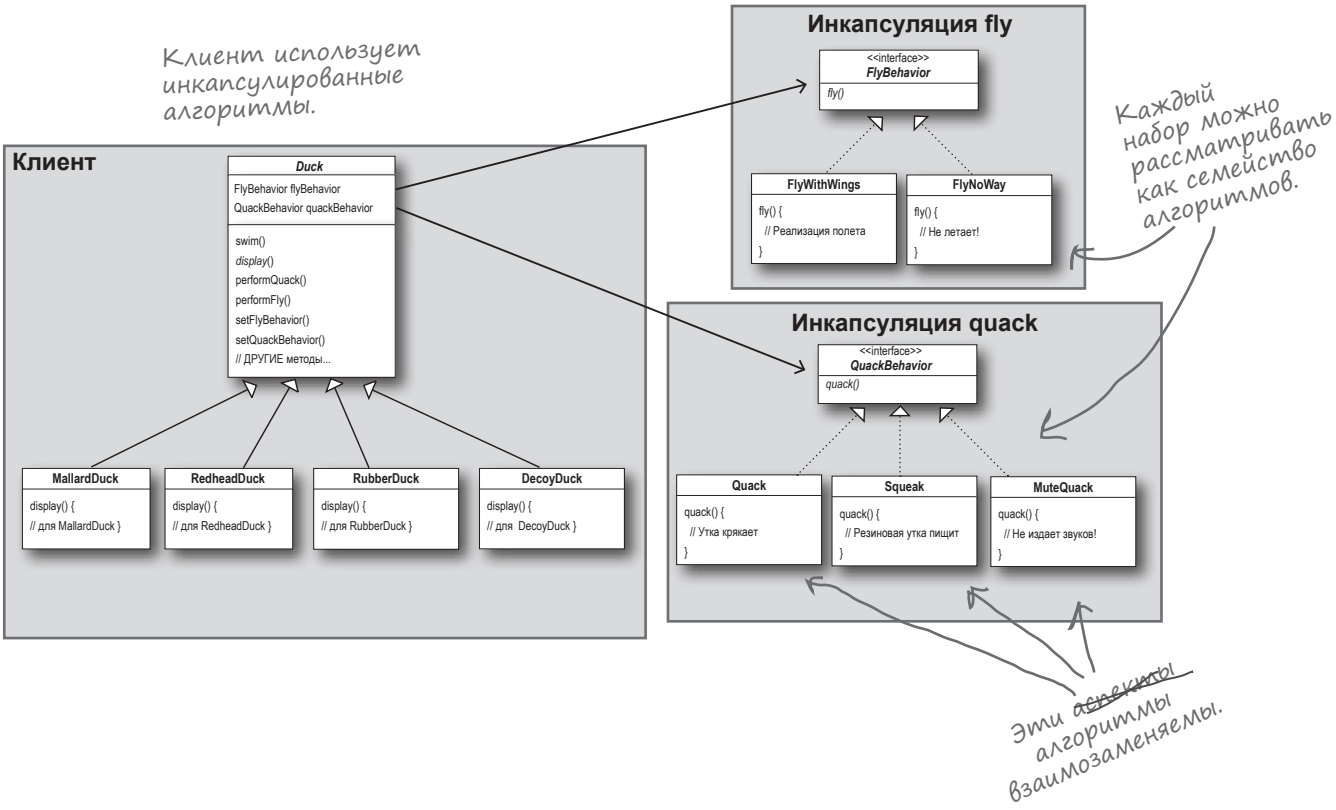
Инкапсуляция поведения: общая картина

Мы основательно повозились с конкретной архитектурой. Пора сделать шаг назад и взглянуть на картину в целом.

Ниже изображена вся переработанная структура классов. В ней есть все, чего можно ожидать: классы уток, расширяющие Duck, а также классы поведения, реализующие FlyBehavior и QuackBehavior.

Стоит заметить, что мы начинаем рассматривать происходящее с несколько иной точки зрения. Поведение утки уже рассматривается не как *совокупность аспектов поведения*, а как *семейство алгоритмов*. В архитектуре SimUDuck алгоритмы представляют то, что делают утки (как они летают, крякают и т. д.), однако эту методологию с таким же успехом можно применить к набору классов для вычисления налога с продаж в разных штатах.

Обратите особое внимание на *отношения* между классами. А еще лучше — возьмите ручку и подпишите тип отношения (ЯВЛЯЕТСЯ, СОДЕРЖИТ или РЕАЛИЗУЕТ) над каждой стрелкой на диаграмме.



Отношения СОДЕРЖИТ бывают удобнее отношений ЯВЛЯЕТСЯ

Каждая утка СОДЕРЖИТ экземпляры FlyBehavior и QuackBehavior, которым делегируются выполнение соответствующих операций.

Подобные связи между двумя классами означают, что вы используете механизм ***композиции***. Поведение не *наследуется*, а предоставляется правильно выбранным объектом.

На самом деле это очень важный момент; мы вплотную подошли к третьему принципу проектирования.

Принцип проектирования

Отдавайте предпочтение композиции перед наследованием.

Как вы убедились, системы, созданные на основе композиции, обладают значительно большей гибкостью. Они позволяют не только инкапсулировать семейства алгоритмов, но и ***изменять поведение во время выполнения*** — при условии, что объект, подключенный посредством композиции, реализует правильный интерфейс.

Композиция используется во многих паттернах проектирования. Мы еще не раз вернемся к ее достоинствам и недостаткам в этой книге.

МОЗГОВОЙ ШТУРМ

Утиный манок используется охотниками для имитации утиного кряканья. Как бы вы реализовали собственную версию утиного манка, которая *не является производной* от класса Duck?

Учитель и Ученик...

Учитель: *Расскажи, что ты узнал о сущности Объектно-Ориентированного подхода.*

Ученик: *Учитель, я узнал, что он открывает путь к повторному использованию.*

Учитель: *Продолжай...*

Ученик: *Наследование позволяет повторно использовать многие полезные вещи, а время разработки исчезает так же стремительно, как срубленные стебли бамбука.*

Учитель: *Когда мы тратим больше времени:* ***до*** *или* ***после*** *завершения разработки?*

Ученик: ***После****, о Учитель. На сопровождение и доработку программ всегда уходит больше времени, чем на начальную разработку.*

Учитель: *Не значит ли это, что повторному использованию следует уделять* **больше** *внимания, чем удобству сопровождения и расширения?*

Ученик: *Полагаю, это так.*

Учитель: *Вижу, тебе еще далеко до Просветления. Иди и продолжай медитировать на наследовании. У наследования есть свои недостатки, и повторное использование может быть достигнуто другими средствами.*

Кстати, о паттернах...

Поздравляем с первым паттерном!

Вы только что применили свой первый паттерн проектирования СТРАТЕГИЯ. Да, вы не ошиблись: при переработке приложения SimUDuck был использован паттерн Стратегия. Благодаря ему проект готов к любым изменениям, возникающим в буйной фантазии вашего начальства.

А теперь, когда мы прошли довольно долгий путь к конечной цели, приведем формальное определение:

Паттерн Стратегия определяет семейство алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость. Он позволяет модифицировать алгоритмы независимо от их использования на стороне клиента.

Используйте ЭТО определение, когда вам понадобится произвести впечатление на друзей или начальство.

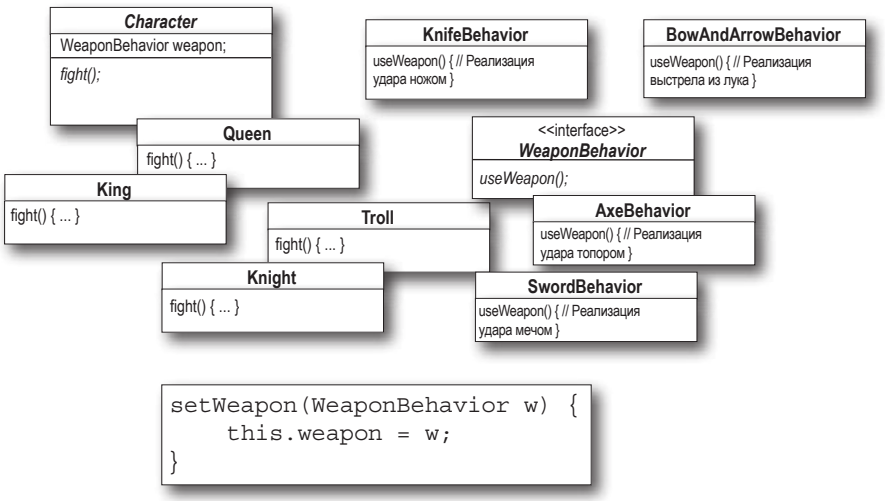
Головоломка

Ниже изображены перепутанные классы и интерфейсы, используемые для программирования приключенческой игры. В иерархию входят классы игровых персонажей и разных типов вооружения. Каждый персонаж в любой момент времени использует только один вид оружия, но может свободно менять оружие в ходе игры. Восстановите отсутствующие связи.

(Ответы приведены в конце главы.)

Ваша задача:

1. Организуйте классы в иерархию.
2. Найдите один абстрактный класс, один интерфейс и восемь классов.
3. Соедините классы стрелками.
   1. Отношение наследования («расширяет»): —▷
   2. Отношение реализации интерфейса: ·····▷
   3. Отношение типа «СОДЕРЖИТ»: ⟶
4. Включите метод setWeapon() в правильный класс.




```
setWeapon(WeaponBehavior w) {
    this.weapon = w;
}
```

В местном бистро...

Элис

Мне шоколадную содовую с ванильным мороженым, горячий сэндвич с сыром и беконом, салат с тунцом на гренке, банановый сплит с мороженым и ломтиками бананов, кофе со сливками и двумя кусочками сахара... и приготовьте гамбургер на гриле!

Фло

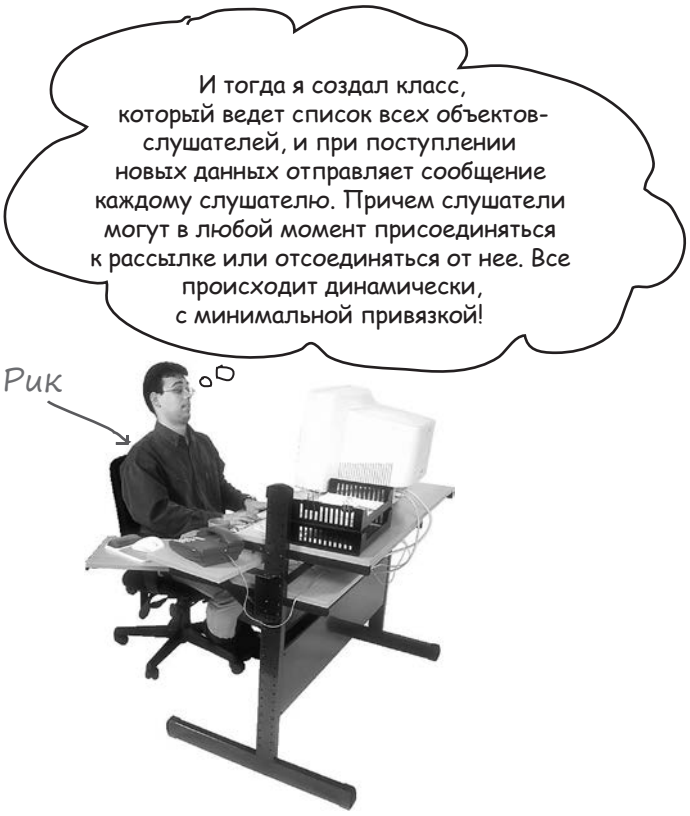
Дайте «черное с белым», «Джека Бенни», «радио», «плавучий дом», стандартный кофе и зажарьте одну штуку!

Чем различаются эти два заказа? Да ничем! Они абсолютно одинаковые, если не считать того, что Элис произнесла вдвое больше слов и едва не вывела из себя старого ворчливого официанта.

Что есть у Фло, чего нет у Элис? **Единая номенклатура с официантом**. Она не только упрощает общение, но и помогает официанту запомнить заказ, потому что все паттерны блюд хранятся у него в голове.

Паттерны проектирования формируют единую номенклатуру для разработчиков. Когда вы овладеете этой номенклатурой, вам будет проще общаться с другими разработчиками — а у тех, кто паттернов не знает, появится лишний стимул для их изучения. Кроме того, вы начнете воспринимать архитектуру на **более высоком уровне *паттернов***, а не на уровне *объектов*.

В соседнем офисе...



МОЗГОВОЙ ШТУРМ

Какие еще примеры использования единой терминологии вам известны, кроме ОО-проектирования и заказов в бистро? (Подсказка: вспомните об автомашинах, столярных работах, управлении воздушным движением.) Какие характеристики передаются при помощи специальных терминов?

А какие аспекты ОО-проектирования передаются в именах паттернов? Какие характеристики передаются в названии паттерна Стратегия?

Сила единой номенклатуры

Использование паттернов в общении не сводится к общей ТЕРМИНОЛОГИИ.

Номенклатуры паттернов обладают большой ВЫРАЗИТЕЛЬНОСТЬЮ. Используя паттерны в общении с другим разработчиком или группой, вы передаете не только название паттерна, но и целый набор характеристик, качеств и ограничений, представленных данным паттерном.

«Для реализации разных вариантов поведения используется паттерн Стратегия». Из этой фразы мы узнаем, что поведение инкапсулируется в отдельном наборе классов, который легко расширяется и изменяется — при необходимости даже во время выполнения.

Паттерны позволяют сказать больше меньшим количеством слов. Когда вы используете паттерн в описании, другие разработчики моментально понимают суть решения, о котором вы говорите.

Сколько совещаний по проектированию, в которых вам довелось участвовать, быстро вырождалось в обсуждение подробностей реализации?

Общение на уровне паттернов помогает дольше оставаться «на уровне архитектуры». Описание программной системы с использованием паттернов позволяет вести обсуждение на более абстрактном уровне, не отвлекаясь на второстепенные подробности реализации объектов и классов.

Единая номенклатура повышает эффективность разработки. Группа, хорошо разбирающаяся в паттернах проектирования, быстрее продвигается вперед, а ее участники лучше понимают друг друга.

В процессе обмена идеями и опытом, выраженными в виде паттернов, формируется сообщество пользователей паттернов.

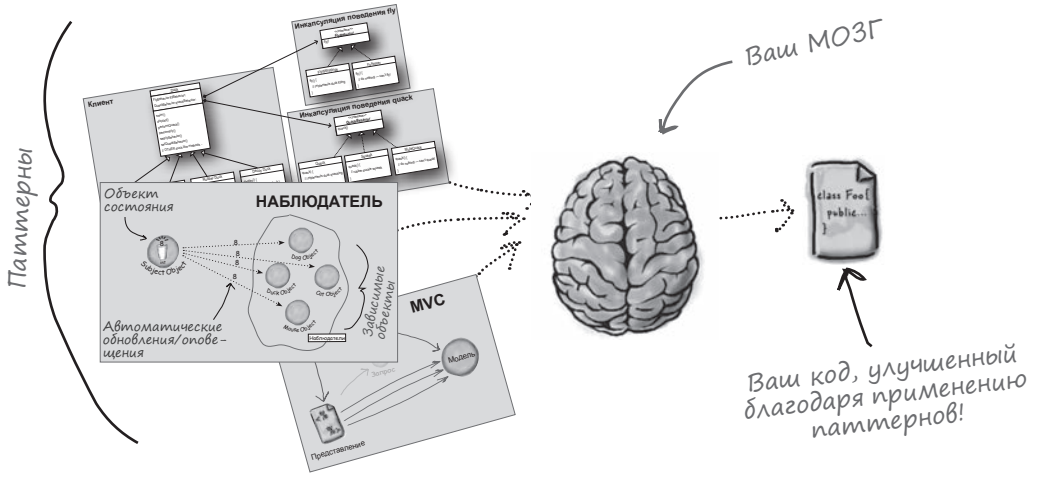
Единые номенклатуры помогают новичкам разработчикам быстрее войти в курс дела. Новички берут пример с опытных разработчиков. Если опытный разработчик применяет паттерны в своей работе, у новичков появляются дополнительные стимулы для их использования. Создайте сообщество пользователей паттернов в своей организации.

Подумайте о создании группы изучения паттернов в своей организации — возможно, время учебы даже будет оплачиваться...

Как пользоваться паттернами?

Все мы пользовались готовыми библиотеками и инфраструктурами. Мы берем их, пишем код с использованием функций API, компилируем и извлекаем пользу из кода, написанного другими людьми. Достаточно вспомнить, какую функциональность предоставляет Java API: сеть, графические интерфейсы, ввод/вывод и т. д. Однако библиотеки и инфраструктуры не помогают нам структурировать приложения так, чтобы они становились более понятными, гибкими и простыми в сопровождении. Для достижения этой цели применяются паттерны проектирования.

Паттерны не сразу воплощаются в вашем коде — сначала они должны проникнуть в ваш МОЗГ. Когда вы начнете достаточно хорошо разбираться в паттернах, вы сможете применять их в своих новых архитектурах, а также перерабатывать старый код, который со временем превращается в хаотическое месиво.



Часто Задаваемые Вопросы

В: Если паттерны так хороши, почему никто не оформил их в виде библиотеки?

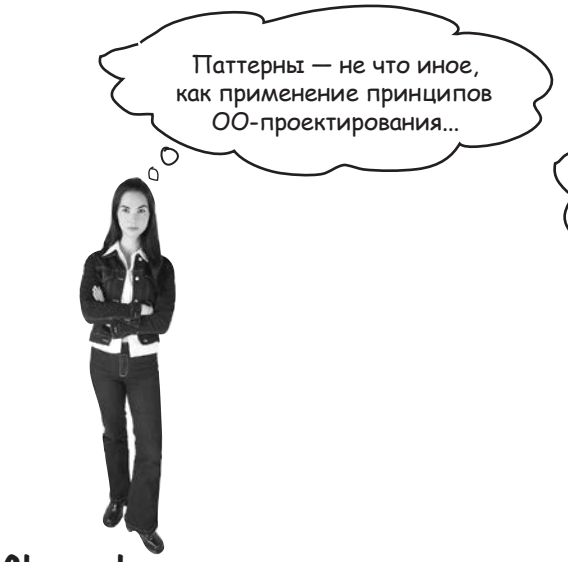
О: Паттерны относятся к более высокому уровню, чем библиотеки. Они определяют способы структурирования классов и объектов для решения некоторых задач, а наша задача — адаптировать их для своих конкретных приложений.

В: А библиотеки и инфраструктуры не являются паттернами проектирования?

О: Нет, не являются; они предоставляют конкретные реализации, которые мы связываем со своим кодом. Впрочем, иногда паттерны используются в реализациях библиотек. И это очень хорошо, потому что понимание паттернов поможет быстрее разобраться в API, архитектура которых базируется на паттернах.

В: Так, значит, библиотек паттернов не существует?

О: Нет, но позднее вы узнаете о каталогах паттернов, которые могут применяться в ваших приложениях.



Разработчик: Хм, но разве дело не сводится к ОО-проектированию? Если я следую принципам инкапсуляции, знаю об абстракции, наследовании и полиморфизме, то зачем мне думать о паттернах проектирования? Для чего тогда были нужны те курсы ОО-проектирования? Я думаю, паттерны проектирования полезны только тем, кто не разбирается в ОО-проектировании.

Гуру: О, это одно из известных заблуждений объектно-ориентированной разработки: будто знание основ ООП автоматически позволит вам строить гибкие, удобные в сопровождении и пригодные к повторному использованию системы.

Разработчик: Нет?

Гуру: Нет. Более того, принципы построения ОО-систем, обладающих такими свойствами, далеко не всегда очевидны.

Разработчик: Кажется, я начинаю понимать. И на основе этих неочевидных принципов построения объектно-ориентированных систем были сформулированы...

Гуру: ...да, были сформулированы паттерны проектирования.

Знание таких концепций, как абстракция, наследование и полиморфизм, еще не делает из вас хорошего ОО-проектировщика. Истинный гуру проектирования стремится создавать гибкие архитектуры, способные адаптироваться к изменениям.

Разработчик: Выходит, зная паттерны, я могу пропустить все технические подробности, а мои решения всегда будут работать?

Гуру: Да, до определенной степени, но не забывайте: проектирование — это искусство, а не ремесло. Компромиссы неизбежны, но если вы будете использовать хорошо продуманные и проверенные временем схемы, ваша работа значительно упростится.

Разработчик: А если я не могу найти подходящий паттерн?

Гуру: Существуют некоторые объектно-ориентированные принципы, заложенные в основу паттернов. Знание этих принципов поможет найти выход из ситуации, для которой вам не удается найти подходящий паттерн.

Разработчик: Принципы? Не считая абстракции, инкапсуляции и...

Гуру: Да, один из секретов построения гибких ОО-систем заключается в прогнозировании их возможных будущих изменений. В этом нам и помогают принципы, о которых я говорю.

Новые инструменты

Первая глава почти завершена! В ней ваш ОО-инструментарий дополнился несколькими новыми инструментами. Давайте вспомним их, прежде чем переходить к главе 2.

Концепции ООП

Абстракция
Инкапсуляция
Полиморфизм
Наследование

Предполагается, что вы уже понимаете основные концепции ООП: полиморфное использование классов, инкапсуляция и т. д.

Принципы

Инкапсулируйте то, что изменяется.

Отдавайте предпочтение композиции перед наследованием.

Программируйте на уровне интерфейсов, а не реализации.

Мы рассмотрим эти принципы более подробно, а также дополним список новыми принципами.

Паттерны

Стратегия — определяет семейство алгоритмов, инкапсулирует и обеспечивает их взаимозаменяемость. Паттерн позволяет модифицировать алгоритмы независимо от их использования на стороне клиента.

Во время чтения обращайте внимание на связь паттернов с основами и принципами ООП.

Один паттерн пройден, но впереди еще много!

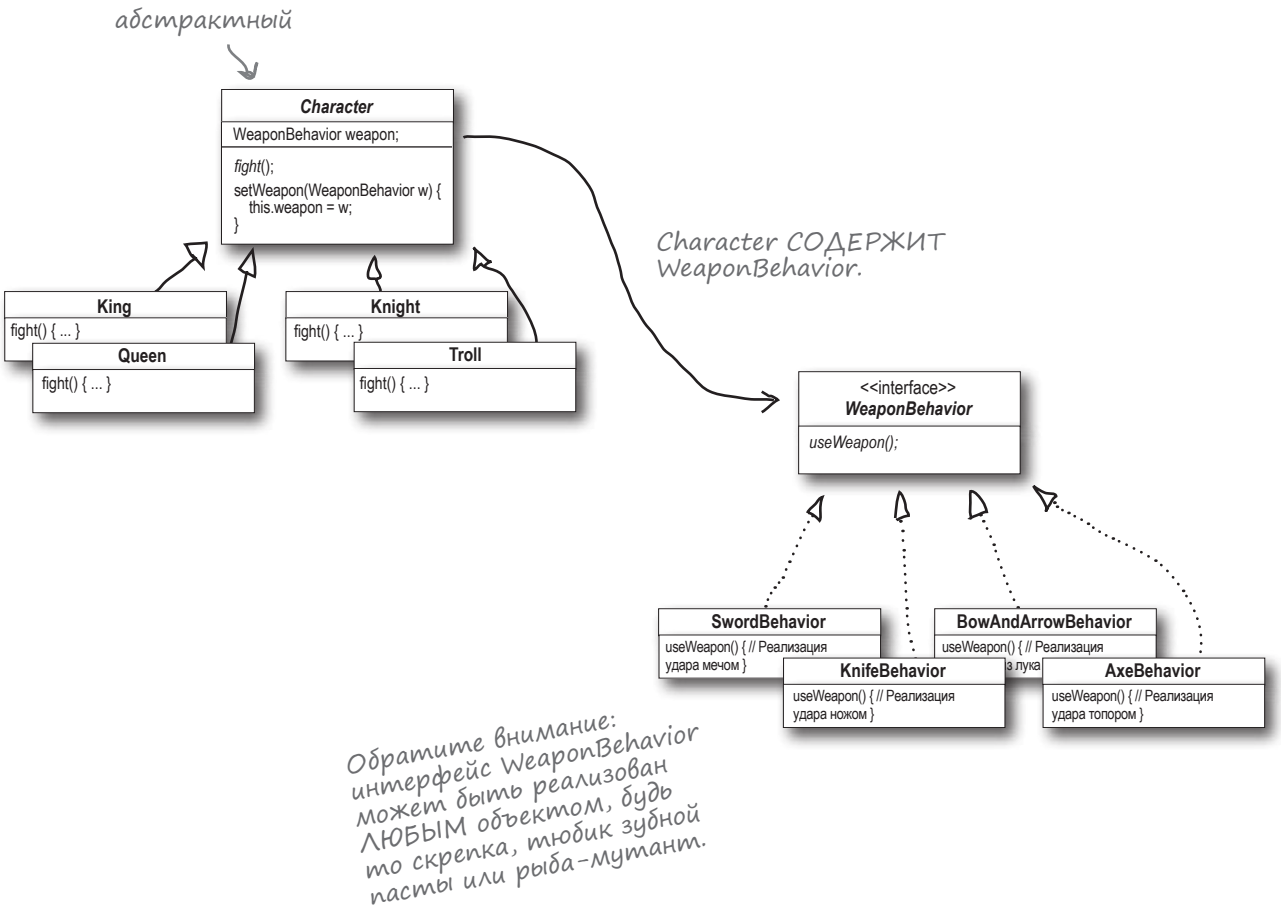
КЛЮЧЕВЫЕ МОМЕНТЫ

- Знание основ ООП не сделает из вас хорошего ОО-проектировщика.
- Хорошие ОО-архитектуры хорошо расширяются, просты в сопровождении и пригодны для повторного использования.
- Паттерны показывают, как строить системы с хорошими качествами ОО-проектирования.
- Паттерны содержат проверенный опыт ОО-проектирования.
- Паттерны описывают общие решения проблем проектирования и применяются в конкретных приложениях.
- Паттерны не *придумывают* — их *находят*.
- Большинство паттернов и принципов направлено на решение проблем изменения программных архитектур.
- Многие паттерны основаны на инкапсуляции переменных аспектов системы.
- Паттерны образуют единую номенклатуру, которая повышает эффективность вашего общения с другими разработчиками.

Ответы к упражнениям

Character — абстрактный суперкласс для всех классов персонажей (King, Queen, Knight и Troll), а интерфейс WeaponBehavior реализуется всеми классами поведения оружия. Все классы персонажей и оружия являются конкретными.

Чтобы сменить оружие, персонаж вызывает метод setWeapon(), определяемый в суперкласс Character. Во время сражения для текущего оружия персонажа вызывается метод useWeapon().



Возьми в руку карандаш
Решение

Какие из перечисленных недостатков относятся к применению *наследования* для реализации Duck? (Укажите все варианты.)

- ☑ A. Дублирование кода в субклассах.
- ☑ B. Трудности с изменением поведения на стадии выполнения.
- ☐ C. Уток нельзя научить танцевать.
- ☑ D. Трудности с получением информации обо всех аспектах поведения уток.
- ☑ E. Утки не могут летать и крякать одновременно.
- ☐ F. Изменения могут оказать непредвиденное влияние на другие классы.

Возьми в руку карандаш
Решение

Какие факторы могут вызвать изменения в вашем приложении? Возможно, ваш список будет выглядеть совершенно иначе... А может, и в нашем списке многое покажется знакомым?

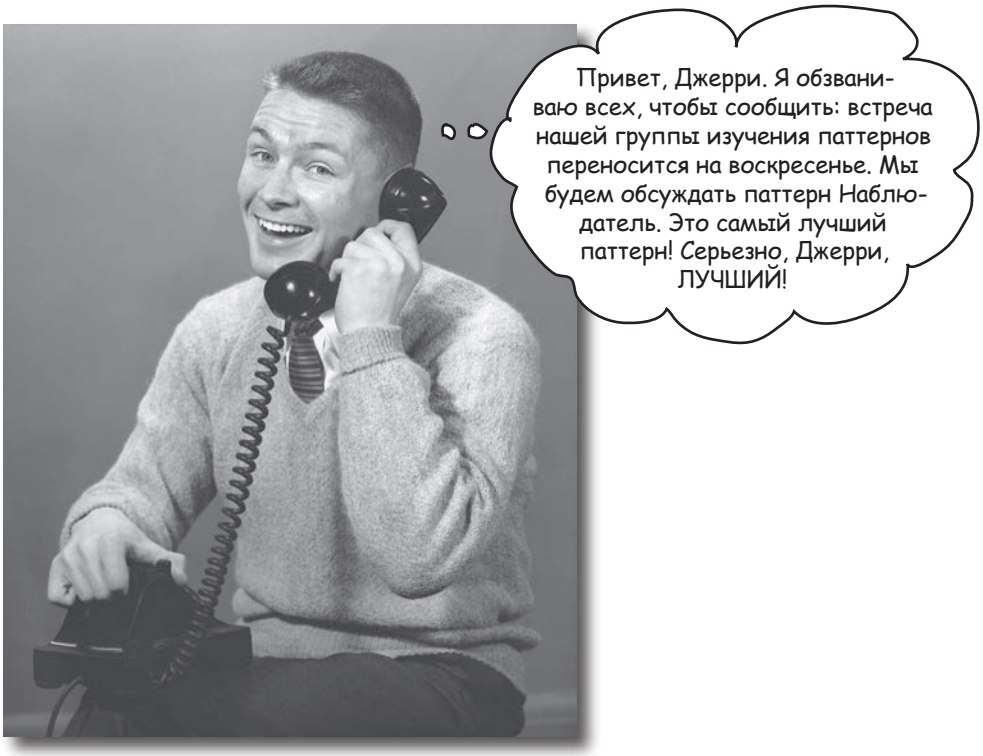
Клиенты или пользователи требуют реализации новой или расширенной функциональности.

Компания переходит на другую СУБД, а данные будут приобретаться у другого поставщика в новом формате. Ужас!

Технология изменилась, и код необходимо изменить для использования новых протоколов.

В ходе построения системы мы получили много полезной информации. Теперь мы хотим вернуться и немного доработать исходную архитектуру.

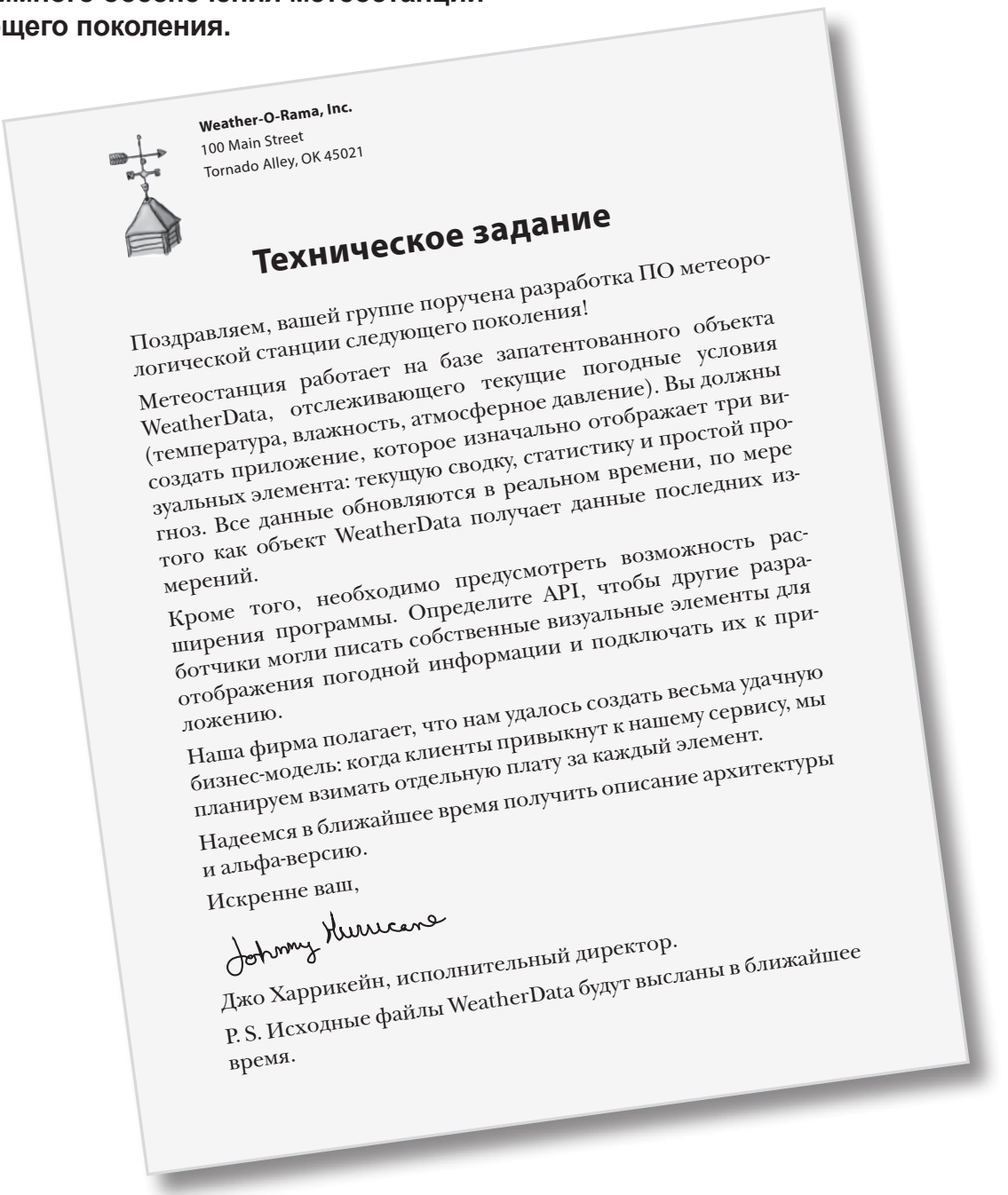
Объекты в курсе событий



Не упустите, когда происходит что-то интересное! Наш следующий паттерн оповещает объекты о наступлении неких событий, которые могут представлять для них интерес, — причем объекты даже могут решать во время выполнения, желают ли они и дальше получать информацию. Паттерн Наблюдатель чрезвычайно полезен и принадлежит к числу наиболее часто используемых паттернов JDK. Также в этой главе будут рассмотрены связи типа «один-ко-многим» и слабые связи. С паттерном Наблюдатель вы станете душой Общества Паттернов.

Поздравляем!

Ваша группа только что выиграла контракт на построение программного обеспечения метеостанции следующего поколения.



Weather-O-Rama, Inc.
100 Main Street
Tornado Alley, OK 45021

Техническое задание

Поздравляем, вашей группе поручена разработка ПО метеорологической станции следующего поколения!

Метеостанция работает на базе запатентованного объекта WeatherData, отслеживающего текущие погодные условия (температура, влажность, атмосферное давление). Вы должны создать приложение, которое изначально отображает три визуальных элемента: текущую сводку, статистику и простой прогноз. Все данные обновляются в реальном времени, по мере того как объект WeatherData получает данные последних измерений.

Кроме того, необходимо предусмотреть возможность расширения программы. Определите API, чтобы другие разработчики могли писать собственные визуальные элементы для отображения погодной информации и подключать их к приложению.

Наша фирма полагает, что нам удалось создать весьма удачную бизнес-модель: когда клиенты привыкнут к нашему сервису, мы планируем взимать отдельную плату за каждый элемент.

Надеемся в ближайшее время получить описание архитектуры и альфа-версию.

Искренне ваш,

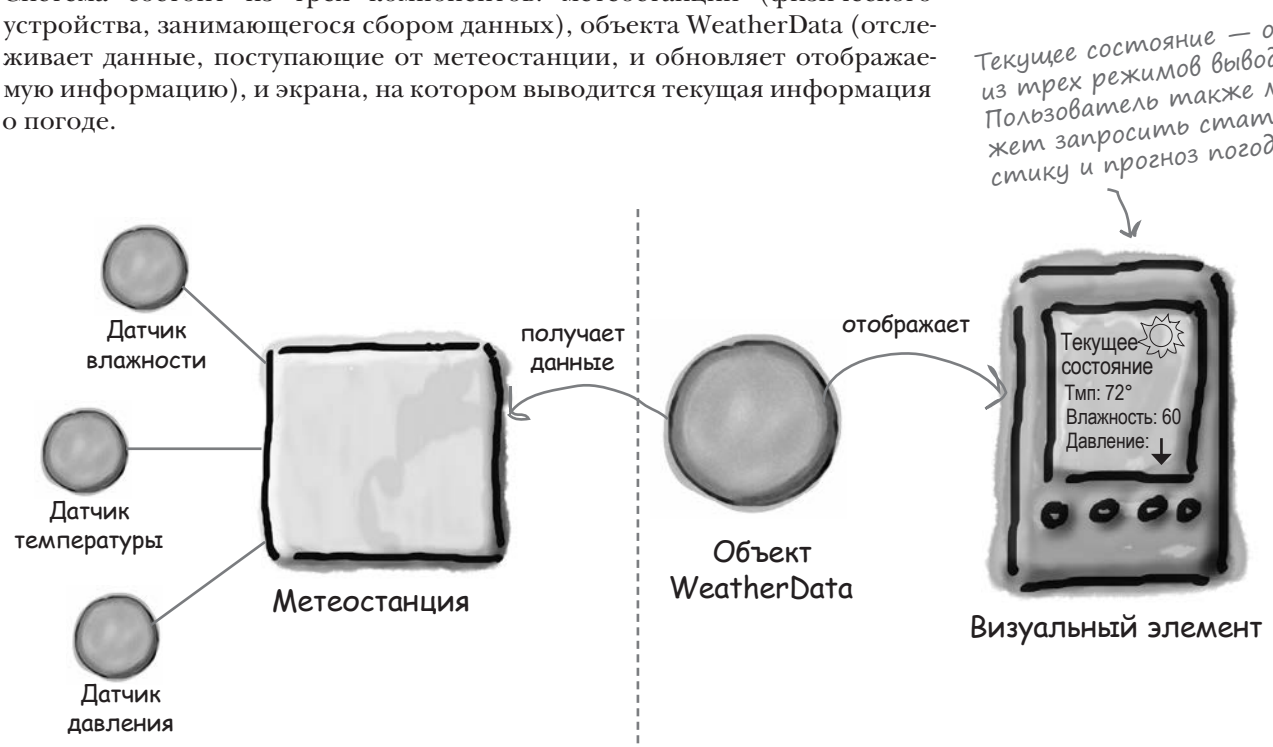
Johnny Hurricane

Джо Харрикейн, исполнительный директор.

P. S. Исходные файлы WeatherData будут высланы в ближайшее время.

Обзор приложения Weather Station

Система состоит из трех компонентов: метеостанции (физического устройства, занимающегося сбором данных), объекта WeatherData (отслеживает данные, поступающие от метеостанции, и обновляет отображаемую информацию), и экрана, на котором выводится текущая информация о погоде.

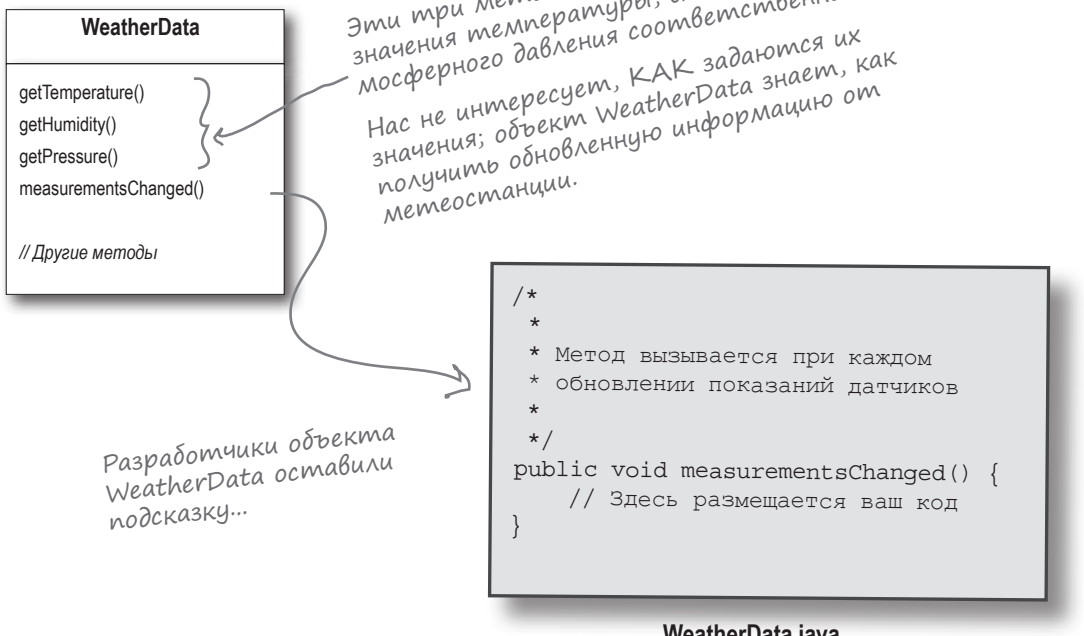


Объект WeatherData умеет получать от физической метеостанции обновленные данные. Затем объект WeatherData обновляет изображение для трех основных элементов: текущего состояния (температура, влажность и давление), статистики и прогноза.

Наша задача — создать приложение, которое использует данные объекта WeatherData для обновления текущих условий, статистики и прогноза погоды.

Как устроен класс WeatherData

Как и было обещано, на следующее утро вы получаете исходные файлы WeatherData. Код выглядит достаточно просто:



Метод measurementsChanged() необходимо реализовать так, чтобы он обновлял изображение для трех элементов: текущего состояния, статистики и прогноза.

Что нам известно?

Спецификация была довольно невразумительной, но мы должны понять, что нам предстоит сделать. Итак, что мы знаем на данный момент?

- Класс WeatherData содержит get-методы для показаний трех датчиков: температуры, влажности и атмосферного давления.

```
getTemperature()
getHumidity()
getPressure()
```

- Метод measurementsChanged() вызывается при появлении новых метеорологических данных. (Мы не знаем, как он вызывается, да это и неважно; достаточно знать, что он *вызывается*.)

```
measurementsChanged()
```

- Необходимо реализовать три экрана вывода, использующих метеорологические данные: экран *текущего состояния*, экран *статистики* и экран *прогноза*. Эти экраны должны обновляться каждый раз, когда у объекта WeatherData появляются новые данные.

Первый элемент

Второй элемент

Третий элемент

- Система должна быть расширяемой — другие разработчики будут создавать новые экраны вывода, а пользователи могут добавлять и удалять их в своих приложениях. В настоящее время определены всего три вида экранов (текущее состояние, статистика и прогноз).

Будущие элементы

Первое, что приходит в голову

Первый вариант реализации, который приходит в голову: мы следуем совету разработчиков WeatherData и включаем свой код в метод measurementsChanged():

```
public class WeatherData {

    // Объявления переменных экземпляров

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // Другие методы WeatherData
}
```

Чтобы получить обновленные данные, мы вызываем (уже реализованные) get-методы класса WeatherData

Обновляем показания...

Запрос на перерисовку каждого элемента с передачей обновленных значений.

Возьми в руку карандаш

Какие из следующих утверждений относятся к первой реализации? (Укажите все варианты.)

- ❑ A. Мы программируем на уровне реализаций, а не интерфейсов.
- ❑ B. Для каждого нового элемента придется изменять код.
- ❑ C. Элементы не могут добавляться (или удаляться) во время выполнения.
- ❑ D. Элементы не реализуют единый интерфейс.
- ❑ E. Переменные аспекты архитектуры не инкапсулируются.
- ❑ F. Нарушается инкапсуляция класса WeatherData.

Чем плоха такая реализация?

Вспомните концепции и принципы из главы 1...

```
public void measurementsChanged() {

    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}
```

Переменная область — необходимо инкапсулировать.

Нечто похожее на общий интерфейс взаимодействия с экранными элементами... Каждый элемент имеет метод update(), которому передаются значения температуры, влажности и давления.

Программируя на уровне конкретной реализации, мы не сможем добавлять и удалять визуальные элементы без внесения изменений в программу.

Ммм... Я, конечно, новичок, но раз уж глава посвящена паттерну Наблюдатель — так, может, мы им воспользуемся?

Сначала мы рассмотрим паттерн Наблюдатель, а потом вернемся и посмотрим, как применить его в этом приложении.

Знакомство с паттерном Наблюдатель

Всем известно, как работает подписка на газету или журнал:

1. Издатель открывает свое дело и начинает выпускать газету.

2. Вы оформляете подписку у конкретного издателя. Каждый раз, когда выходит новый номер, он доставляется вам. Пока подписка действует, вы получаете новые выпуски газеты.

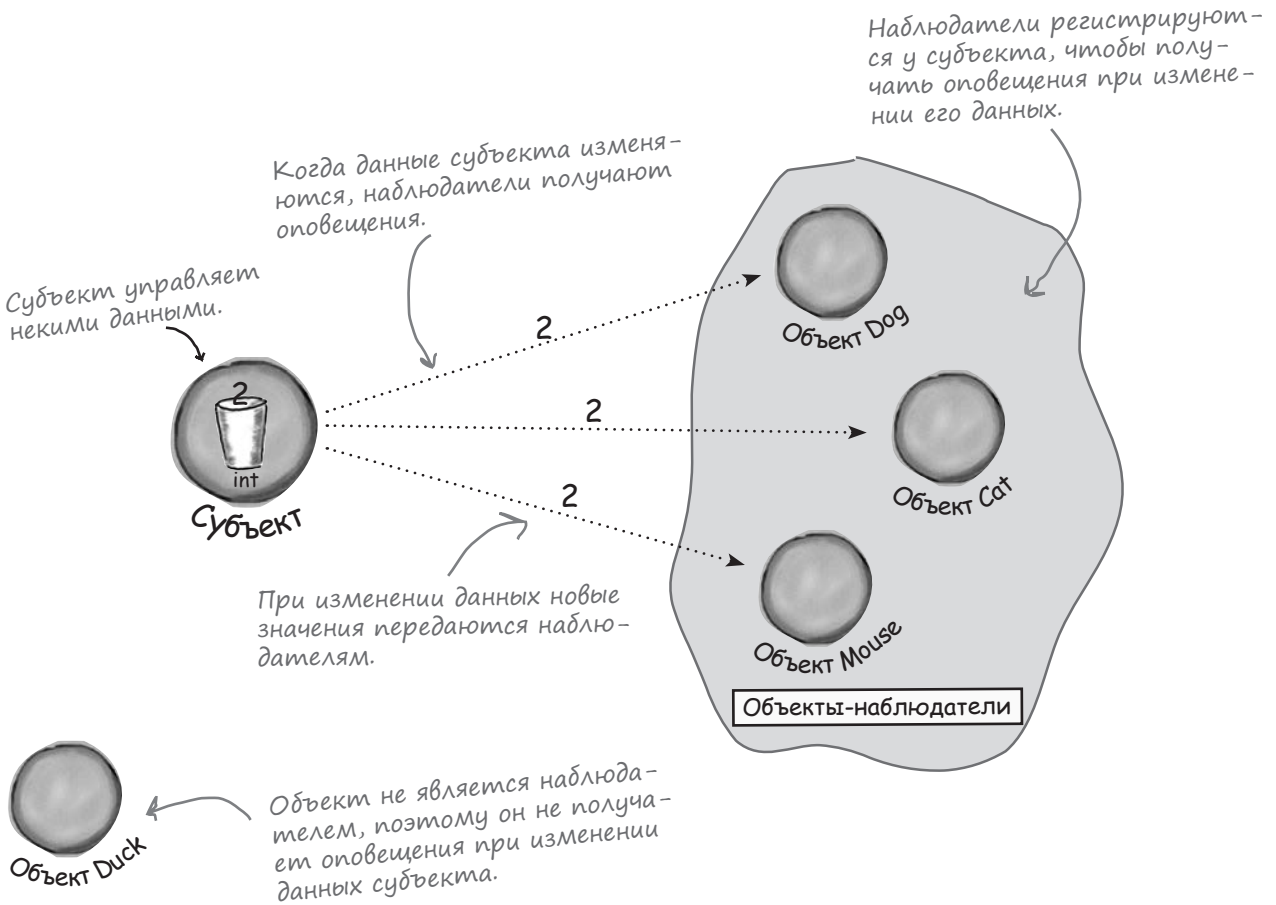
3. Если вы не хотите больше получать газету, вы прекращаете подписку.

4. Пока газета продолжает публиковаться, люди, гостиницы, авиалинии и т. д. постоянно оформляют и прекращают подписку.

Издатели + Подписчики = Паттерн Наблюдатель

Если вы понимаете, как работает газетная подписка, вы в значительной мере понимаете и паттерн Наблюдатель — только в данном случае издатель называется СУБЪЕКТОМ, а подписчики — НАБЛЮДАТЕЛЯМИ.

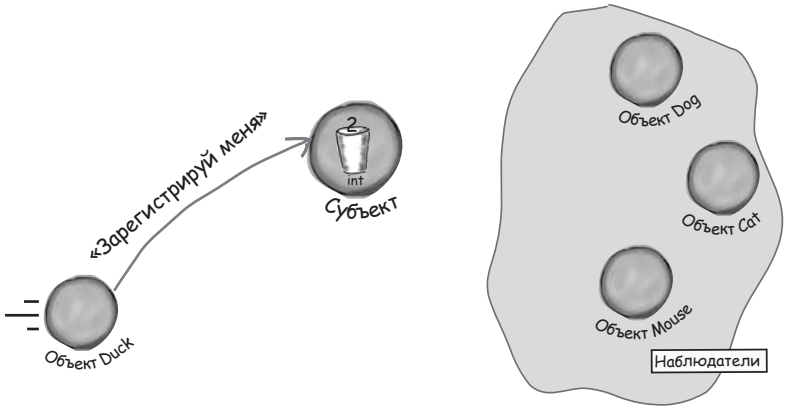
Присмотримся повнимательнее



Один день из жизни паттерна Наблюдатель

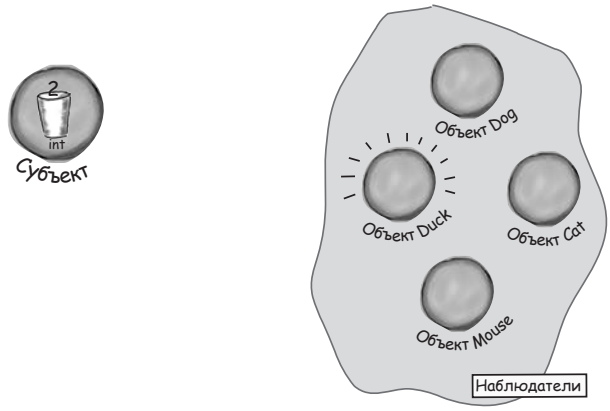
Объект Duck сообщает субъекту, что он хочет стать наблюдателем.

Объект Duck хочет быть в курсе дела; эти значения int, которые субъект рассылает при изменении состояния, выглядят так интересно...



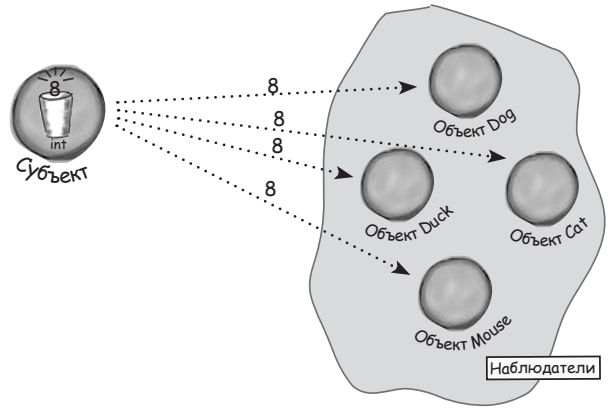
Объект Duck стал официальным наблюдателем.

Объект Duck включен в список... Теперь он с нетерпением ждет следующего оповещения, с которым он получит интересующее его значение int.



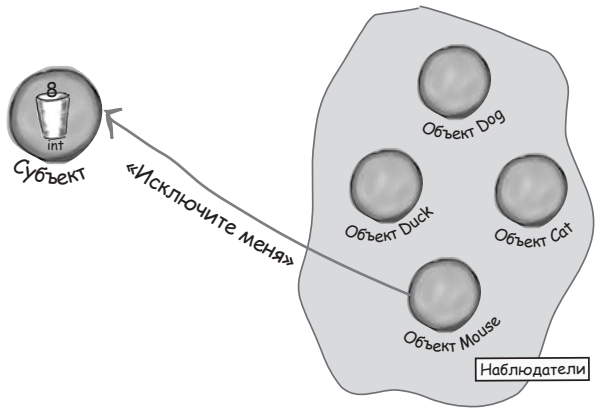
У субъекта появились новые данные!

Duck и все остальные наблюдатели оповещаются об изменении состояния субъекта.



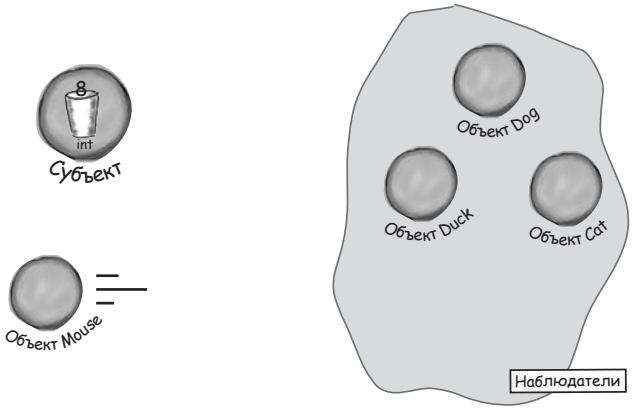
Объект Mouse требует исключить его из числа наблюдателей.

Объекту Mouse надоело получать оповещения, и он решил, что пришло время выйти из числа наблюдателей.



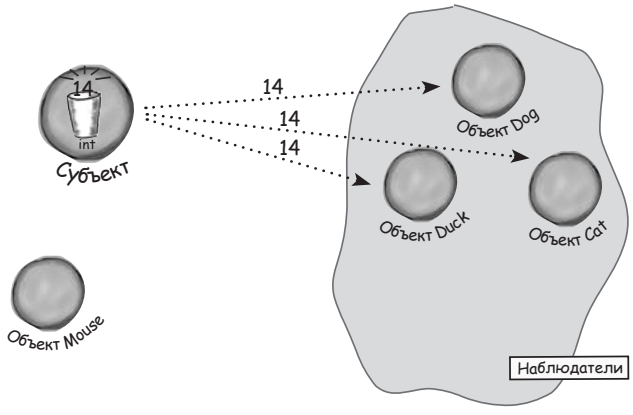
Mouse уходит!

Субъект принимает запрос объекта Mouse и исключает его из числа наблюдателей.



У субъекта появилось новое значение int.

Все наблюдатели получают очередное оповещение — кроме объекта Mouse, который исключен из списка. Не говорите никому, но он тайно скучает по этим оповещениям... и, возможно, когда-нибудь снова войдет в число наблюдателей.



Пятиминутная драма: субъект для наблюдения

В сегодняшней серии два программиста, переживших крах «доткомов», встречают настоящего «охотника за головами»...

1

Программист №1

2

Субъект/«Охотник за головами»

3

Программист №2

4

Субъект

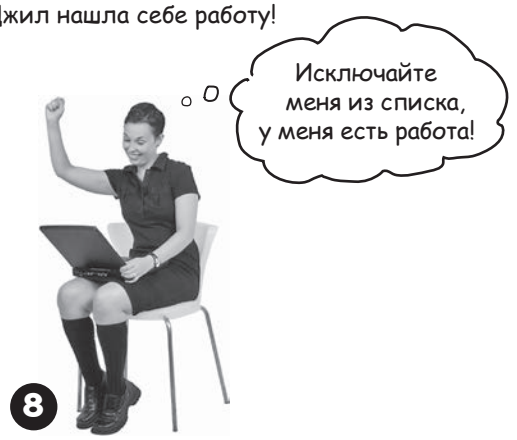
5 Тем временем жизнь Лори и Джил продолжается. Если появится вакансия Java-программиста, они об этом узнают — ведь они стали наблюдателями.

Субъект

Наблюдатель

Наблюдатель

Джил нашла себе работу!



Наблюдатель

Субъект

Прошло две недели...

Джил вышла из числа наблюдателей и наслаждается жизнью. Кроме того, она получила неплохую поощрительную премию при вступлении в должность, потому что компании не пришлось оплачивать услуги «охотника за головами».

А что случилось с нашей дорогой Лори? Говорят, она сама стала подрабатывать поиском вакансий. Теперь она не только остается в списке, но и завела собственный список. Таким образом, Лори одновременно является и субъектом, и наблюдателем.

Определение паттерна Наблюдатель

Если вы пытаетесь мысленно представить паттерн Наблюдатель, модель подписки с издателями и подписчиками дает неплохое представление о ней.

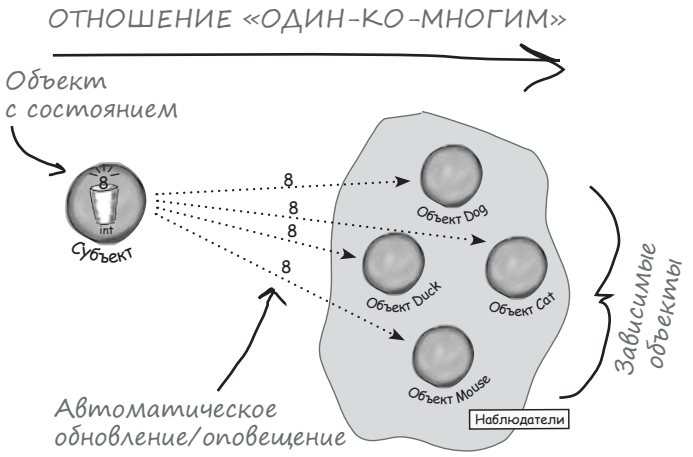
А в реальном мире паттерн Наблюдатель обычно определяется так:

> **Паттерн Наблюдатель** определяет отношение «один-ко-многим» между объектами таким образом, что при изменении состояния одного объекта происходит автоматическое оповещение и обновление всех зависимых объектов.

Паттерн Наблюдатель определяет отношение типа «один-ко-многим» между объектами.

Когда состояние одного объекта изменяется, все зависимые объекты получают оповещения.

Давайте посмотрим, как это определение соответствует нашей модели паттерна.

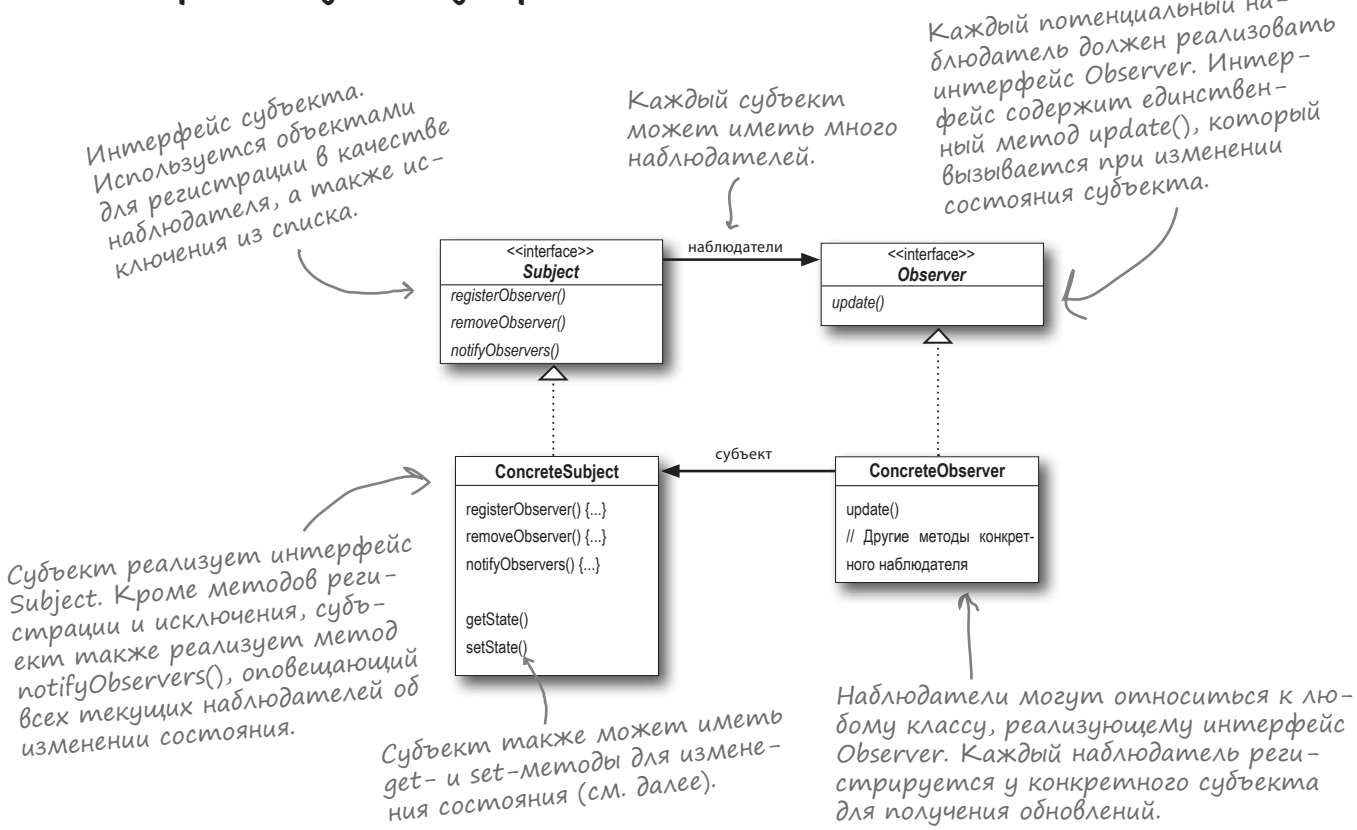


Субъект и наблюдатели определяют отношение «один-ко-многим». Наблюдатели зависят от субъекта: при изменении состояния последнего наблюдатели получают оповещения. В зависимости от способа оповещения также возможно обновление состояния наблюдателей.

Как вы вскоре увидите, существует много разных вариантов реализации паттерна Наблюдатель, но большинство из них строится на основе классов, реализующих интерфейсы субъекта или наблюдателя.

Давайте посмотрим...

Паттерн Наблюдатель: диаграмма классов



Часто Задаваемые Вопросы

В: При чем здесь отношения «один-ко-многим»?

О: В паттерне Наблюдатель субъект обладает состоянием и управляет им. Таким образом, существует ОДИН субъект, обладающий состоянием. С другой стороны, наблюдатели используют состояние, хотя и не обладают им. Они зависят от субъекта, который оповещает их об изменении состояния. Возникает отношение, в котором участвует ОДИН субъект и МНОГО наблюдателей.

В: При чем здесь зависимости?

О: Так как субъект является единоличным владельцем данных, работа наблюдателей зависит от субъекта, оповещающего их об изменении данных. Так формируется элегантная ОО-структура, в которой многие объекты используют одни и те же данные.

Сила слабых связей

Если два объекта могут взаимодействовать, не обладая практически никакой информацией друг о друге, такие объекты называют *слабосвязанными*.

В архитектуре паттерна Наблюдатель между субъектами и наблюдателями существует слабая связь. Почему?

Единственное, что знает субъект о наблюдателе, — то, что тот реализует некоторый интерфейс (Observer). Ему не нужно знать ни конкретный класс наблюдателя, ни его функциональность... ничего.

Новые наблюдатели могут добавляться в любой момент. Так как субъект зависит только от списка объектов, реализующих интерфейс Observer, вы можете добавлять новых наблюдателей по своему усмотрению. Любого наблюдателя во время выполнения можно заменить другим наблюдателем или исключить его из списка — субъект этого не заметит.

Сколько разных видов изменений вы здесь насчитаете?

Добавление новых типов наблюдателей не требует модификации субъекта. Допустим, у нас появился новый класс, который должен стать наблюдателем. Вносить изменения в субъект не потребуется — достаточно реализовать интерфейс Observer в новом классе и зарегистрировать его в качестве наблюдателя. Субъект будет доставлять оповещения любому объекту, реализующему интерфейс Observer.

Субъекты и наблюдатели могут повторно использоваться независимо друг от друга. Между ними не существует сильных связей, что позволяет повторно использовать их для других целей.

Изменения в субъекте или наблюдателе не влияют на другую сторону. Благодаря слабым связям мы можем вносить любые изменения на любой из двух сторон — при условии, что объект реализует необходимый интерфейс субъекта или наблюдателя.

Принцип проектирования

Стремитесь к слабой связанности взаимодействующих объектов.

На базе слабосвязанных архитектур строятся гибкие ОО-системы, которые хорошо адаптируются к изменениям благодаря минимальным зависимостям между объектами.

Возьми в руку карандаш

Прежде чем двигаться дальше, попробуйте составить предварительную диаграмму классов проекта Weather Station, включая класс WeatherData и его визуальные элементы. На диаграмме должны быть обозначены связи между всеми компонентами, а также механизмы реализации визуальных элементов другими разработчиками.

Если вам понадобится помощь, обратитесь к следующей странице. На ней ваши коллеги уже обсуждают архитектуру Weather Station.

Разговор в офисе

Тем временем ваши коллеги по проекту Weather Station уже начали обдумывать проблему...

Мэри: Надо воспользоваться паттерном Наблюдатель.

Сью: Верно... Но как мы будем его применять?

Мэри: Хмм... Давай еще раз взглянем на определение:

> *Паттерн Наблюдатель определяет отношение «один-ко-многим» между объектами таким образом, что при изменении состояния одного объекта происходит автоматическое оповещение и обновление всех зависимых объектов.*

Мэри: В принципе вполне логично. Класс WeatherData — «один», а разные визуальные элементы, использующие данные метеостанции, — «многие».

Сью: Точно. Класс WeatherData обладает состоянием — это температура, влажность и давление. И, безусловно, состояние будет изменяться.

Мэри: И при изменении состояния мы оповещаем визуальные элементы, чтобы они могли поступить с новыми данными по своему усмотрению.

Сью: Отлично. Я вижу, что паттерн Наблюдатель подходит для нашей задачи.

Мэри: Однако некоторые моменты мне пока непонятны.

Сью: Например?

Мэри: Скажем, как мы будем передавать обновленные данные визуальным элементам?

Сью: Давай-ка еще раз посмотрим на диаграмму паттерна Наблюдатель... Если сделать объект WeatherData субъектом, а визуальные элементы наблюдателями, то элементы будут регистрироваться у объекта WeatherData для получения нужной информации?

Мэри: Да... и если субъект будет располагать информацией о визуальном элементе, то он сможет просто вызвать его метод для передачи данных.

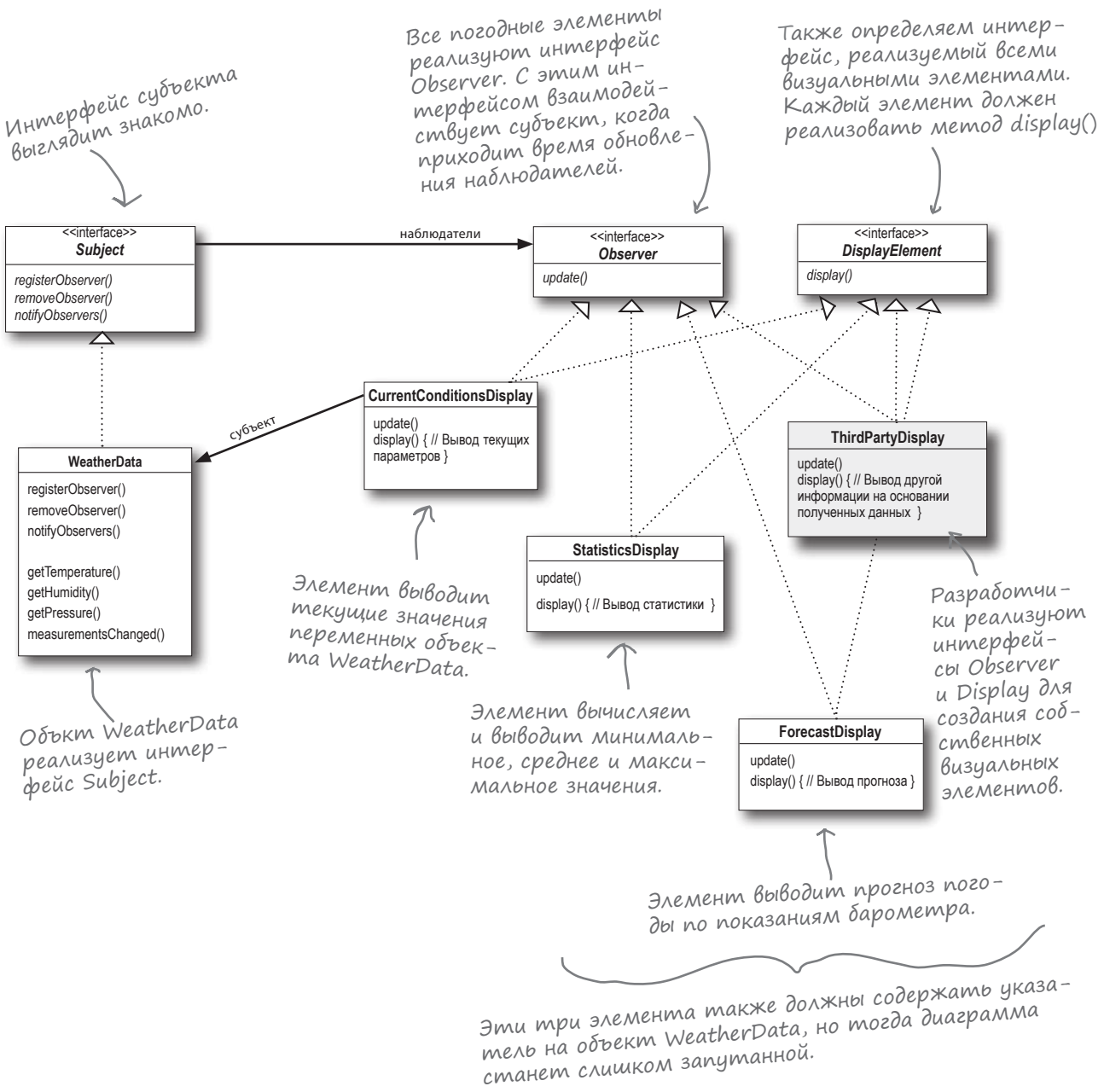
Сью: Но визуальные элементы такие разные... Похоже, здесь уместно воспользоваться общим интерфейсом. Хотя все компоненты относятся к разным типам, они реализуют общий интерфейс, поэтому объект WeatherData всегда будет знать, как передать им обновленные данные.

Мэри: Выходит, каждый визуальный элемент будет поддерживать метод... допустим, update(), который будет вызываться объектом WeatherData?

Сью: Да, причем метод update() будет определяться в общем интерфейсе, который реализуется всеми элементами.

Проектирование Weather Station

Похожа ли эта диаграмма на ту, которую нарисовали вы?



Реализация Weather Station

Мы начнем строить свою реализацию на основании диаграммы классов и приводившихся ранее рекомендаций Мэри и Сью. Как будет показано далее, в языке Java предусмотрена встроенная поддержка паттерна Наблюдатель, но мы сделаем собственную реализацию. В некоторых случаях встроенной поддержки Java бывает достаточно, но самостоятельная реализация обладает большей гибкостью. Начнем с интерфейсов:

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

Оба метода получают в аргументе реализацию Observer (регистрируемый или исключаемый наблюдатель).

Этом метод вызывается для оповещения наблюдателей об изменении состояния субъекта.

```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

Данные состояния, передаваемые наблюдателям при изменении состояния субъекта.

Интерфейс Observer реализуется всеми наблюдателями; таким образом, все наблюдатели должны реализовать метод update().

```
public interface DisplayElement {
    public void display();
}
```

Интерфейс DisplayElement содержит всего один метод display(), который вызывается для отображения визуального элемента.

МОЗГОВОЙ ШТУРМ

Мэри и Сью считают, что прямая передача данных наблюдателям является самым простым способом обновления состояния. Насколько это разумно, по вашему мнению? Подсказка: может ли эта область приложения измениться в будущем? А если изменится, то будут ли изменения надежно инкапсулированы, или изменения придется вносить во многих местах кода?

Можете ли вы предложить другие способы передачи обновленного состояния наблюдателям?

Мы еще вернемся к этому аспекту архитектуры после завершения исходной реализации.

Реализация интерфейса Subject в WeatherData

Помните нашу первую попытку реализации класса WeatherData в начале главы? Пора вернуться и привести ее в порядок, ориентируясь на паттерн Наблюдатель.

ВНИМАНИЕ: директивы import и package в листингах не приводятся. Полный исходный код можно загрузить на сайте http://wickedlysmart.com/headfirst-design-patterns/.

```
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    // Другие методы WeatherData
}
```

Реализация интерфейса Subject.

Теперь WeatherData реализует интерфейс Subject.

Добавляем контейнер ArrayList для хранения наблюдателей и создаем его в конструкторе.

Новые наблюдатели просто добавляются в конец списка.

Если наблюдатель хочет отменить регистрацию, мы просто удаляем его из списка.

Самое интересное: оповещение наблюдателей об изменении состояния через метод update(), реализуемый всеми наблюдателями.

Оповещение наблюдателей о появлении новых данных.

Приложить метеостанцию к каждому экземпляру книги нам не разрешили, поэтому вместо чтения данных с устройства мы воспользуемся тестовым методом. При желании вы можете написать код для загрузки погодных данных из Интернета.

Переходим к визуальным элементам

Итак, мы разобрались с классом WeatherData; пришло время заняться визуальными элементами. В задании перечислены три элемента: для вывода текущего состояния, статистики и прогноза. Начнем с текущего состояния; когда вы достаточно хорошо поймете его код, рассмотрите другие примеры из архива на сайте книги — они очень похожи.

Элемент реализует Observer, чтобы получать данные от объекта WeatherData.

Также он реализует интерфейс DisplayElement, как и все визуальные элементы в нашем API.

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

Конструктору передается объект WeatherData, который используется для регистрации элемента в качестве наблюдателя.

При вызове update() мы сохраняем значения температуры и влажности, после чего вызываем display().

Метод display() просто выводит текущие значения температуры и влажности.

Часто Задаваемые Вопросы

В: Правильно ли вызывать display() в методе update()?

О: В нашем простом примере метод display() логично вызывать при изменении данных. Однако вы правы, существуют и более элегантные способы проектирования отображения данных. Они будут представлены при рассмотрении паттерна «модель–представление–контроллер».

В: Зачем сохранять ссылку на Subject, если она не используется после вызова конструктора?

О: Верно, но в будущем мы реализуем отмену регистрации наблюдателей. Для этого будет удобно иметь готовую ссылку на реализацию Subject.

Тестирование Weather Station

❶ Для начала напишем тестовую программу.

Первая версия Weather Station готова; нужен лишь код, который свяжет воедино все компоненты. Позднее мы еще вернемся к этому проекту и реализуем простое подключение компонентов через конфигурационный файл. А пока это будет происходить так:

```
public class WeatherStation {
    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay =
            new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);

    }
}
```

Сначала создаем объект WeatherData.

Если вы не хотите загружать код с сайта, закомментируйте эти две строки.

Создаем три визуальных элемента, передавая им объект WeatherData.

Имитация новых погодных данных.

❷ Выполняем код и следим за тем, как работает паттерн Наблюдатель.

```
File  Edit  Window  Help  StormyWeather

%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
%
```

Беседа у камина

Субъект и Наблюдатель обсуждают механизм получения данных состояния.

Субъект

Я рад, что у нас наконец-то появилась возможность побеседовать лично.

Ну я же делаю свое дело, верно? Я всегда сообщаю вам о том, что происходит... И хотя я не знаю, кто вы, это не значит, что мне это безразлично. К тому же я знаю самое главное — вы реализуете интерфейс Observer.

Вот как? Например?

Ах, *извините*. Я должен передавать свое состояние с оповещениями, чтобы ленивые наблюдатели были в курсе дела!

В принципе такое возможно. Но мне придется ослабить ограничения доступа, чтобы вы, наблюдатели, могли прийти и получить нужную информацию. А это, знаете ли, рискованно. Я не могу позволить всем желающим копаться в своих личных данных.

Наблюдатель

Правда? Я думал, что вы не обращаете никакого внимания на нас, наблюдателей.

Да, но это далеко не все. К тому же о вас я знаю намного больше...

Вы постоянно передаете нам, наблюдателям, данные состояния, чтобы мы знали, что у вас происходит. Иногда это начинает слегка раздражать...

Одну минуту; во-первых, мы не ленивые — просто нам приходится заниматься другими делами между вашими *важными* оповещениями. А во-вторых, почему бы вам не разрешить нам обращаться за данными, когда мы этого захотим, вместо того, чтобы заталкивать их насильно?

Субъект

Да, я могу разрешить вам **запрашивать** данные состояния. Но разве это не будет менее удобно для вас? Возможно, для получения всех необходимых данных вам придется многократно обращаться ко мне с вызовами. Вот почему я предпочитаю активную **доставку**... Все необходимое отправляется за один вызов.

Что ж, у каждого варианта есть свои преимущества. Кстати, встроенная реализация паттерна в языке Java позволяет использовать как передачу, так и запрос состояния.

Отлично... Может, я увижу хороший пример использования запроса данных и изменю свое мнение.

Наблюдатель

Почему бы вам не предоставить открытые get-методы, которые позволят нам получить необходимые данные состояния?

Существует много разных наблюдателей; невозможно заранее предвидеть все, что нам может понадобиться. Если нам нужна только часть данных, не обязательно загружать все состояние. Кроме того, упрощается внесение изменений в будущем. Если в программе вводятся новые данные состояния, не нужно изменять вызовы update() на каждом наблюдателе — просто включите в свою реализацию get-методы для чтения дополнительного состояния.

Вот как? Мы обязательно рассмотрим эту возможность...

Неужели мы хоть в чем-то согласились? Что ж, надежда умирает последней.

Возьми в руку карандаш

Вам только что звонили из руководства Weather-O-Rama — в приложение необходимо добавить новый визуальный элемент для отображения теплового индекса. Подробности:

Тепловым индексом называется показатель эффективной (то есть субъективно воспринимаемой) температуры, который вычисляется по значениям температуры T и относительной влажности RH. Формула вычисления теплового индекса выглядит так :

$$heatindex = 16.923 + 1.85212 * 10^{-1} * T + 5.37941 * RH - 1.00254 * 10^{-1} * T * RH + 9.41695 * 10^{-3} * T^2 + 7.28898 * 10^{-3} * RH^2 + 3.45372 * 10^{-4} * T^2 * RH - 8.14971 * 10^{-4} * T * RH^2 + 1.02102 * 10^{-5} * T^2 * RH^2 - 3.8646 * 10^{-5} * T^3 + 2.91583 * 10^{-5} * RH^3 + 1.42721 * 10^{-6} * T^3 * RH + 1.97483 * 10^{-7} * T * RH^3 - 2.18429 * 10^{-8} * T^3 * RH^2 + 8.43296 * 10^{-10} * T^2 * RH^3 - 4.81975 * 10^{-11} * T^3 * RH^3$$

Начинайте вводить!

Шутка. Не беспокойтесь, вам не придется вводить эту формулу; создайте файл HeatIndexDisplay.java и скопируйте в него формулу из файла heatindex.txt.

Файл heatindex.txt можно загрузить на сайте wickedlysmart.com

Как работает эта формула? Понятия не имеем. Попробуйте спросить кого-нибудь в Национальной метеорологической службе (или воспользуйтесь поиском Google).

В новой версии результат будет выглядеть так:

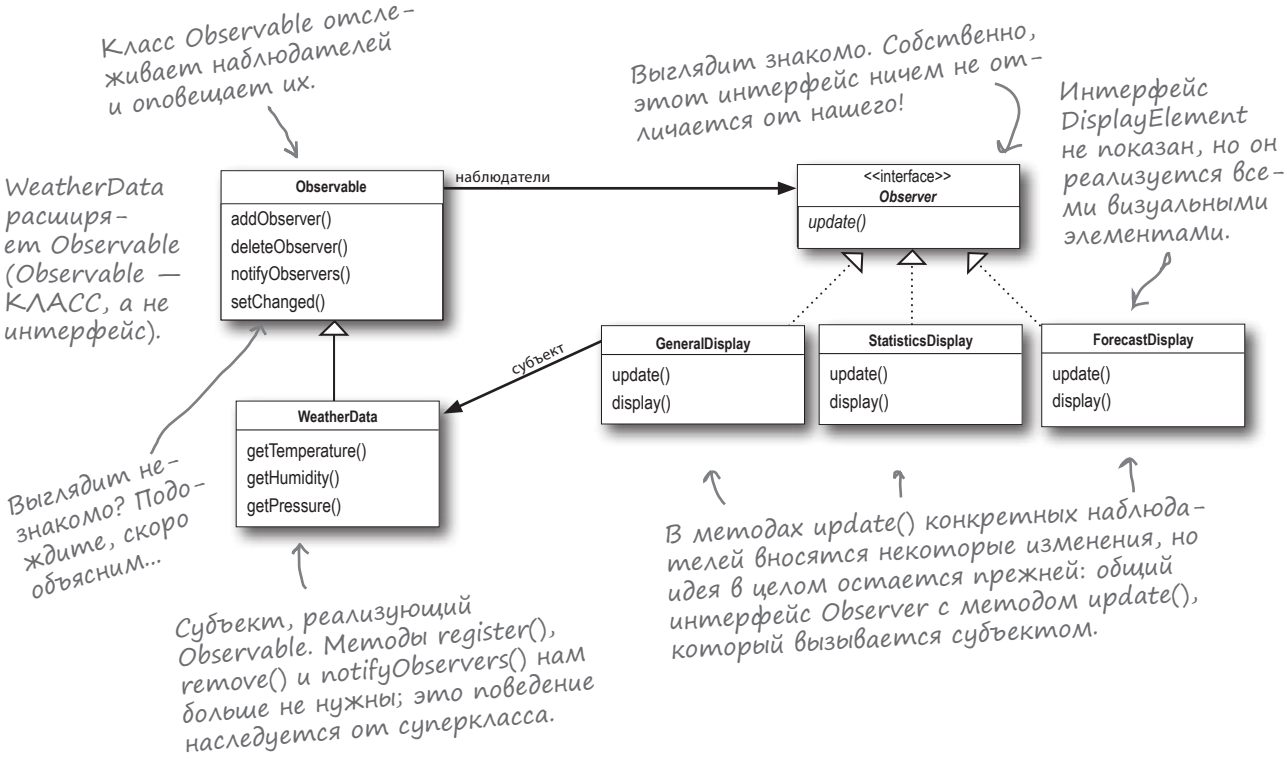
Изменения

```
File  Edit  Window  Help  OverdaRainbow
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Heat index is 82.95535
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Heat index is 86.90124
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
Heat index is 83.64967
%
```

Встроенная реализация в языке Java

До настоящего момента мы реализовали паттерн Наблюдатель самостоятельно, но в некоторых API Java имеется встроенная поддержка. Самые общие средства — интерфейс Observer и класс Observable из пакета java.util — имеют много общего с нашими интерфейсами Subject и Observer, но предоставляют значительное количество готовых функций. Как будет показано ниже, вы также можете реализовать нужный способ обновления наблюдателей (запрос или активная доставка).

Следующая переработанная версия WeatherStation дает представление об использовании java.util.Observer и java.util.Observable:



Как встроенная поддержка паттерна работает в Java

Встроенная поддержка паттерна несколько отличается от реализации, которую мы использовали ранее. Самое очевидное различие заключается в том, что WeatherData (субъект) теперь расширяет класс Observable и наследует методы add, delete и notify (среди прочих). Правила использования Java-версии:

Чтобы объект стал наблюдателем...

Как обычно, реализуйте интерфейс Observer (на этот раз интерфейс java.util.Observer) и вызовите addObserver() для любого объекта Observable. Чтобы исключить себя из списка наблюдателей, вызовите deleteObserver().

Чтобы объект Observable рассылал оповещения...

Прежде всего необходимо наделить его функциями субъекта, расширив суперкласс java.util.Observable. Далее процесс состоит из двух шагов:

1. **Вызовите метод setChanged(), чтобы обозначить изменение состояния в вашем объекте.**

2. **Вызовите один из двух методов notifyObservers():**

either `notifyObservers()` **или** `notifyObservers(Object arg)`

Версия получает произвольный объект данных, который передается каждому наблюдателю при оповещении.

Чтобы наблюдатель рассылал оповещения...

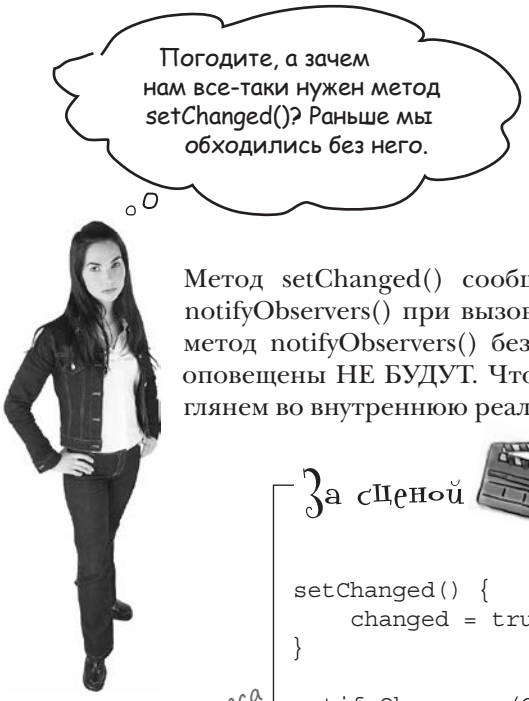
Он должен, как и прежде, реализовать метод update, но сигнатура этого метода несколько изменяется:


```
update(Observable o, Object arg)
```

В аргументе передается субъект, отправивший оповещение.

Объект данных, переданный notifyObservers() или null, если объект данных не был указан.

Если данные должны «доставляться» наблюдателям, передайте их в объекте данных при вызове notifyObserver(arg). Если нет, то наблюдатель должен «запросить» нужные данные от переданного ему объекта Observable. Как? Чтобы это понять, давайте переработаем приложение Weather Station.



Метод setChanged() сообщает об изменении состояния и о том, что метод notifyObservers() при вызове должен обновить своих наблюдателей. Если вызвать метод notifyObservers() без предварительного вызова setChanged(), наблюдатели оповещены НЕ БУДУТ. Чтобы понять, как работает механизм оповещения, мы заглянем во внутреннюю реализацию Observable.

За сценой

Псевдокод класса Observable

```
setChanged() {
    changed = true
}

notifyObservers(Object arg) {
    if (changed) {
        for every observer on the list {
            call update (this, arg)
        }
        changed = false
    }
}

notifyObservers() {
    notifyObservers(null)
}
```

Метод setChanged() устанавливает флаг changed.

notifyObservers() оповещает наблюдателей только при установленном флаге changed.

А после оповещения наблюдателей флаг changed снова сбрасывается.

Зачем это нужно? Метод setChanged() предоставляет большую гибкость в обновлении наблюдателей, так как он позволяет оптимизировать процесс оповещения. Например, представьте, что в приложении Weather Station показания температуры постоянно изменяются на десятые доли градуса. Чтобы объект WeatherData не рассылал слишком частые оповещения, разумнее вызвать setChanged() только при изменении температуры более чем на полградуса.

Возможно, эта функциональность используется не так часто, но она доступна на случай необходимости. В любом случае вызов setChanged() необходим для рассылки оповещений. Если вы захотите воспользоваться данной возможностью, вам также пригодится метод clearChanged(), сбрасывающий флаг changed, и метод hasChanged(), который сообщает текущее состояние флага changed.

Использование встроенной поддержки в приложении Weather Station

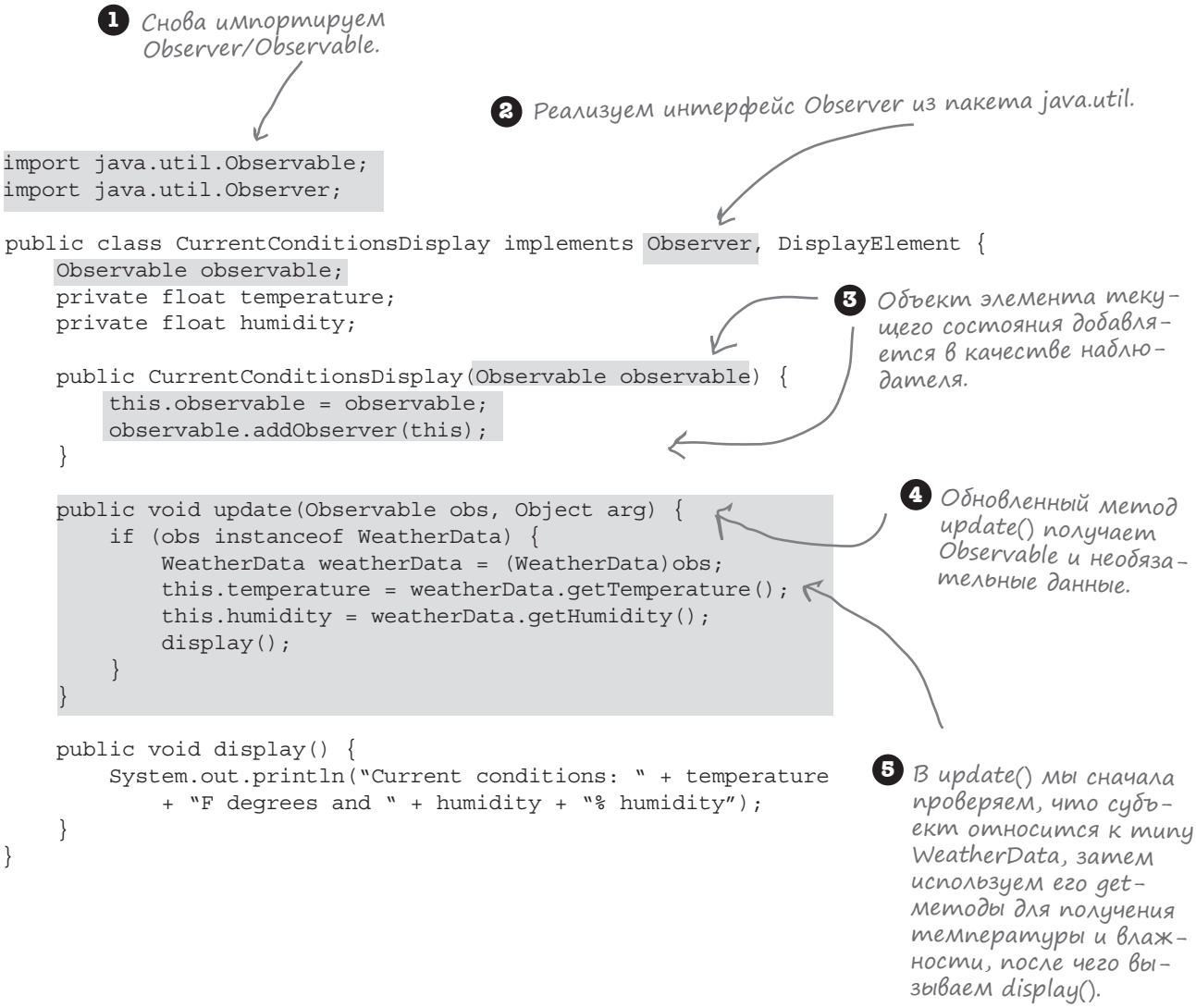
Переработаем объект WeatherData, чтобы он использовал java.util.Observable

1. Импортируем правильные Observer/Observable.
2. Субклассируем Observable.
3. Суперкласс берет на себя все управление наблюдателями, поэтому мы удаляем код регистрации, добавления и оповещения.
4. Теперь структура для хранения наблюдателей не нужна.
5. Перед вызовом notifyObservers() необходимо вызвать setChanged().
6. Эти методы будут использоваться наблюдателями для получения состояния объекта WeatherData.

* Объект данных не передается — это означает, что мы используем модель ЗАПРОСА ДАННЫХ.

```
import java.util.Observable;
import java.util.Observer;

public class WeatherData extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() { }

    public void measurementsChanged() {
        setChanged();
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float
pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

    public float getPressure() {
        return pressure;
    }
}
```

Давайте переработаем код CurrentConditionsDisplay




```
import java.util.Observable;
import java.util.Observer;

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    Observable observable;
    private float temperature;
    private float humidity;

    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }

    public void update(Observable obs, Object arg) {
        if (obs instanceof WeatherData) {
            WeatherData weatherData = (WeatherData)obs;
            this.temperature = weatherData.getTemperature();
            this.humidity = weatherData.getHumidity();
            display();
        }
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

Магниты с кодами

Код класса ForecastDisplay полностью перепутан. Сможете ли вы расставить фрагменты в правильном порядке? Некоторые фигурные скобки упали на пол. Они слишком малы, чтобы их подбирать, — добавьте столько скобок, сколько считаете нужным!

```
public ForecastDisplay(Observable
observable) {
```

```
display();
```

```
observable.addObserver(this);
```

```
if (observable instanceof WeatherData) {
```

```
public class ForecastDisplay implements
Observer, DisplayElement {
```

```
public void display() {
    // display code here
}
```

```
lastPressure = currentPressure;
currentPressure = weatherData.getPressure();
```

```
private float currentPressure = 29.92f;
private float lastPressure;
```

```
WeatherData weatherData =
(WeatherData)observable;
```

```
public void update(Observable observable,
    Object arg) {
```

```
import java.util.Observable;
import java.util.Observer;
```

Выполнение нового кода

Запускаем новый код — на всякий случай...

```
File Edit Window Help TryTihisAtHome
%java WeatherStation
Forecast: Improving weather on the way!
Avg/Max/Min temperature = 80.0/80.0/80.0
Current conditions: 80.0F degrees and 65.0% humidity
Forecast: Watch out for cooler, rainy weather
Avg/Max/Min temperature = 81.0/82.0/80.0
Current conditions: 82.0F degrees and 70.0% humidity
Forecast: More of the same
Avg/Max/Min temperature = 80.0/82.0/78.0
Current conditions: 78.0F degrees and 90.0% humidity
%
```

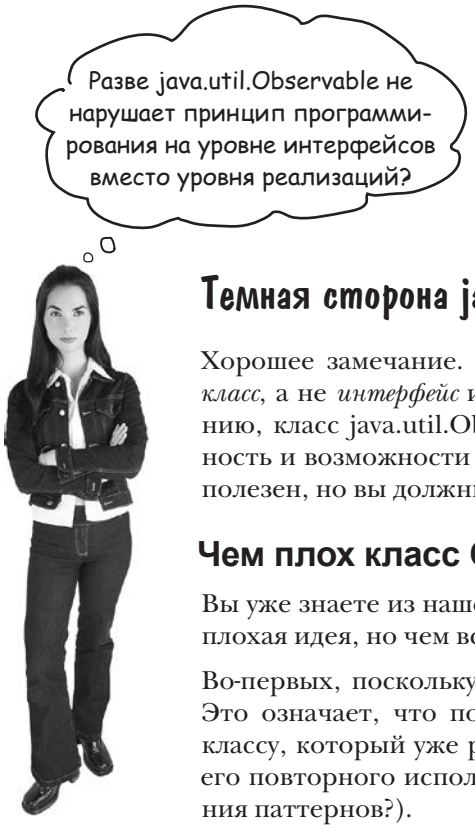
Вам не кажется, что результаты изменились?

Программа выполняет те же вычисления, но порядок выводимого текста загадочным образом изменился. Почему это могло произойти? Подумайте минутку, прежде чем читать дальше...

Никогда не полагайтесь на определенный порядок оповещения наблюдателей

Метод notifyObservers() класса java.util.Observable реализован таким образом, что порядок оповещения наблюдателей *отличается* от порядка, использованного в нашей реализации. Какой порядок считать правильным? Любой; мы просто выбрали разные способы реализации.

Неправильным было бы другое: *зависимость* кода от конкретного порядка оповещения. Почему? Если вдруг потребуется изменить реализации Observable/Observer, порядок оповещения может измениться, и приложение будет выдавать неправильные результаты. Такую архитектуру определенно *нельзя* назвать слабосвязанной.



Темная сторона java.util.Observable

Хорошее замечание. Как уже упоминалось ранее, Observable представляет собой *класс*, а не *интерфейс* и — что еще хуже — даже не *реализацию* интерфейса. К сожалению, класс java.util.Observable обладает рядом недостатков, снижающих его полезность и возможности повторного использования. Это не значит, что он совсем бесполезен, но вы должны быть осторожны.

Чем плох класс Observable

Вы уже знаете из нашего принципа, что программирование на уровне реализации — плохая идея, но чем все-таки плох класс?

Во-первых, поскольку Observable является *классом*, его необходимо *субклассировать*. Это означает, что поведение Observable невозможно добавить к существующему классу, который уже расширяет другой суперкласс. Это ограничивает возможность его повторного использования (а разве оно не является главной целью использования паттернов?).

Во-вторых, при отсутствии интерфейса Observable вы даже не сможете создать собственную реализацию, которая хорошо сочетается со встроенным API Java. Также не удастся заменить реализацию java.util другой (скажем, многопоточной).

Защищенные методы Observable

В API класса Observable метод setChanged() объявлен защищенным. И что? А то, что вы не сможете вызвать setChanged() без субклассирования Observable. Следовательно, вы даже не сможете создать экземпляр класса Observable и включить его в свои объекты посредством композиции — *только* субклассирование. Так нарушается второй принцип проектирования — *отдавать предпочтение композиции перед наследованием*.

Что делать?

Класс Observable *может* подойти для вашей задачи, если вы можете расширить java.util.Observable. В остальных случаях приходится создавать собственную реализацию, как было сделано в начале главы. В любом случае вы уже достаточно хорошо понимаете паттерн Наблюдатель и сможете работать с любым API, в котором этот паттерн используется.

Другие примеры использования паттерна Наблюдатель в JDK

Реализация Observer/Observable в java.util — не единственный пример поддержки паттерна Наблюдатель в JDK; JavaBeans и Swing тоже предоставляют собственные реализации паттерна. К этому моменту вы достаточно хорошо разбираетесь в теме, чтобы проанализировать эти API самостоятельно; но мы рассмотрим короткий простой пример с использованием Swing просто для интереса.

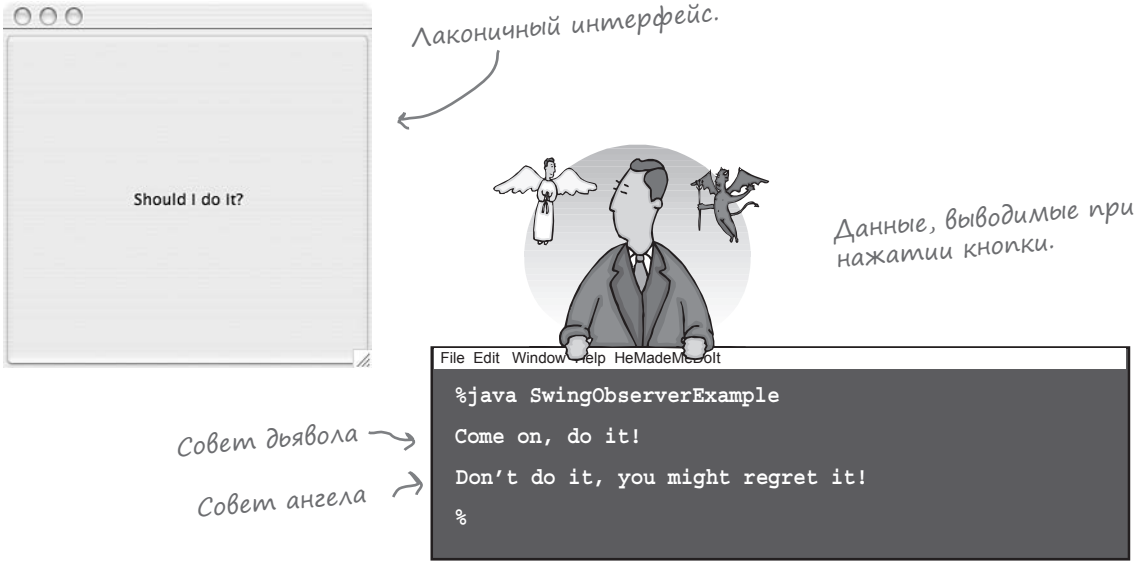
Если вас интересует поддержка паттерна Наблюдатель в JavaBeans, присмотритесь к интерфейсу PropertyChangeListener.

Небольшая справка...

Рассмотрим простую часть Swing API — JButton. Заглянув во внутреннюю реализацию суперкласса JButton, AbstractButton, вы найдете в ней многочисленные методы добавления/удаления наблюдателей (слушателей в терминологии Swing). Эти методы позволяют добавлять и удалять слушателей и прослушивать различные типы событий, происходящих с компонентом Swing. Например, ActionListener позволяет прослушивать различные типы действий, выполняемых с кнопками (например, нажатий). Разные типы слушателей часто встречаются в Swing API.

Судьбоносное решение

Приложение довольно простое: при нажатии кнопки с вопросом слушатели (наблюдатели) могут ответить на вопрос так, как считают нужным. Мы реализовали двух таких наблюдателей: AngelListener и DevilListener. Вот как работает приложение:



Наконец, программный код...

Объем необходимого кода совсем невелик. Все, что от вас потребуется — создать объект JButton, разместить его на компоненте JFrame и настроить слушателей. Для реализации слушателей будут использоваться внутренние классы (стандартный прием в Swing-программировании).

```
public class SwingObserverExample {
    JFrame frame;

    public static void main(String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }
    public void go() {
        frame = new JFrame();

        JButton button = new JButton("Should I do it?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());

        // Set frame properties here
    }


    class AngelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Don't do it, you might regret it!");
        }
    }

    class DevilListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Come on, do it!");
        }
    }
}
```

Простое приложение Swing создает панель и размещает на ней кнопку.

Назначает объекты слушателями (наблюдателями) событий кнопки.

Здесь размещается код подготовки фрейма.

Определения наблюдателей в виде внутренних классов (хотя возможны и другие способы).

При изменении состояния субъекта (в данном случае кнопки) вместо update() вызывается метод actionPerformed().

Я использовал лямбда-выражения вместо простых слушателей действий в своем коде Swing. При этом я по-прежнему использую паттерн «Наблюдатель»?

Поддержка лямбда-выражений была добавлена в Java 8. Если вы еще не знакомы с лямбда-выражениями, не беспокойтесь; вы можете продолжить использование внутренних классов для своих наблюдателей Swing.

Да, при использовании лямбда-выражений вместо внутреннего класса вы просто пропускаете этап создания объекта ActionListener. С лямбда-выражением вы вместо этого создаете объект функции, который и является наблюдателем. При передаче этого объекта функции addActionListener() Java проверяет его сигнатуру на соответствие actionPerformed() – единственному методу интерфейса ActionListener.

Когда пользователь щелкнет на кнопке, объект кнопки оповещает об этом своих наблюдателей, включая объекты функций, созданные лямбда-выражениями, и вызывает метод actionPerformed() каждого слушателя.

А теперь посмотрим, как использовать лямбда-выражения в качестве наблюдателей для упрощения приведенного кода:

Обновленный код с использованием лямбда-выражений:

```
public class SwingObserverExample {
    JFrame frame;
    public static void main(String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }
    public void go() {
        frame = new JFrame();
        JButton button = new JButton("Should I do it?");
        button.addActionListener(event ->
            System.out.println("Don't do it, you might regret it!"));
        button.addActionListener(event ->
            System.out.println("Come on, do it!"));

        // Set frame properties here
    }
}
```

Мы заменили объекты AngelListener и DevilListener лямбда-выражениями, которые реализуют ту же функциональность, что и прежде.

Когда пользователь щелкает на кнопке, объекты функций, созданные лямбда-выражениями, получают оповещения об этом событии, после чего выполняется реализуемый ими метод.

Лямбда-выражения делают этот код намного более компактным.

Два класса ActionListener (DevilListener и AngelListener) полностью исключены.

За дополнительной информацией о лямбда-выражениях обращайтесь к документации Java и главе 6.

Новые инструменты

Подошла к концу глава 2. В ваш ОО-инструментарий добавились новые инструменты...

Концепции

Абстракция
...ляция
...физм
...ание

Принципы

Инкапсулируйте то, что изменяется.

Предпочитайте композицию наследованию.

Программируйте на уровне интерфейсов.

Стремитесь к слабой связанности взаимодействующих объектов.

Новый принцип. Помните: слабосвязанные структуры более гибки и лучше выдерживают изменения.

Паттерны

Стратегия — ...определяет семейство алгоритмов, инкапсулирует их и делает взаимозаменяемыми. Позволяет алгоритму изменяться независимо от клиентов, которые его используют.

Наблюдатель — определяет отношение «один-ко-многим» между объектами таким образом, что при изменении состояния одного объекта происходит автоматическое оповещение и обновление всех зависимых объектов

Новый паттерн для слабосвязанного оповещения групп объектов об изменении состояния. Мы еще вернемся к паттерну Наблюдатель, когда речь пойдет о MVC!

КЛЮЧЕВЫЕ МОМЕНТЫ

- Паттерн Наблюдатель определяет отношение «один-ко-многим» между объектами.
- Субъекты обновляют наблюдателей через единый интерфейс.
- Субъект ничего не знает о наблюдателях — кроме того, что они реализуют интерфейс Observer.
- При использовании паттерна возможен как запрос, так и активная доставка данных от субъекта (запрос считается более «правильным»).
- Работа кода не должна зависеть от порядка оповещения наблюдателей.
- Java содержит несколько реализаций паттерна Наблюдатель, включая обобщенную реализацию java.util.Observable.
- Помните о недостатках java.util.Observable.
- Не бойтесь самостоятельно реализовать Observable при необходимости.
- Swing, как и многие GUI-инфраструктуры, широко применяет паттерн Наблюдатель.
- Паттерн также встречается во многих других местах, включая JavaBeans и RMI.

Упражнение

Проверка принципов проектирования

Опишите, как каждый из принципов проектирования используется в паттерне Наблюдатель.

Принцип проектирования

Определите аспекты вашего приложения, которые могут изменяться, и отделите их от тех, которые будут оставаться неизменными.

Принцип проектирования

Программируйте на уровне интерфейсов, а не на уровне реализаций.

Принцип проектирования

Отдавайте предпочтение композиции перед наследованием.

Сложная задача. Подсказка: подумайте, как наблюдатели взаимодействуют с субъектами.

Возьми в руку карандаш
Решение

Какие из следующих утверждений относятся к первой реализации? (Укажите все варианты.)

- ☑ A. Мы программируем на уровне реализаций, а не интерфейсов.
- ☑ B. Для каждого нового элемента придется изменять код.
- ☑ C. Элементы не могут добавляться (или удаляться) во время выполнения.
- ☐ D. Элементы не реализуют единый интерфейс.
- ☑ E. Переменные аспекты архитектуры не инкапсулируются.
- ☐ F. Нарушается инкапсуляция класса WeatherData.

Проверка принципов проектирования. Решение

Принцип проектирования

Определите аспекты вашего приложения, которые могут изменяться, и отделите их от тех, которые будут оставаться неизменными.

Переменные аспекты — состояние субъекта, количество и тип наблюдателей. Паттерн позволяет изменять объекты, зависящие от состояния субъекта, без изменения самого субъекта.

Принцип проектирования

Программируйте на уровне интерфейсов, а не на уровне реализаций.

И субъект, и наблюдатели используют интерфейсы. Субъект отслеживает объекты, реализующие интерфейс Observer, а наблюдатели регистрируются и оповещаются через интерфейс Subject.

Принцип проектирования

Отдавайте предпочтение композиции перед наследованием.

Отношения наблюдателей с субъектом не определяются иерархией наследования, а задаются во время выполнения посредством композиции!

Магниты с кодами
Решение

```
import java.util.Observable;
import java.util.Observer;

public class ForecastDisplay implements
Observer, DisplayElement {

    private float currentPressure = 29.92f;
    private float lastPressure;

    public ForecastDisplay(Observable
    observable) {

        observable.addObserver(this);
    }

    public void update(Observable observable,
        Object arg) {

        if (observable instanceof WeatherData) {

            WeatherData weatherData =
            (WeatherData)observable;

            lastPressure = currentPressure;
            currentPressure = weatherData.getPressure();

            display();
        }
    }

    public void display() {
        // display code here
    }
}
```

Украшение объектов

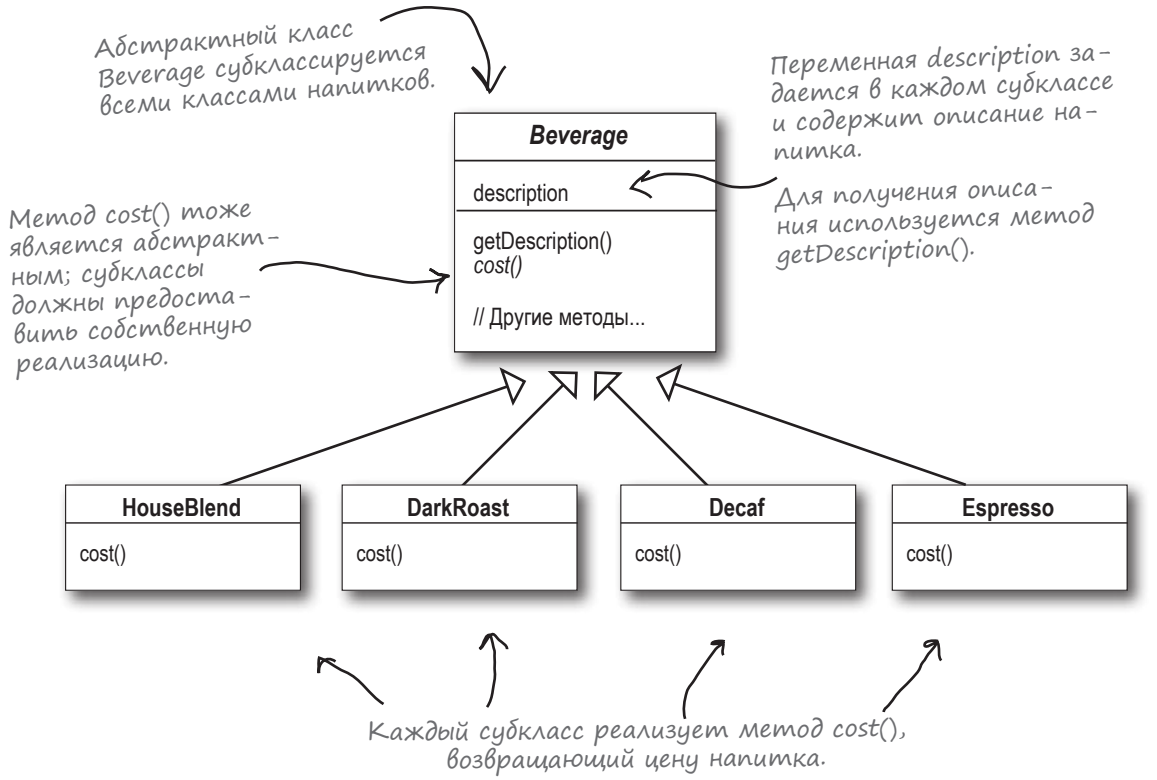
Эту главу можно назвать «Взгляд на архитектуру для любителей наследования». Мы проанализируем типичные злоупотребления из области наследования, и вы научитесь декорировать свои классы во время выполнения с использованием разновидности композиции. Зачем? Затем, что этот прием позволяет вам наделить свои (или чужие) объекты новыми возможностями *без модификации кода классов*.

Добро пожаловать в Starbuzz!

Сеть кофеен Starbuzz стремительно развивается. Если вы увидите одну из этих кофеен на углу, посмотрите через дорогу — и вы наверняка увидите другую.

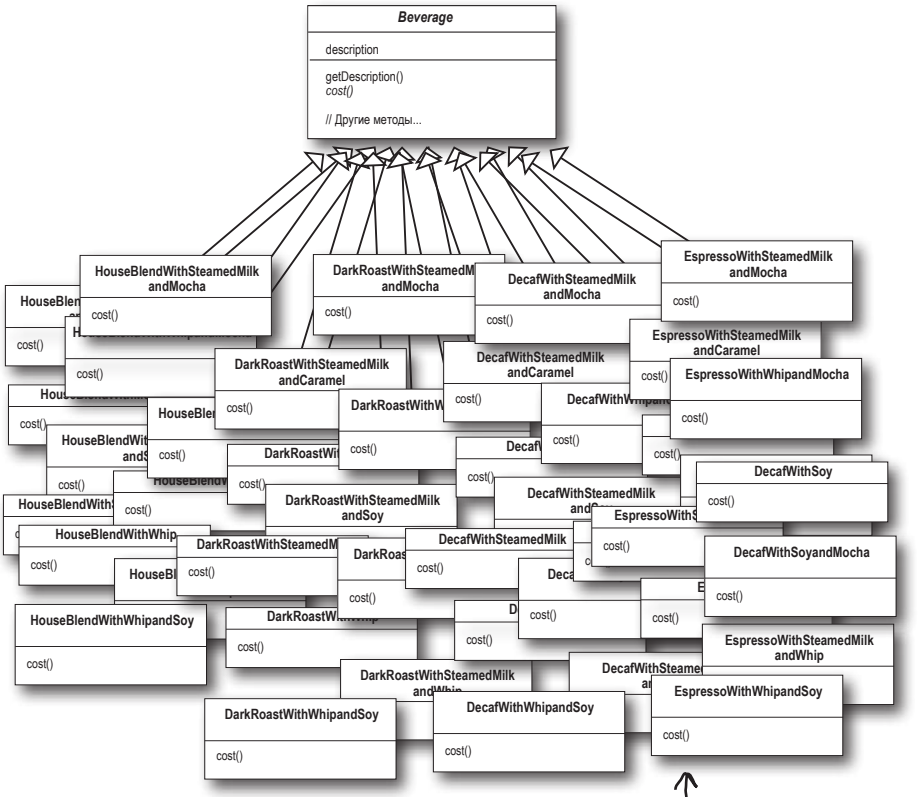
Из-за бурного роста руководству Starbuzz никак не удается привести свою систему заказов в соответствие с реальным ассортиментом.

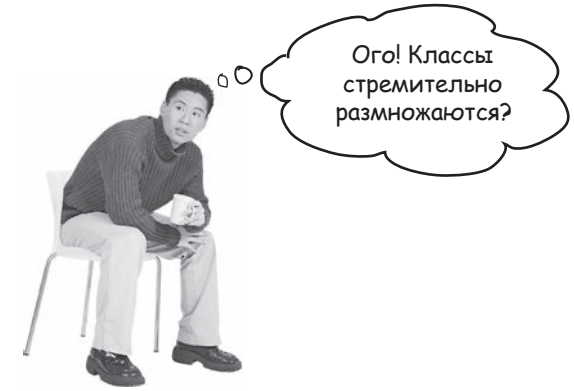
Когда бизнес только начинался, иерархия классов выглядела примерно так...



К кофе можно заказать различные дополнения (пенка, шоколад и т. д.), да еще украсить все сверху взбитыми сливками. Дополнения не бесплатны, поэтому они должны быть встроены в систему оформления заказов.

Первая попытка...





Каждый метод cost вычисляет стоимость кофе вместе со всеми дополнениями.

МОЗГОВОЙ ШТУРМ

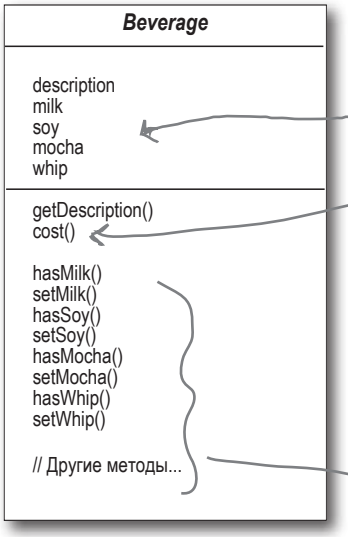
Понятно, что сопровождение классов Starbuzz станет сущим кошмаром. А если молоко подорожает? И что произойдет, если в меню появится новая карамельная добавка?

Даже если отвлечься от проблем с сопровождением — какие принципы проектирования нарушает такой подход?

Подсказка: нарушены два принципа, и причем серьезно!

Глупо; зачем нужны все эти классы? Разве для отслеживания дополнений нельзя использовать переменные экземпляров в суперклассе и наследование?

Давайте попробуем. Начнем с базового класса Beverage и добавим переменные, которые указывают, присутствует ли в кофе то или иное дополнение...

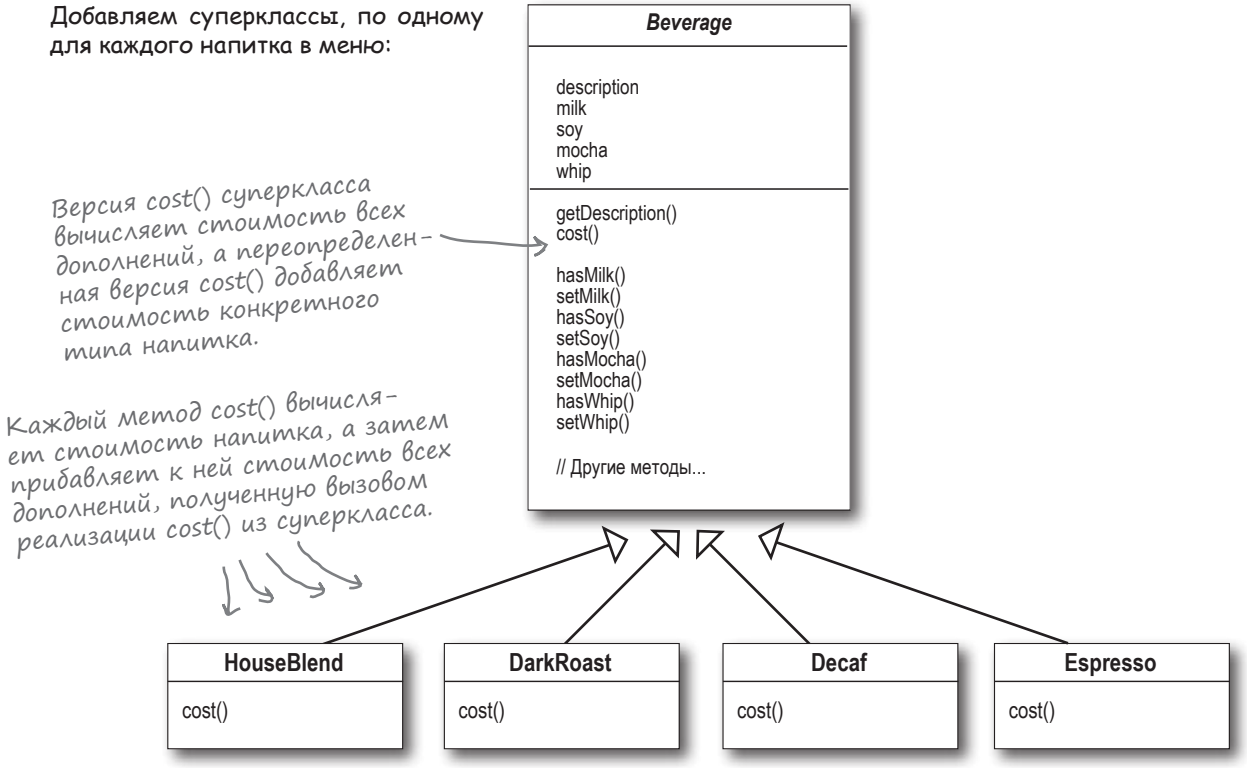


Логическая переменная для каждого дополнения.

Теперь мы реализуем cost() в Beverage, чтобы метод вычислял суммарную стоимость напитка вместе со всеми дополнениями. Субклассы по-прежнему переопределяют cost(), но они также вызывают версию суперкласса для вычисления общей стоимости базового напитка со всеми дополнениями.

Эти методы читают и устанавливают флаги дополнений.

Добавляем суперклассы, по одному для каждого напитка в меню:



Возьми в руку карандаш

Напишите реализации cost( ) для следующих классов (достаточно псевдокода Java):

```
public class Beverage {
    public double cost() {




    }
}
```

```
public class DarkRoast extends Beverage {

    public DarkRoast() {
        description = "Most Excellent Dark
Roast";
    }

    public double cost() {




    }
}
```

Возьми в руку карандаш

Какие изменения требований или других факторов могут отразиться на работоспособности этой архитектуры?

Изменение цены дополнений потребует модификации существующего кода.

При появлении новых дополнений нам придется добавлять новые методы и изменять реализацию cost в суперклассе.

Для некоторых новых напитков (холодный чай?) дополнения могут оказаться неуместными, но субкласс Tea все равно будет наследовать hasWhip() и другие методы.

Как было показано в главе 1, это крайне нежелательно.

А если клиент захочет двойную порцию шоколада?

Ваша очередь:

Учитель и Ученик...

Учитель: *С момента нашей встречи прошло много времени. Ты усердно медитировал на наследовании?*

Ученик: *Да, учитель. Наследование обладает большими возможностями, но я осознал, что оно не всегда приводит к самой гибкой или удобной в сопровождении архитектуре.*

Учитель: *Да, ты кое-чего достиг. Тогда скажи мне, можно ли обеспечить повторное использование кода без наследования?*

Ученик: *Учитель, я узнал о механизмах «наследования» поведения посредством композиции и делегирования.*

Учитель: *Продолжай...*

Ученик: *Поведение, унаследованное посредством субклассирования, задается статически на стадии компиляции. Кроме того, оно должно наследоваться всеми субклассами. Расширение поведения объекта посредством композиции может осуществляться динамически.*

Учитель: *Очень хорошо, ты начинаешь видеть силу композиции.*

Ученик: *Да, я могу наделить объект новыми возможностями — даже теми, которые не были предусмотрены при проектировании суперкласса. И мне не придется изменять его код!*

Учитель: *Что ты узнал о влиянии композиции на простоту сопровождения твоего кода?*

Ученик: *Динамическая композиция объектов позволяет добавлять новую функциональность посредством написания нового кода (вместо изменения существующего). Так как мы не изменяем готовый код, риск введения ошибок или непредвиденных побочных эффектов значительно снижается.*

Учитель: *Очень хорошо. Иди и медитируй дальше... Помни: код должен быть закрытым (для изменений), словно цветок лотоса на закате, но при этом открытым (для расширения), словно цветок лотоса на утренней заре.*

Принцип открытости/закрытости

Мы подошли к одному из важнейших принципов проектирования:

Принцип проектирования

Классы должны быть открыты для расширения, но закрыты для изменения.

Заходите, мы *открыты*. Не стесняйтесь, расширяйте наши классы любым нужным поведением. Если ваши потребности изменятся (а это наверняка произойдет), просто создайте собственное расширение.

Извините, мы *закрыты*. Мы потратили много времени на проверку и отладку этого кода и не можем позволить вам изменять его. Код должен остаться закрытым для изменения. Если вас это не устраивает — обратитесь к директору.

Наша цель заключается в том, чтобы классы можно было легко расширять новым поведением без изменения существующего кода. Что это нам дает? Архитектуры, устойчивые к изменениям и достаточно гибкие для поддержки новой функциональности в соответствии с изменившимися требованиями.

Часто Задаваемые Вопросы

В: Открыты для расширений и закрыты для изменения? Звучит весьма противоречиво. Разве такое возможно?

О: Хороший вопрос. Ведь чем труднее изменить что-либо, тем меньше возможностей для расширения, верно?

Однако некоторые ОО-приемы позволяют расширять системы даже в том случае, если вы не можете изменить их базовый код. Вспомните главу 2: добавляя новых наблюдателей, мы можем в любой момент расширить субъекта без добавления в него нового кода. Вы еще увидите немало других способов расширения поведения с применением других методов ОО-проектирования.

В: Хорошо, я могу понять расширение субъектов, но как спроектировать архитектуру, рассчитанную на расширение, но закрытую для изменения?

О: Многие паттерны представляют собой надежные, проверенные временем механизмы расширения поведения без изменения кода.

В: Но как обеспечить соблюдение принципа открытости/закрытости во всех компонентах моей архитектуры?

О: Обычно это невозможно. Чтобы ОО-архитектура была гибкой и открытой для расширения без изменения существующего кода, обычно приходится затратить немало времени и усилий. Следование принципу открытости/закрытости обычно приводит к введению новых уровней абстракции, усложняющих код. Лучше сосредоточиться только на тех областях вашей архитектуры, которые с наибольшей вероятностью будут изменяться, и применять принципы именно там.

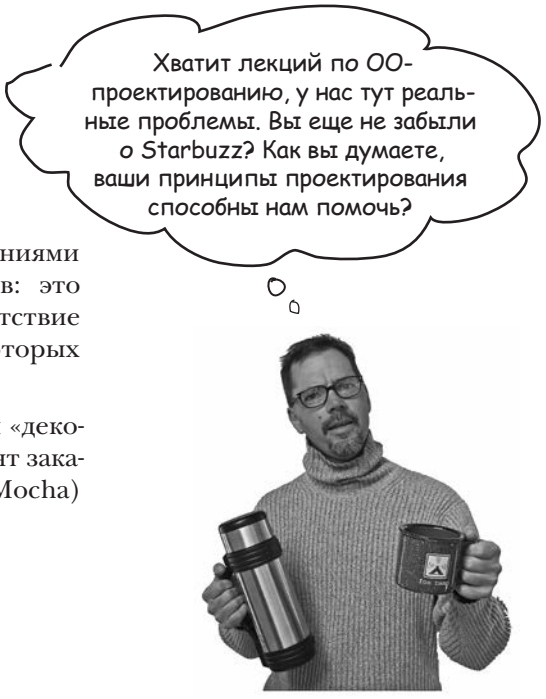
В: Как узнать, какие области изменений более важны?

О: Отчасти это зависит от опыта проектирования ОО-систем и знания предметной области. Знакомство с другими примерами поможет вам выявить переменные области в ваших собственных архитектурах.

На первый взгляд формулировка принципа кажется противоречивой, однако существуют приемы, обеспечивающие возможность расширения кода без его модификации.

Будьте осторожны с выбором расширяемых областей. ПОВСЕМЕСТНОЕ применение принципа открытости/закрытости неэффективно и расточительно, оно приводит к созданию сложного, малопонятного кода.

Знакомство с паттерном Декоратор



Итак, схема вычисления стоимости напитка с дополнениями посредством наследования обладает рядом недостатков: это и разрастание классов, и негибкая архитектура, и присутствие в базовом классе функциональности, неуместной в некоторых субклассах.

Поэтому мы поступим иначе: начнем с базового напитка и «декорируем» его на стадии выполнения. Например, если клиент заказывает кофе темной обжарки (Dark Roast) с шоколадом (Mocha) и взбитыми сливками (Whip), мы

1. **берем объект DarkRoast;**
2. **декорируем его объектом Mocha;**
3. **декорируем его объектом Whip;**
4. **вызываем метод cost() и пользуемся делегированием для прибавления стоимости дополнений.**

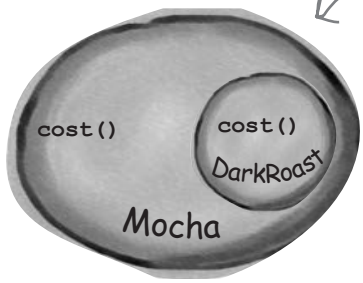
Хорошо, но как «декорировать» объект и как в этой схеме работает делегирование? Давайте разберемся...

Построение заказанного напитка

1 **Начинаем с объекта DarkRoast.**

Напоминаем, что DarkRoast наследует от Beverage и содержит метод cost() для вычисления стоимости напитка.

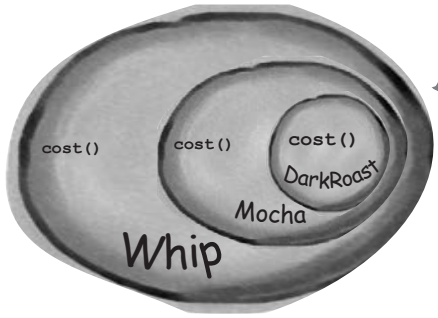
2 **Клиент заказывает шоколад, поэтому мы создаем объект Mocha и «заворачиваем» в него DarkRoast.**



Объект Mocha является декоратором. Его тип повторяет тип декорируемого объекта — в данном случае Beverage.

Объект Mocha тоже содержит метод cost(), а благодаря полиморфизму он может интерпретироваться как Beverage (так как Mocha является субклассом Beverage).

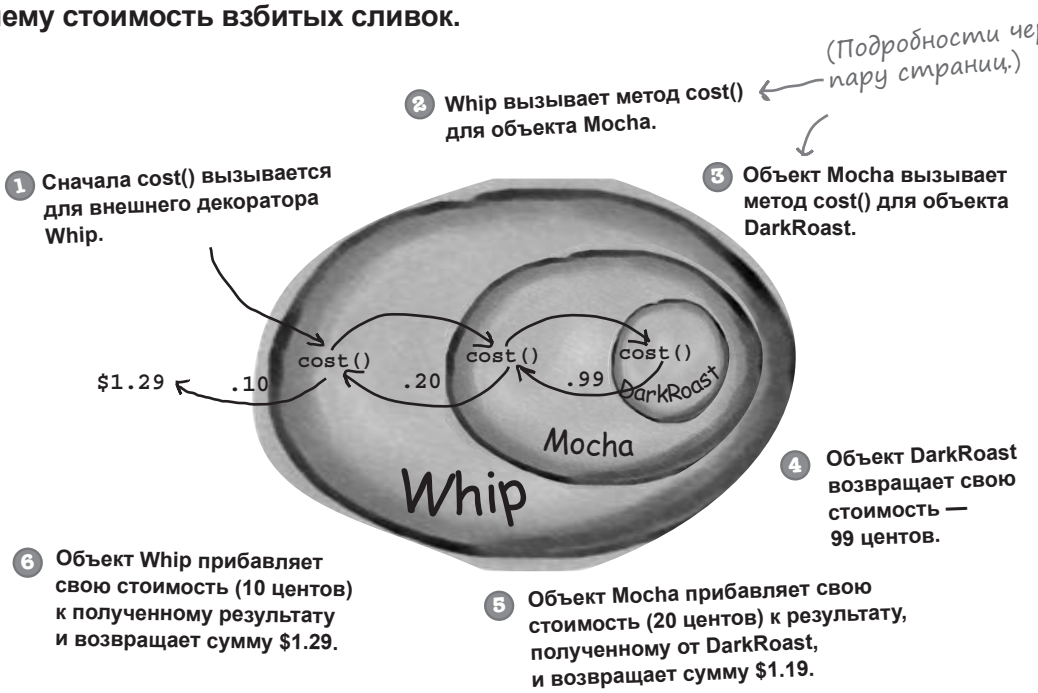
3 **Клиент также хочет взбитые сливки, поэтому мы создаем объект Whip и «заворачиваем» в него Mocha.**



Декоратор Whip также повторяет тип DarkRoast и содержит метод cost().

Таким образом, объект DarkRoast, «завернутый» в Mocha и Whip, сохраняет признаки Beverage, и с ним можно делать все, что можно делать с DarkRoast, включая вызов метода cost().

4 Пришло время вычислить общую стоимость напитка. Для этого мы вызываем метод cost() внешнего декоратора Whip, а последний делегирует вычисление декорируемым объектам. Получив результат, он прибавляет к нему стоимость взбитых сливок.



Что нам уже известно...

- Декораторы имеют тот же супертип, что и декорируемые объекты.
- Объект можно «завернуть» в один или несколько декораторов.
- Так как декоратор относится к тому же супертипу, что и декорируемый объект, мы можем передать декорированный объект вместо исходного.
- Декоратор добавляет свое поведения до и (или) после делегирования операций декорируемому объекту, выполняющему остальную работу.

Очень важно!

- Объект может быть декорирован в любой момент времени, так что мы можем декорировать объекты динамически и с произвольным количеством декораторов.

А теперь давайте посмотрим на паттерн Декоратор в действии и немного попрограммируем.

Определение паттерна Декоратор

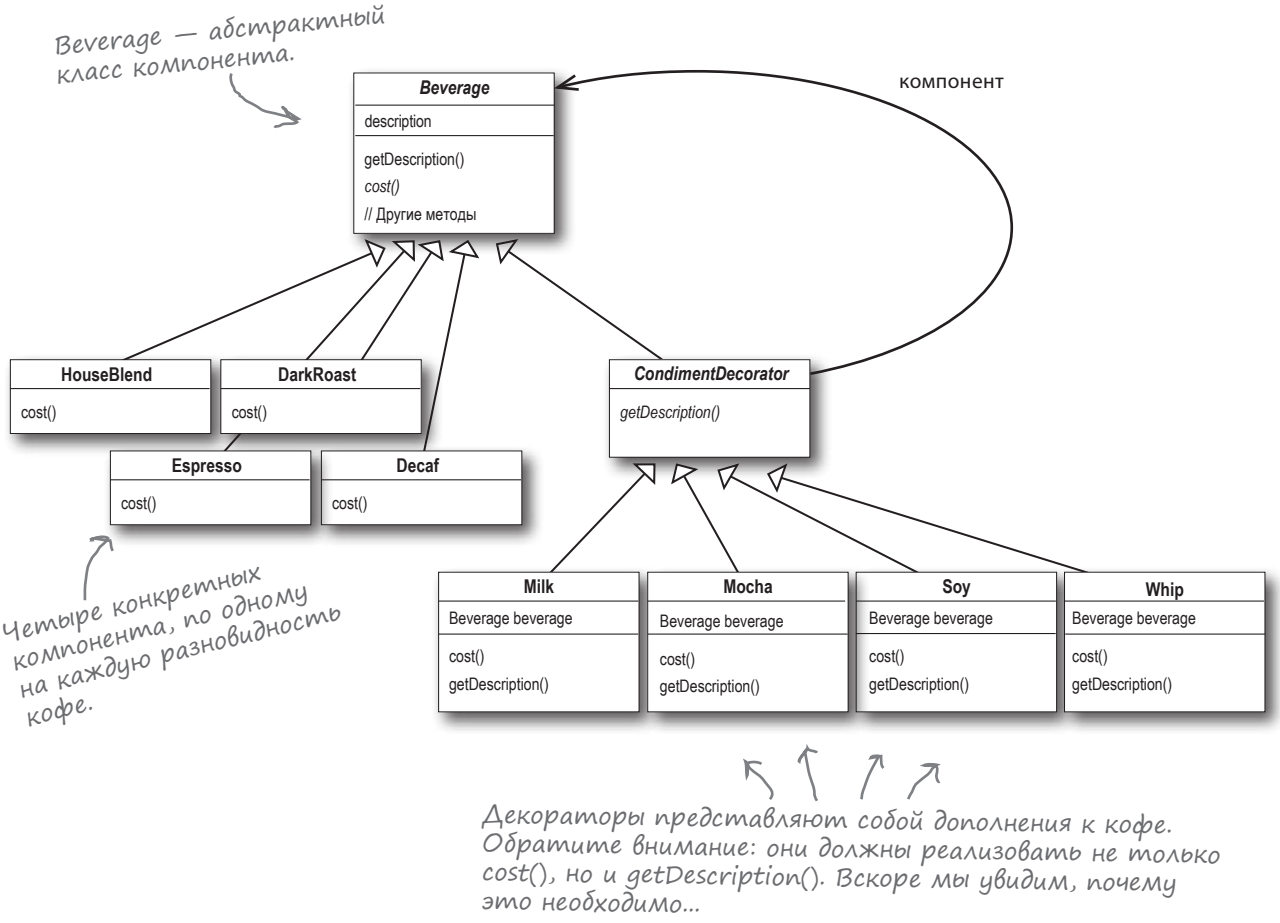
Начнем, как обычно, с описания паттерна Декоратор.

> **Паттерн Декоратор** динамически наделяет объект новыми возможностями и является гибкой альтернативой субклассированию в области расширения функциональности.

Приведенное определение описывает *роль* паттерна Декоратор, но ничего не говорит о том, как *применять* паттерн в наших реализациях. Следующая диаграмма классов выглядит более содержательно (а на следующей странице эта структура применена к классам напитков).

Декораторы и напитки

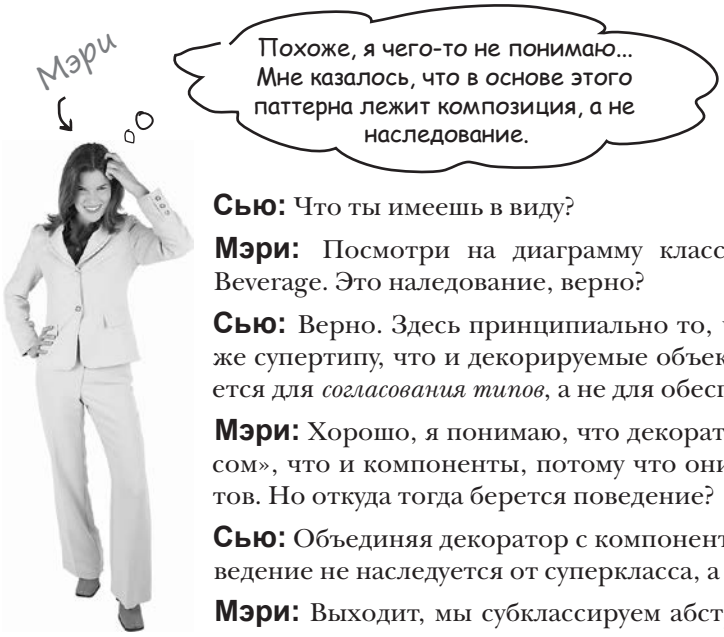
Давайте преобразуем иерархию напитков Starbuzz к этой структуре...



Прежде чем двигаться дальше, подумайте, как бы вы реализовали метод cost() в классах напитков и дополнений. А как бы вы реализовали метод getDescription() для дополнений?

Разговор в офисе

Путаница с наследованием и композицией



Сью: Что ты имеешь в виду?

Мэри: Посмотри на диаграмму классов. CondimentDecorator расширяет класс Beverage. Это наледование, верно?

Сью: Верно. Здесь принципиально то, что декораторы должны относиться к тому же супертипу, что и декорируемые объекты. Таким образом, наследование применяется для *согласования типов*, а не для обеспечения *поведения*.

Мэри: Хорошо, я понимаю, что декораторы должны обладать таким же «интерфейсом», что и компоненты, потому что они должны использоваться вместо компонентов. Но откуда тогда берется поведение?

Сью: Объединяя декоратор с компонентом, мы добавляем новое поведение. Это поведение не наследуется от суперкласса, а добавляется посредством композиции.

Мэри: Выходит, мы субклассируем абстракный класс Beverage только для приведения к нужному типу, а не для наследования его поведения. Поведение формируется в результате композиции декораторов с базовыми компонентами и другими декораторами.

Сью: Точно.

Мэри: Ооо, начинаю понимать. Композиция объектов дает нам значительную гибкость в смешении напитков и дополнений. Очень элегантно.

Сью: Да. Если бы мы воспользовались наследованием, то все поведение определялось бы статически во время компиляции. Другими словами, мы могли бы использовать только то поведение, которое нам предоставляет суперкласс — или которое мы переопределяем в субклассе. Композиция делает возможным произвольное смешивание декораторов... *во время выполнения*.

Мэри: И, насколько я поняла, мы можем в любой момент реализовать новые декораторы для добавления нового поведения.

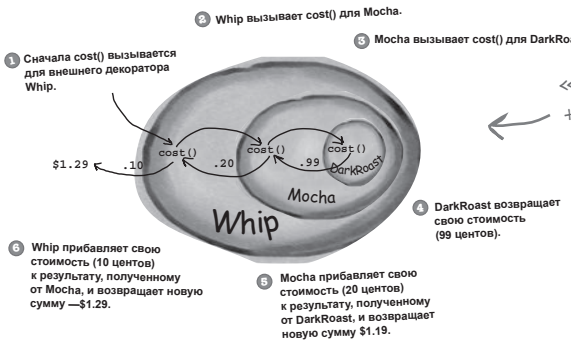
Сью: Вот именно.

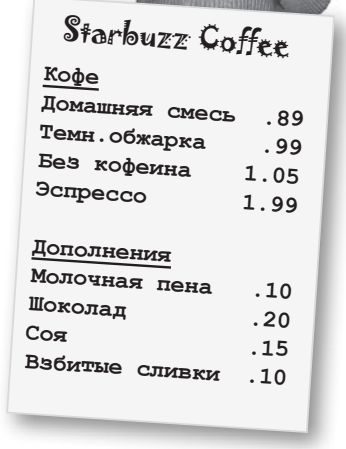
Мэри: Остался последний вопрос. Если наследуется только тип компонента, то почему не воспользоваться интерфейсом вместо абстрактного класса Beverage?

Сью: Потому что когда нам передали этот код, у Starbuzz уже *был* абстрактный класс Beverage. Изменения в существующем коде всегда нежелательны; не стоит «исправлять» код, если можно просто воспользоваться абстрактным классом.

Обучение баристы

Нарисуйте, как будет вычисляться цена заказа «латте с двойным шоколадом, соей и взбитыми сливками». Возьмите цены из меню и нарисуйте свою схему в формате, который мы использовали несколько страниц назад.





Возьми в руку карандаш

Нарисуйте здесь свою схему.

Пишем код для Starbuzz

Пора воплотить наши замыслы в реальном коде.

Начнем с класса Beverage, который достался нам из исходной архитектуры Starbuzz. Он выглядит так:

```
public abstract class Beverage {
    String description = "Unknown Beverage";

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

Beverage — абстрактный класс с двумя методами: getDescription() и cost().

Метод getDescription уже реализован, а метод cost() необходимо реализовать в субклассах.

Как видите, класс Beverage достаточно прост. Давайте реализуем абстрактный класс для дополнений:

Объекты должны быть взаимозаменяемы с Beverage, поэтому расширяем класс Beverage.

```
public abstract class CondimentDecorator extends Beverage {
    public abstract String getDescription();
}
```

Также все декораторы должны заново реализовать метод getDescription(). Зачем? Скоро узнаете...

Программируем классы напитков

Разобравшись с базовыми классами, переходим к реализации некоторых напитков. Начнем с эспрессо. Как говорилось ранее, мы должны задать описание конкретного напитка в методе getDescription() и реализовать метод cost().

Все классы конкретных напитков расширяют Beverage.

```
public class Espresso extends Beverage {

    public Espresso() {
        description = "Espresso";
    }

    public double cost() {
        return 1.99;
    }
}
```

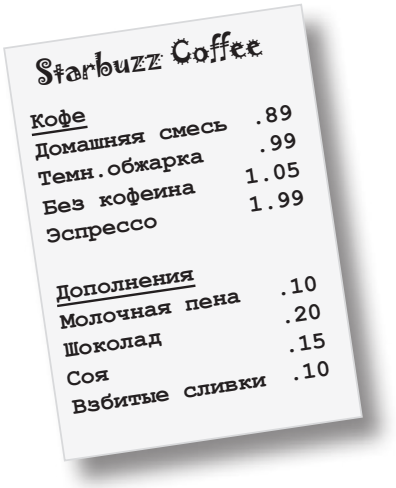
Описание задается в конструкторе класса. Стоит напомнить, что переменная description наследуется от Beverage.

Остается вычислить стоимость напитка. В этом классе беспокоиться о дополнениях не нужно, поэтому мы просто возвращаем стоимость «базового» эспрессо: $1.99.

```
public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = "House Blend Coffee";
    }

    public double cost() {
        return .89;
    }
}
```

Другой класс напитка. От нас требуется лишь назначить подходящее описание и вернуть правильную стоимость.

Два других класса напитков (DarkRoast и Decaf) создаются аналогично.



Программирование дополнений

Взглянув на диаграмму классов паттерна Декоратор, вы увидите, что мы написали абстрактный компонент (Beverage), конкретные компоненты (HouseBlend) и абстрактный декоратор (CondimentDecorator). Пришло время реализации конкретных декораторов. Код декоратора Mocha:

Класс декоратора расширяет CondimentDecorator.

Не забудьте, что CondimentDecorator расширяет Beverage.

```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }

    public double cost() {
        return .20 + beverage.cost();
    }
}
```

Чтобы в объекте Mocha хранилась ссылка на Beverage, нам понадобятся:

(1) Переменная для хранения ссылки.

(2) Способ присваивания переменной ссылки на объект. Мы будем передавать ссылку при вызове конструктора.

Теперь необходимо вычислить стоимость напитка с шоколадом. Сначала вызов делегируется декорируемому объекту, а затем стоимость шоколада прибавляется к результату.

В описании должны содержаться не только название напитка (допустим, «Dark Roast»), но и все дополнения — например, «Dark Roast, Mocha». Таким образом, мы сначала получаем описание, делегируя вызов декорируемому объекту, а затем присоединяем к нему строку «, Mocha».

На следующей странице мы создадим экземпляр напитка и декорируем его дополнениями, но сначала...

Упражнение

Напишите и откомпилируйте код для дополнений Soy и Whip. Он необходим для завершения и тестирования приложения.

Готовим кофе

Поздравляем. Можно устроиться поудобнее, заказать кофе и полюбоваться гибкой архитектурой, построенной на основе паттерна Декоратор.

Тестовый код* для оформления заказов:

```
public class StarbuzzCoffee {

    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()
                + " $" + beverage.cost());

        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription()
                + " $" + beverage2.cost());

        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription()
                + " $" + beverage3.cost());
    }
}
```

Заказываем эспрессо без дополнений, выводим описание и стоимость.

Создаем объект DarkRoast.

«Заворачиваем» в объект Mocha...

...Потом во второй...

... И еще в объект Whip.

Напоследок заказываем «домашнюю смесь» с соей, шоколадом и взбитыми сливками.

* Более элегантный способ создания декорированных объектов будет представлен при описании паттерна Фабрика (а также паттерна Строитель в приложении).

Получаем свой заказ:

```
File  Edit  Window  Help  CloudsInMyCoffee

% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
%
```

Часто Задаваемые Вопросы

В: Как быть с кодом, который проверяет конкретную разновидность компонента (скажем, HouseBlend) и выполняет некоторые действия (допустим, оформляет скидку)? После декорирования объекта такой код перестанет работать.

О: Абсолютно точно. Если код зависит от типа конкретного компонента, декораторы нарушат его работоспособность. Пока код пишется для типа абстрактного компонента, использование декораторов остается прозрачным для кода. Впрочем, если вы начинаете программировать на уровне конкретных компонентов, вам стоит переосмыслить архитектуру приложения и использования в нем декораторов.

В: Нет ли опасности, что клиент получит ссылку, которая не относится к самому внешнему декоратору? Допустим, если имеется компонент DarkRoast с декораторами Mocha, Soy и Whip, кто-нибудь по неосторожности может использовать ссылку на Soy вместо Whip, и тогда внешний декоратор не будет учтен в заказе.

О: В паттерне Декоратор возрастает количество объектов, а следовательно, и опасность проблем, связанных с ошибками программирования. Однако декораторы обычно создаются при использовании других паттернов (таких, как Фабрика или Строитель). Создание конкретных компонентов с декораторами в них хорошо инкапсулировано, а ошибки такого рода маловероятны.

В: Могут ли декораторы располагать информацией о других декораторах в цепочке? Допустим, если я хочу, чтобы метод getDecription() вместо строки «Mocha, Whip, Mocha» выводил строку «Whip, Double Mocha»?

О: Декораторы предназначены для расширения поведения декорируемых объектов. Получение информации с других уровней цепочки декораторов выходит за рамки их традиционного назначения. Впрочем, такие ситуации возможны — допустим, декоратор CondimentPrettyPrint может разбирать итоговое описание и выводить его в нужном виде. Для упрощения реализации приведенного примера метод getDecription() может возвращать объект ArrayList с описаниями.

Возьми в руку карандаш

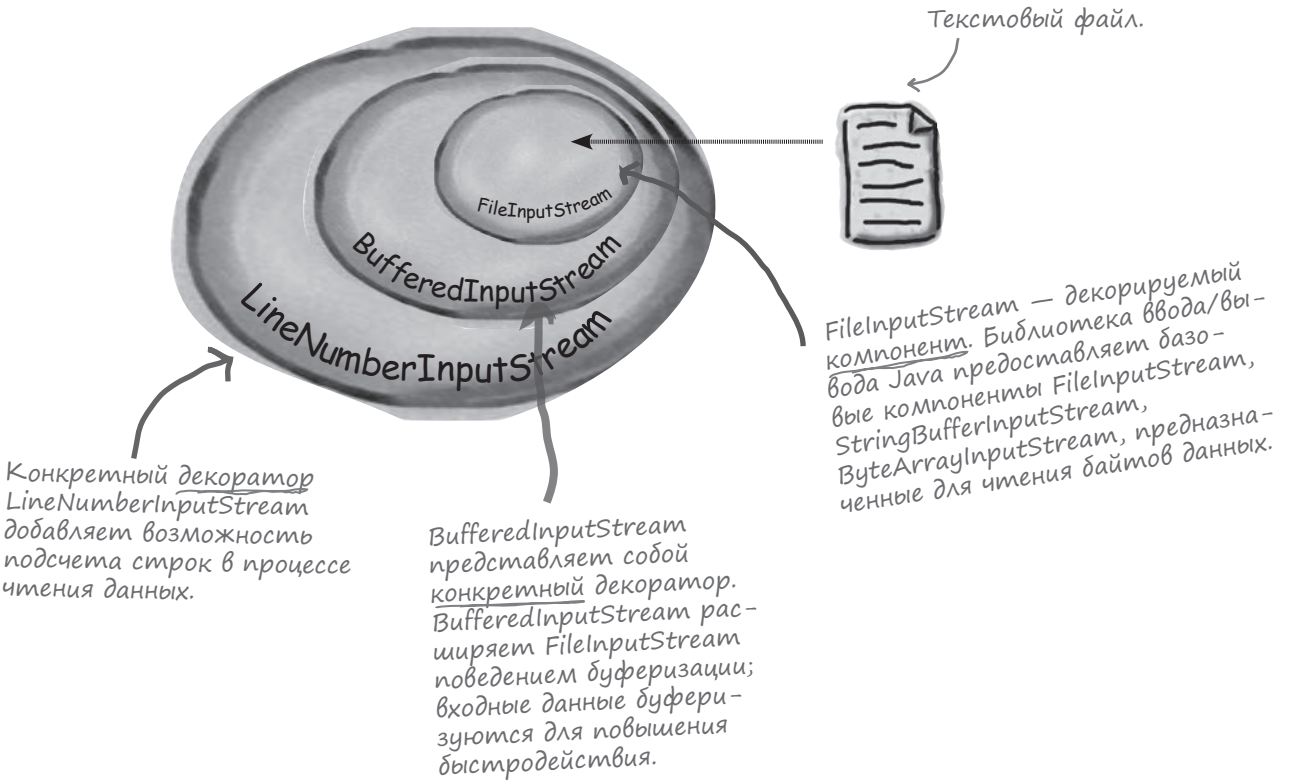
Наши друзья из Starbuzz ввели в меню разные размеры порций. Теперь кофе можно заказать в маленькой, средней или большой чашке. Starbuzz считает размер порции неотъемлемой частью класса кофе, поэтому в класс Beverage были добавлены два новых метода: setSize() и getSize(). Стоимость дополнений также зависит от размера порции, так что, скажем, добавка сои должна стоить 10, 15 или 20 центов для маленькой, средней или большой порции соответственно. Обновленный класс напитков показан ниже.

Как бы вы изменили классы декораторов в соответствии с новыми требованиями?

```
public abstract class Beverage {
        public enum Size { TALL, GRANDE, VENTI };
        Size size = Size.TALL;
        String description = "Unknown Beverage";
        public String getDescription() {
                return description;
        }
        public void setSize(Size size) {
                this.size = size;
        }
        public Size getSize() {
                return this.size;
        }
        public abstract double cost();
}
```

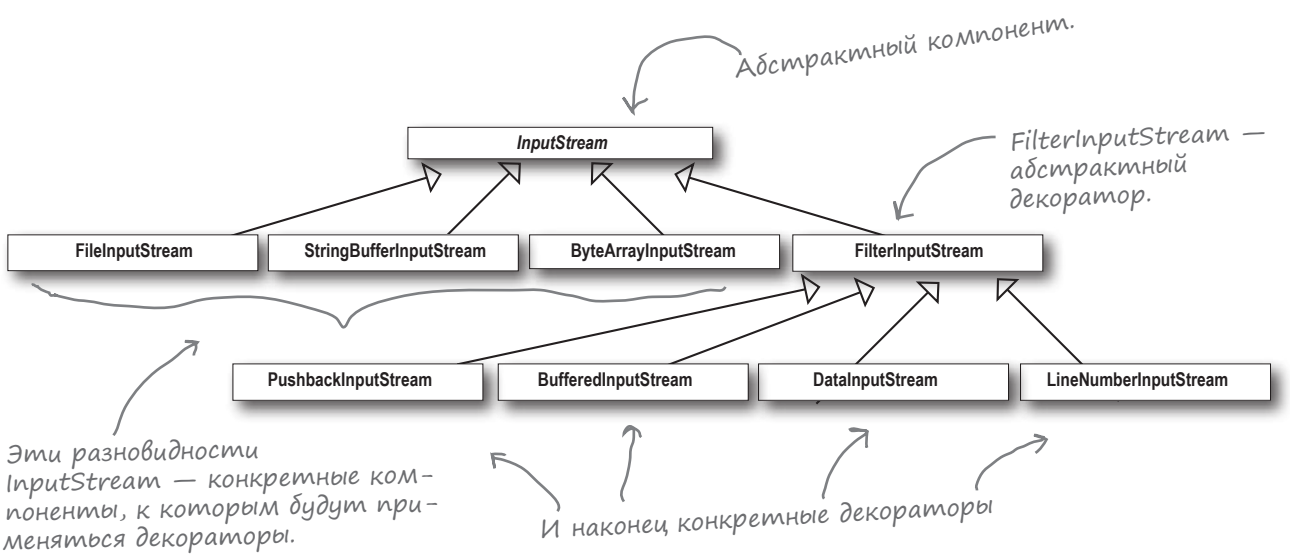
Декораторы в реальном мире: ввод/вывод в языке Java

Количество классов в пакете java.io... *устрашает*. Если вы сказали «ого!» при первом взгляде на этот API (а также при втором и третьем), не огорчайтесь, вы не одиноки. Но теперь, когда вы знакомы с паттерном Декоратор, классы ввода/вывода смотрятся более осмысленно, потому что пакет java.io в основном базируется на паттерне Декоратор. Типичный набор объектов, использующих декораторы для расширения функциональности чтения данных из файла:



BufferedInputStream и ***LineNumber***InputStream расширяют ***Filter***InputStream — абстрактный класс декоратора.

Декорирование классов java.io



Как видите, архитектура не так уж сильно отличается от архитектуры Starbuzz. Полученной информации вполне достаточно для того, чтобы просмотреть документацию java.io и составить декораторы для различных *входных* потоков.

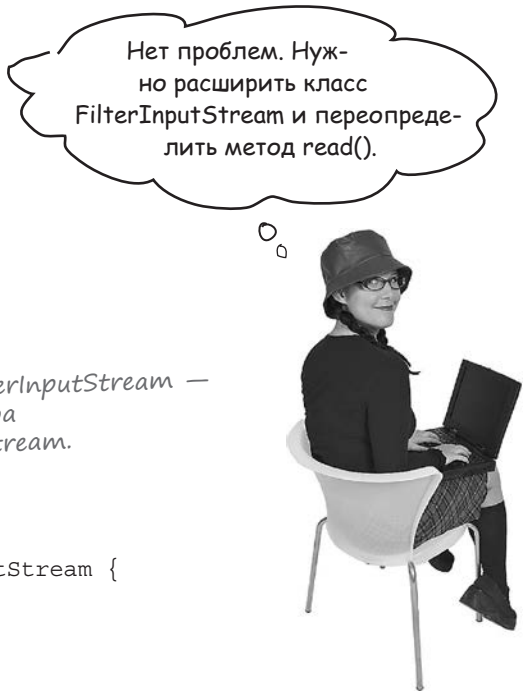
Выходные потоки используют аналогичную архитектуру. Вероятно, вы уже поняли, что потоки Reader/Writer (для символьных данных) достаточно близко отражают архитектуру потоковых классов (с небольшими различиями и расхождениями, но достаточно близко, чтобы вы могли разобраться в происходящем).

Библиотека ввода/вывода Java также подчеркивает один из *недостатков* паттерна Декоратор: иерархии, построенные с использованием этого паттерна, часто состоят из множества мелких классов, в которых трудно разобраться разработчику, пытающемуся использовать API. Но теперь вы знаете, как работает Декоратор, представляете общую картину и при использовании чужого API на базе паттерна Декоратор разберетесь в структуре его классов, чтобы получить доступ к нужному поведению.

Написание собственного декоратора ввода/вывода

Итак, вы знаете паттерн Декоратор и видели диаграмму классов ввода/вывода. У вас есть все необходимое для написания собственного декоратора.

Давайте напишем декоратор, который преобразует все символы верхнего регистра во входном потоке к нижнему регистру. Другими словами, если из потока читается строка «I know the Decorator Pattern therefore I RULE!», то наш декоратор преобразует ее к виду «i know the decorator pattern therefore i rule!»



Не забудьте импортировать java.io... (директива не показана).

Начнем с расширения FilterInputStream — абстрактного декоратора для всех объектов InputStream.

```
public class LowerCaseInputStream extends FilterInputStream {
    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = in.read();
        return (c == -1 ? c : Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int offset, int len) throws IOException {
        int result = in.read(b, offset, len);
        for (int i = offset; i < offset+result; i++) {
            b[i] = (byte)Character.toLowerCase((char)b[i]);
        }
        return result;
    }
}
```

Теперь необходимо реализовать два метода read. Они получают байт (или массив байтов) и преобразуют каждый символ верхнего регистра к нижнему регистру.

НАПОМИНАЕМ: директивы import и package в листингах не приводятся. Полный исходный код примеров можно загрузить на сайте http://wickedlysmart.com/head-first-design-patterns/.

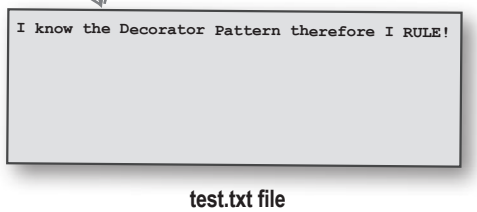
Тестирование декоратора ввода/вывода

Пишем короткий фрагмент кода для тестирования декоратора:

```
public class InputTest {
    public static void main(String[] args) throws IOException {
        int c;
        try {
            InputStream in =
                new LowerCaseInputStream(
                    new BufferedInputStream(
                        new FileInputStream("test.txt")));

            while((c = in.read()) >= 0) {
                System.out.print((char)c);
            }

            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Создаем объект FileInputStream и декорируем его — сначала декоратором BufferedInputStream, затем нашим фильтром LowerCaseInputStream.

Просто используем поток для чтения символов до конца файла и выводим символы в процессе чтения.



test.txt file

Необходимо создать этот файл.

Проверяем:

```
File Edit Window Help DecoratorsRule
% java InputTest
i know the decorator pattern therefore i rule!
%
```

ПАТТЕРНЫ ДЛЯ ВСЕХ

Интервью недели: Признания Декоратора

HeadFirst: Добро пожаловать. Говорят, в последнее время вы находитесь в подавленном настроении?

Декоратор: Да, я знаю, что меня считают модным паттерном проектирования... Но знаете, у меня тоже есть свои проблемы.

HeadFirst: Может, поделитесь с нами своими трудностями?

Декоратор: Конечно. Вам известно, что я могу сделать архитектуру более гибкой, это бесспорно, но у меня есть и своя *темная сторона*. Видите ли, иногда я добавляю в архитектуру много мелких классов, и разобраться в ней становится весьма непросто.

HeadFirst: Приведете пример?

Декоратор: Возьмем библиотеки ввода/вывода Java. Известно, как трудно на первых порах в них разбираться. Но когда разработчик видит, что классы представляют собой набор «оберток» для InputStream, его задача заметно упрощается.

HeadFirst: Выходит, все не так уж плохо. Вы остаетесь замечательным паттерном, а проблема решается повышением квалификации?

Декоратор: К сожалению, это еще не все. Инода пользователи берут клиентский код, работа которого зависит от конкретных типов, и расширяют его декораторами, не продумав все до конца. Одно из моих главных достоинств заключается в том, что *декораторы обычно вводятся в архитектуру прозрачно, а клиенту даже не нужно знать, что он имеет дело с декоратором*. Если код зависит от конкретных типов, а вы применяете обобщенные декораторы, дело добром не кончится.

HeadFirst: Надеемся, все понимают, что при использовании декораторов необходима осмотрительность. Не стоит из-за этого переживать.

Декоратор: Стараюсь. Но есть и другие проблемы: скажем, введение декораторов усложняет код создания экземпляра компонента. Если в архитектуре участвуют декораторы, необходимо не только создать компонент, но и «завернуть» его в сколько-то декораторов.

HeadFirst: На следующей неделе мы будем беседовать с паттернами Фабрика и Строитель. Говорят, они очень помогают в решении этой задачи?

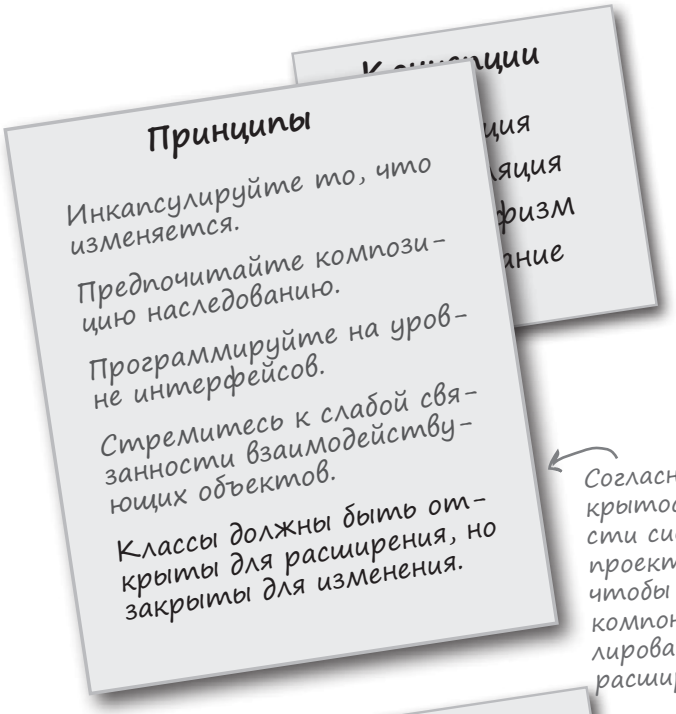
Декоратор: Верно. Мне следовало бы почаще общаться с ними.

HeadFirst: Что ж, мы все считаем, что вы — выдающийся паттерн для создания гибких архитектур и соблюдения принципа открытости/закрытости. Не вешайте нос и смотрите на жизнь веселей!

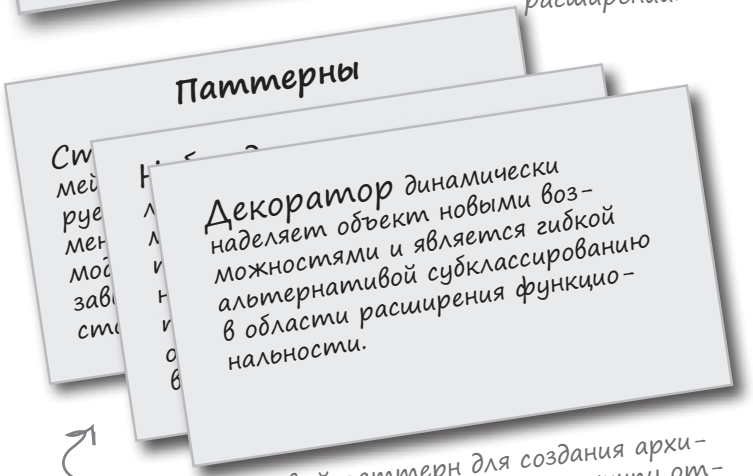
Декоратор: Спасибо, я постараюсь.

Новые инструменты

Подошла к концу очередная глава, а в вашем ОО-инструментарии появился новый принцип и паттерн.



Согласно принципу открытости/закрытости системы должны проектироваться так, чтобы их закрытые компоненты были изолированы от новых расширений.



Наш первый паттерн для создания архитектур, удовлетворяющих принципу открытости/закрытости. Или не первый? А может, один из описанных ранее паттернов тоже следовал этому принципу?

КЛЮЧЕВЫЕ МОМЕНТЫ

- Наследование — одна из форм расширения, но оно не всегда обеспечивает гибкость архитектуры.
- Следует предусмотреть возможность расширения поведения без изменения существующего кода.
- Композиция и делегирование часто используются для динамического добавления нового поведения.
- Паттерн Декоратор предоставляет альтернативу субклассированию в области расширения поведения.
- Типы декораторов соответствуют типам декорируемых компонентов (соответствие достигается посредством наследования или реализации интерфейса).
- Декораторы изменяют поведение компонентов, добавляя новую функциональность до и (или) после (или даже вместо) вызовов методов компонентов.
- Компонент может декорироваться любым количеством декораторов.
- Декораторы обычно прозрачны для клиентов компонента (если клиентский код не зависит от конкретного типа компонента).

Возьми в руку карандаш
Решение

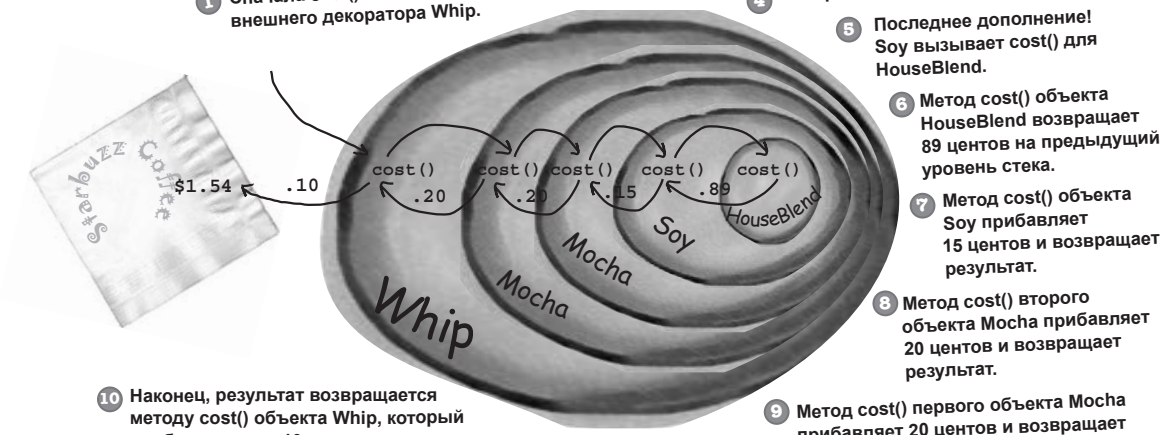
Напишите методы cost() для следующих классов (подойдет и псевдокод Java). Наше решение выглядит так:

```
public class Beverage {

// Объявление переменных экземпляров для milkCost,
// soyCost, mochaCost и whipCost, а также
// get/set-методов для дополнений

public double cost() {

     float condimentCost = 0.0;
     if (hasMilk()) {
         condimentCost += milkCost;
     }
     if (hasSoy()) {
         condimentCost += soyCost;
     }
     if (hasMocha()) {
         condimentCost += mochaCost;
     }
     if (hasWhip()) {
         condimentCost += whipCost;
     }
     return condimentCost;
  }
}


public class DarkRoast extends Beverage {

    public DarkRoast() {
        description = "Most Excellent Dark Roast";
    }

    public double cost() {

        return 1.99 + super.cost();

    }
}
```

Возьми в руку карандаш
Решение

Обучение баристы

«латте с двойным шоколадом, соей и взбитыми сливками»



1. Сначала cost() вызывается для внешнего декоратора Whip.
2. Whip вызывает cost() для Mocha.
3. Mocha вызывает cost() для другого объекта Mocha.
4. Теперь Mocha вызывает cost() для Soy.
5. Последнее дополнение! Soy вызывает cost() для HouseBlend.
6. Метод cost() объекта HouseBlend возвращает 89 центов на предыдущий уровень стека.
7. Метод cost() объекта Soy прибавляет 15 центов и возвращает результат.
8. Метод cost() второго объекта Mocha прибавляет 20 центов и возвращает результат.
9. Метод cost() первого объекта Mocha прибавляет 20 центов и возвращает результат.
10. Наконец, результат возвращается методу cost() объекта Whip, который прибавляет еще 10 центов, и мы получаем итоговую стоимость $1.54.

Возьми в руку карандаш
Решение

Наши друзья из Starbuzz ввели в меню разные размеры порций. Теперь кофе можно заказать в маленькой, средней или большой чашке. Starbuzz считает размер порции неотъемлемой частью класса кофе, поэтому в класс Beverage были добавлены два новых метода: setSize() и getSize(). Стоимость дополнений также зависит от размера порции, так что, скажем, добавка сои должна стоить 10, 15 или 20 центов для маленькой, средней или большой порции соответственно.

Как бы вы изменили классы декораторов в соответствии с новыми требованиями?

```
public abstract class CondimentDecorator extends Beverage {
    public Beverage beverage;
    public abstract String getDescription();

    public Size getSize() {
        return beverage.getSize();
    }
}
```

Вызов getSize() должен передаваться декорируемым объектам. Кроме того, этот метод следует переместить в абстрактный класс, так как он используется всеми декораторами.

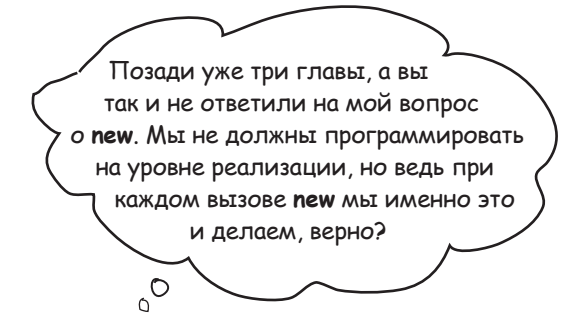
```
public class Soy extends CondimentDecorator {
    public Soy(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Soy";
    }

    public double cost() {
        double cost = beverage.cost();
        if (beverage.getSize() == Size.TALL) {
            cost += .10;
        } else if (beverage.getSize() == Size.GRANDE) {
            cost += .15;
        } else if (beverage.getSize() == Size.VENTI) {
            cost += .20;
        }
        return cost;
    }
}
```

Здесь мы получаем размер напитка (вызов передается вниз до уровня конкретного напитка), после чего прибавляем стоимость.

4 Паттерн Фабрика

Домашняя ОО-выпечка

Приготовьтесь заняться выпечкой *объектов* в слабосвязанных *ОО*-архитектурах. Создание объектов отнюдь не сводится к простому вызову оператора *new*. Оказывается, создание экземпляров не всегда должно осуществляться открыто; оно часто создает проблемы *сильного связывания*. А ведь вы *этого* не хотите, верно? Паттерн Фабрика спасет вас от неприятных зависимостей.



Видим *new* — подразумеваем *конкретный*.

Да, при использовании **new** вы создаете экземпляр конкретного класса, поэтому эта операция относится к уровню реализации, а не интерфейса. А вы уже знаете, что привязка кода к конкретному классу делает его менее гибким и устойчивым к изменениям.

```
Duck duck = new MallardDuck();
```

Использование интерфейсов делает код более гибким.

Но создается экземпляр конкретного класса!

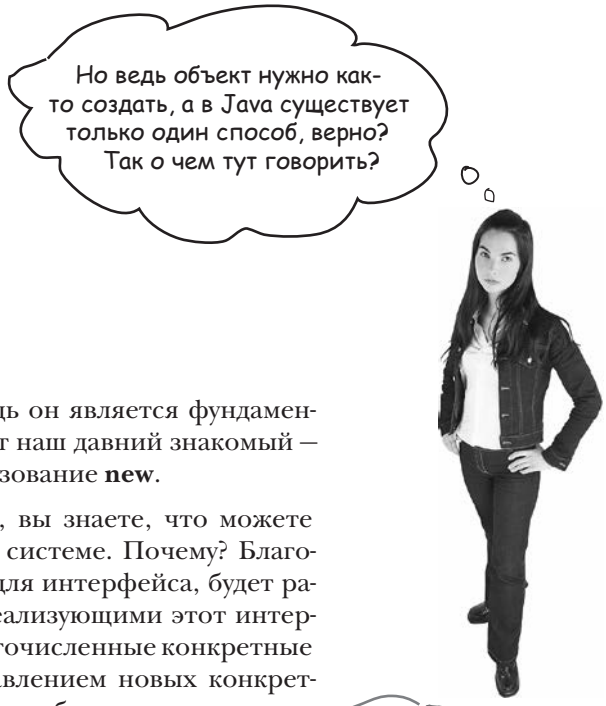
При использовании групп взаимосвязанных конкретных классов часто приходится писать код следующего вида:

```
Duck duck;

if (picnic) {
    duck = new MallardDuck();
} else if (hunting) {
    duck = new DecoyDuck();
} else if (inBathTub) {
    duck = new RubberDuck();
}
```

Иерархия состоит из многих классов, и класс создаваемого экземпляра определяется только во время выполнения.

В программе создаются экземпляры разных конкретных классов, причем класс выбирается во время выполнения в зависимости от неких условий.

Когда настанет время изменения или расширения, вам придется снова открыть код и разобраться в том, что нужно добавить (или удалить). Часто подобный код размещается в разных частях приложения, что основательно затрудняет его сопровождение и обновление.



Чем плох оператор *new*?

С оператором **new** все в порядке — ведь он является фундаментальной частью Java. Проблемы создает наш давний знакомый — ИЗМЕНЕНИЕ и его влияние на использование **new**.

Программируя на уровне интерфейса, вы знаете, что можете оградить себя от многих изменений в системе. Почему? Благодаря полиморфизму код, написанный для интерфейса, будет работать с любыми новыми классами, реализующими этот интерфейс. Но если в коде используются многочисленные конкретные классы, его придется изменять с добавлением новых конкретных классов. Иначе говоря, код перестает быть «закрытым для изменения» — для расширения новыми конкретными типами его придется открывать.

Так что же делать? В подобных ситуациях полезно вернуться к принципам проектирования и поискать в них подсказки. Напоминаем: первый принцип, непосредственно относящийся к изменениям, предлагает *определить аспекты, которые будут изменяться, и отделить их от тех, которые останутся неизменными.*

Помните, что архитектуры должны быть «открыты для расширения, но закрыты для изменения». За подробностями обращайтесь к главе 3.

МОЗГОВОЙ ШТУРМ

А если взять все части приложения, в которых создаются экземпляры конкретных классов, и отделить их от других частей приложения? Как бы вы это сделали?

Определение изменяемых аспектов

Допустим, вы открыли пиццерию. Будучи современным предпринимателем из Объективиля, вы пишете код следующего вида:

```
Pizza orderPizza() {
        Pizza pizza = new Pizza();

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
}
```

По соображениям гибкости хотелось бы использовать абстрактный класс или интерфейс, но их экземпляры создать невозможно.

Но существует много разновидностей пиццы...

Поэтому вы добавляете код, который *определяет* нужный тип пиццы, а затем переходит к ее *изготовлению*:

```
Pizza orderPizza(String type) {
        Pizza pizza;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("greek") {
            pizza = new GreekPizza();
        } else if (type.equals("pepperoni") {
            pizza = new PepperoniPizza();
        }

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
}
```

Тип пиццы передается orderPizza при вызове.

В зависимости от типа мы создаем экземпляр нужного конкретного класса и присваиваем его переменной pizza. Обратите внимание: каждый тип пиццы должен реализовать интерфейс Pizza.

Получив объект Pizza, мы готовим его, выпекаем, разрезаем и кладем в коробку!

Каждый подтип Pizza (CheesePizza, VeggiePizza и т. д.) умеет готовить себя.

Добавление новых типов пиццы

Вы прослышали, что все конкуренты включили в свои меню модные виды пиццы: с мидиями и вегетарианскую. Разумеется, чтобы не отстать, вы должны включить их в свое меню. А греческая пицца в последнее время продается плохо, поэтому вы решаете отказаться от этой позиции:

```
Pizza orderPizza(String type) {
    Pizza pizza;

    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("greek") {
        pizza = new GreekPizza();
    } else if (type.equals("pepperoni") {
        pizza = new PepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new ClamPizza();
    } else if (type.equals("veggie") {
        pizza = new VeggiePizza();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

Этот код НЕ закрыт для изменения. При изменении ассортимента придется вернуться к коду и вносить в него изменения.

С течением времени вам придется изменять этот код снова и снова.

В целом процедура приготовления, выпечки и упаковки пиццы остается неизменной в течение многих лет. Таким образом, этот код меняться не должен — меняются только виды пиццы, с которыми он работает.

Выбор конкретного класса для создания экземпляра усложняет метод orderPizza() и не позволяет закрыть его для изменений. А если одни аспекты системы изменяются, а другие остаются неизменными — пора заняться инкапсуляцией.

Инкапсуляция создания объектов

Итак, код создания объекта следует исключить из метода orderPizze(). Но как? Мы переместим его в другой объект, единственной задачей которого будет создание объектов пиццы.

```
if (type.equals("cheese")) {
    pizza = new CheesePizza();
  } else if (type.equals("pepperoni") {
    pizza = new PepperoniPizza();
  } else if (type.equals("clam") {
    pizza = new ClamPizza();
  } else if (type.equals("veggie") {
    pizza = new VeggiePizza();
}
```

```
Pizza orderPizza(String type) {
       Pizza pizza;

       pizza.prepare();
       pizza.bake();
       pizza.cut();
       pizza.box();
       return pizza;
}
```

Извлекаем код создания объекта из метода orderPizza.

Что будет на этом месте?

Код выделяется в объект, который занимается только созданием пиццы, и ничем более. Если другому объекту понадобится создать пиццу, он обратится к нему с запросом.

У нового объекта имеется подходящее имя: мы назовем его Фабрикой.

Фабрика инкапсулирует подробности создания объектов. Метод orderPizza() становится обычным клиентом фабрики SimplePizzaFactory. Каждый раз, когда ему понадобится новая пицца, он просит фабрику ее создать. Прошли те времена, когда метод orderPizza() должен был знать, чем греческая пицца отличается от вегетарианской. Теперь метод orderPizza() знает лишь то, что полученный им объект реализует интерфейс Pizza для вызова методов prepare(), bake(), cut() и box().

Осталось разобраться с некоторыми подробностями; например, чем заменяется код создания объекта в методе orderPizza()? Давайте реализуем простую фабрику пиццы и узнаем...

Построение Простой Фабрики для пиццы

Начнем с самой фабрики. Наша задача — определить класс, инкапсулирующий создание объектов для всех видов пиццы. Вот как он выглядит...

Класс SimplePizzaFactory занимается исключительно созданием пиццы для своих клиентов.

В фабрике определяется метод createPizza(), который будет использоваться всеми клиентами для создания новых объектов.

```
public class SimplePizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

Код, выделенный из метода orderPizza().

Код параметризуется по типу пиццы, как и наш исходный метод orderPizza().

В: И что нам это дает? Проблема просто перекладывается на другой объект.

О: Класс SimplePizzaFactory может использоваться многими клиентами. Пока мы видим только метод orderPizza(), но фабрика также может использоваться классом PizzaShopMenu для получения пиццы по описанию и цене. Класс HomeDelivery может использовать объекты еще как-нибудь иначе — при этом он тоже остается клиентом фабрики.

Часто Задаваемые Вопросы

Таким образом, создание объекта инкапсулируется в одном классе, и будущие изменения реализации придется вносить только в одном месте.

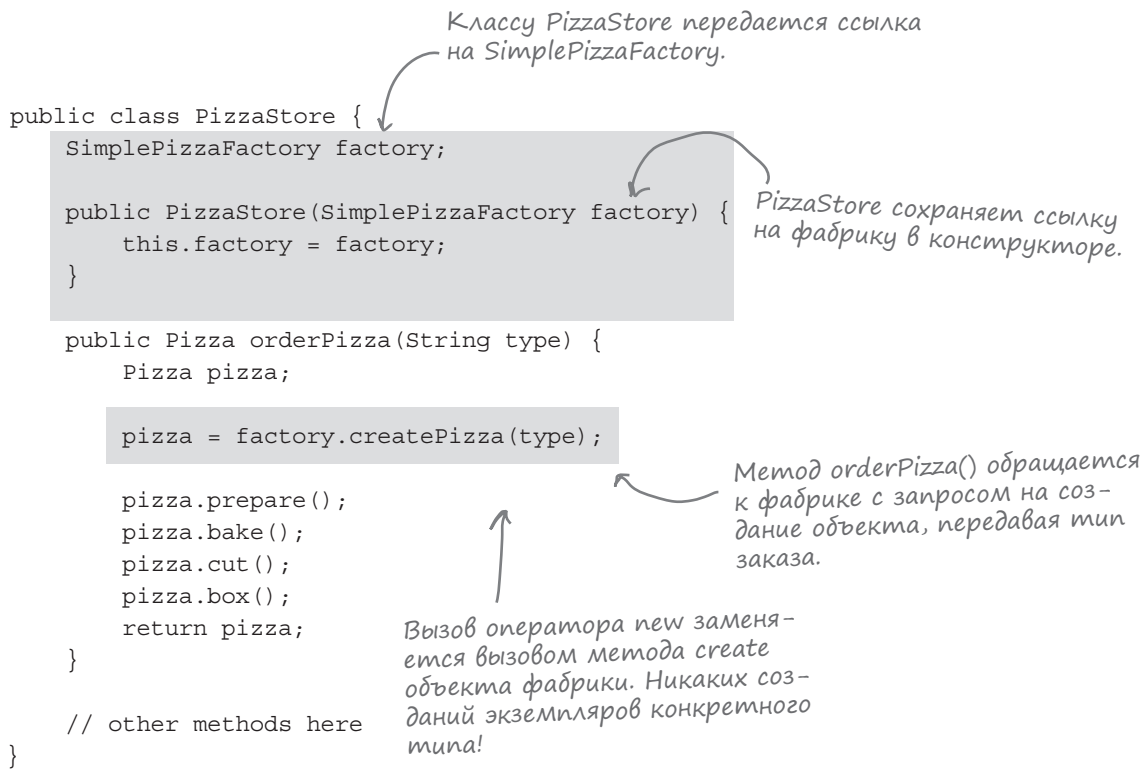
И не забывайте: из клиентского кода также исключаются операции создания экземпляров конкретных типов!

В: Мне встречалось похожее решение, в котором фабрика объявлялась статическим методом. Чем они различаются?

О: Зачем фабрики оформляются в виде статических методов? Чтобы метод create можно было вызывать и без создания экземпляра объекта. С другой стороны, теряется возможность субклассирования и изменения поведения метода create.

Переработка класса PizzaStore

Пора заняться клиентским кодом. Мы хотим, чтобы все операции создания объектов выполнялись фабрикой. Для этого необходимо внести следующие изменения:




```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }

    // other methods here
}
```

МОЗГОВОЙ ШТУРМ

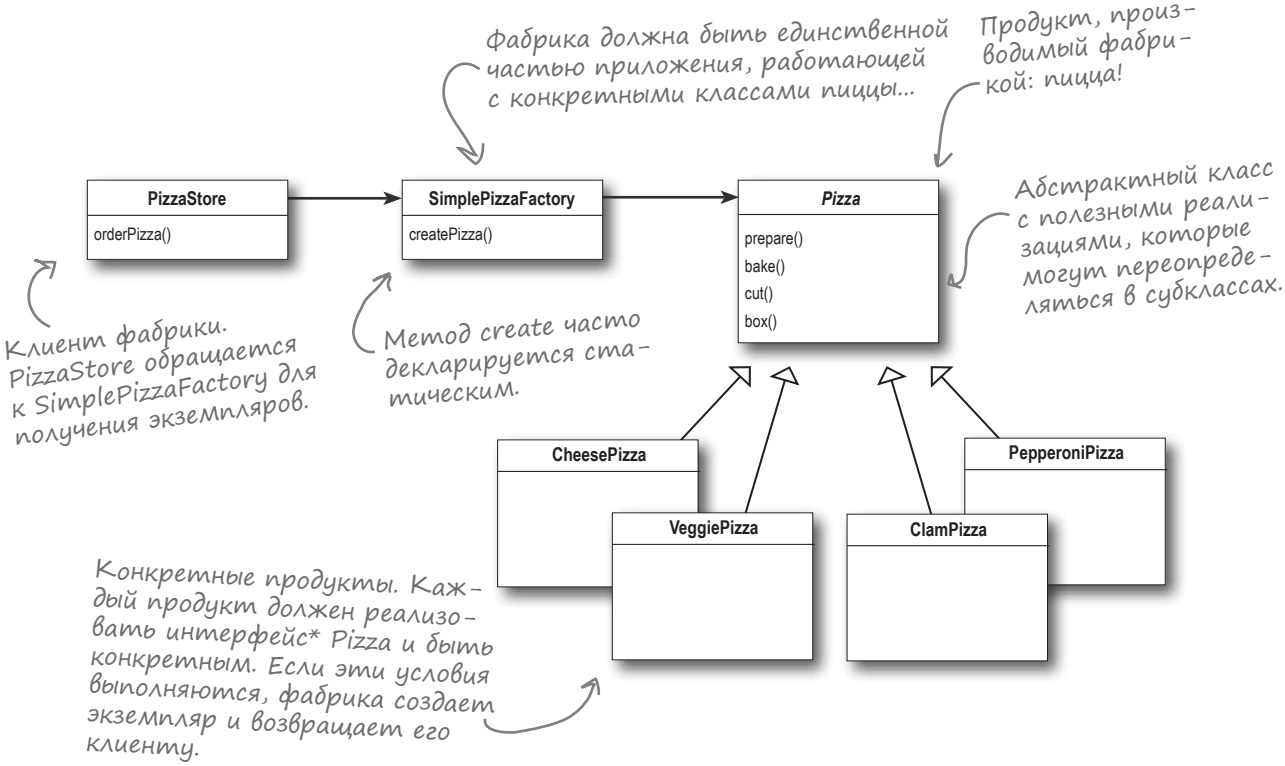
Композиция объектов позволяет (среди прочего) динамически изменять поведение во время выполнения, так как мы можем свободно подключать и отключать реализации. Как использовать эту возможность в PizzaStore?

Например, можно предоставить себе фабрики пиццы по стандартам, принятым в разных регионах: Нью-Йорк, Чикаго, Калифорния (да, и не забудьте про Нью-Хейвен!).

Определение Простой Фабрики

Строго говоря, Простая Фабрика не является паттерном проектирования; скорее, это идиома программирования. Но она используется так часто, что мы решили упомянуть ее здесь. Некоторые разработчики путают эту идиому с паттерном Фабрика, так что если у вас когда-нибудь возникнет неловкое молчание в беседе с малознакомым программистом — считайте, у вас есть хорошая тема для обсуждения.

Хотя Простая Фабрика не является ПОЛНОЦЕННЫМ паттерном, это не значит, что ее не стоит изучить более подробно. Рассмотрим диаграмму классов нашего нового магазина пиццы:



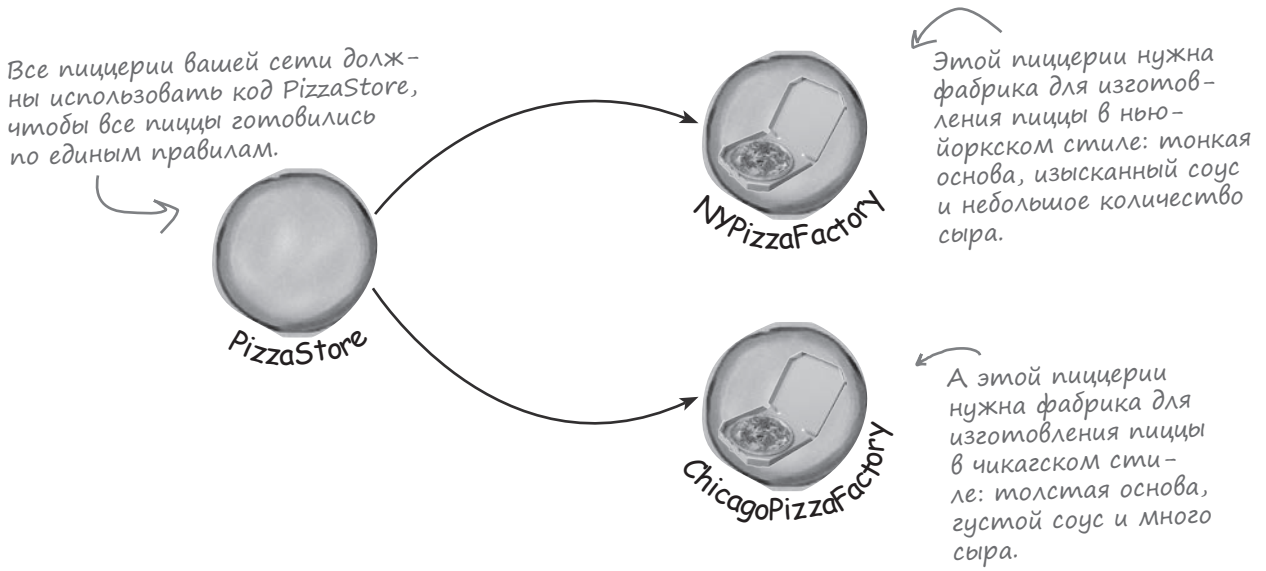
Впрочем, Простая Фабрика была всего лишь разминкой. Сейчас мы изучим две другие разновидности Фабрики, и обе они являются полноценными паттернами. Больше паттернов, больше пиццы!

*В паттернах проектирования фраза «реализация интерфейса» НЕ ВСЕГДА обозначает «класс, реализующий интерфейс Java с ключевым словом implements в объявлении». В более общем смысле реализация конкретным классом метода супертипа (класса ИЛИ интерфейса) считается «реализацией интерфейса» этого супертипа.

Расширение бизнеса

Дела вашей пиццерии в Объективиле идут так хорошо, что вы сокрушили всех конкурентов, и теперь вы планируете открыть целую сеть пиццерий PizzaStore по всей стране. Вы как правообладатель желаете обеспечить высокое качество пиццы в заведениях, работающих под вашей маркой, и поэтому требуете, чтобы они использовали ваш проверенный код.

Но что делать с региональными различиями? Заведения могут предлагать разные стили пиццы в зависимости от своего местонахождения (Нью-Йорк, Чикаго, Калифорния и т. д.) и предпочтений местных ценителей итальянской кухни.



Мы рассмотрели один способ...

SimplePizzaFactory заменяется тремя разными фабриками: NYPizzaFactory, ChicagoPizzaFactory и CaliforniaPizzaFactory. В этом случае PizzaStore связывается с подходящей фабрикой посредством композиции.

Давайте посмотрим, как будет выглядеть реализация...

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.order("Veggie");
```

Сначала создаем фабрику для пиццы в нью-йоркском стиле.

Затем создаем объект PizzaStore и передаем ему ссылку на фабрику.

...и при создании экземпляров получаем пиццу в нью-йоркском стиле.

```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.order("Veggie");
```

Аналогично для пиццерии из Чикаго: создаем фабрику для объектов и связываем ее с классом пиццерии. Фабрика создает пиццу в чикагском стиле.

Как избежать проблем с качеством?

Исследование рынка показало, что пиццерии вашей сети используют ваши фабрики для создания пиццы, но в остальных фазах процесса применяют свои доморощенные правила: выбирают свой режим выпечки, забывают нарезать пиццу, используют упаковку сторонних фирм и т. д.

После некоторых размышлений становится ясно: нам нужна инфраструктура, которая связывает пиццерию с процессом создания пиццы, но при этом сохраняет достаточную гибкость.

До создания SimplePizzaFactory наш код изготовления пиццы был привязан к PizzaStore, но гибкости не было и в помине. Что же делать?

В хорошей торговой сети такого быть НЕ ДОЛЖНО. Кто знает, что он кладет в свою пиццу?

Инфраструктура для пиццерии

Итак, мы хотим локализовать все операции по изготовлению пиццы в классе PizzaStore, но при этом сохранить для пиццерий достаточную гибкость для выбора своего регионального стиля. И это *можно* сделать!

Для этого мы вернем метод createPizza() в класс PizzaStore, но в виде **абстрактного метода**, а затем создадим субкласс для каждого регионального стиля.

Начнем с изменений в PizzaStore:

Класс PizzaStore стал абстрактным (почему — см. ниже).

```
public abstract class PizzaStore {

        public Pizza orderPizza(String type) {
                Pizza pizza;

                pizza = createPizza(type);

                pizza.prepare();
                pizza.bake();
                pizza.cut();
                pizza.box();

                return pizza;
        }

        abstract Pizza createPizza(String type);
}
```

Метод createPizza снова принадлежит PizzaStore, а не классу фабрики.

Здесь ничего не изменилось...

Объект фабрики перемещается в этот метод.

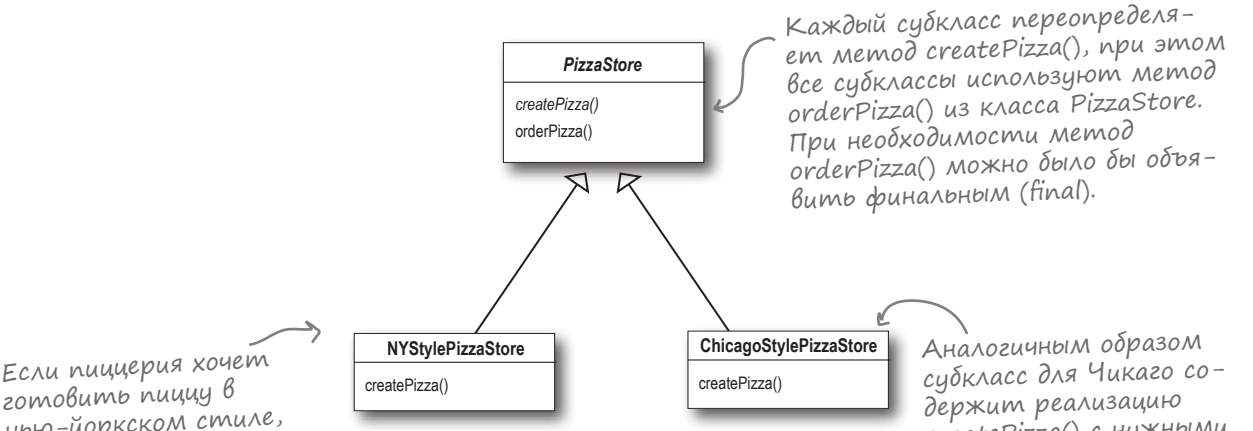
«Фабричный метод» стал абстрактным методом PizzaStore.

Мы создадим субкласс для каждой региональной разновидности пиццерии (NYPizzaStore, ChicagoPizzaStore, CaliforniaPizzaStore), и каждый субкласс будет сам принимать решения относительно создания объекта. Давайте посмотрим, как это происходит.

Принятие решений в субклассах

Итак, метод orderPizza() класса PizzaStore уже содержит проверенную систему оформления заказа, и вы хотите, чтобы во всех пиццериях эта процедура оставалась одинаковой.

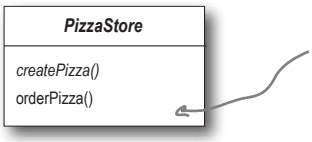
Региональные версии PizzaStore различаются стилем своей пиццы — у нью-йоркской пиццы тонкая основа, у чикагской — толстая, и т. д. Мы инкапсулируем все эти различия в методе createPizza() и сделаем его ответственным за создание правильного вида пиццы. Для этого каждому субклассу PizzaStore будет разрешено самостоятельно определить свой метод createPizza(). Итак, у нас получается группа конкретных субклассов PizzaStore со своими разновидностями пиццы, причем все эти классы входят в инфраструктуру PizzaStore и продолжают использовать проверенный метод orderPizza().



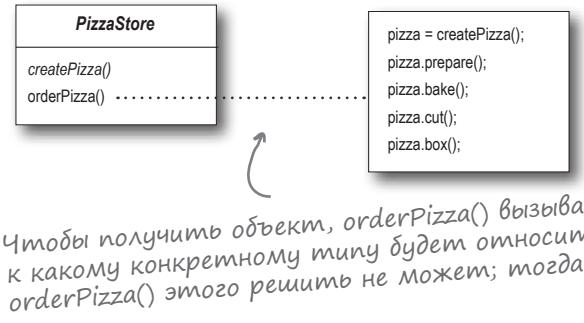

```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new NYStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new NYStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new NYStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new NYStyleVeggiePizza();
    }
}
```

```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new ChicagoStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new ChicagoStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new ChicagoStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new ChicagoStyleVeggiePizza();
    }
}
```

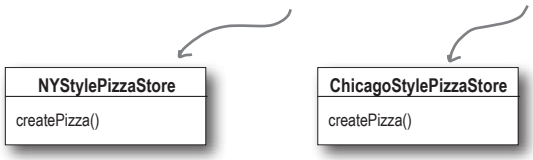
Взгляните на происходящее с точки зрениям метода orderPizza() класса PizzaStore: он определяется в абстрактном классе PizzaStore, но конкретные типы создаются только в субклассах.



Метод orderPizza() выполняет целый ряд операций с объектом Pizza (prepare, bake, cut, box), но, так как класс Pizza является абстрактным, orderPizza() не знает, с какими конкретными классами он работает. Иначе говоря, здесь используется слабая связь!



Когда orderPizza() вызывает createPizza(), управление передается одному из субклассов. Какая именно из конкретных разновидностей пиццы будет создана? Это зависит от типа пиццерии: NYStylePizzaStore или ChicagoStylePizzaStore.



Итак, есть ли здесь решение, принимаемое субклассами во время выполнения? Нет, но с точки зрения orderPizza() при выборе NYStylePizzaStore этот субкласс определяет разновидность создаваемой пиццы. Таким образом, «решение» принимают не субклассы, а *вы*, когда выбираете нужный тип пиццерии. Но субклассы определяют тип пиццы, которая будет создана по запросу.

Субклассы PizzaStore

Региональные пиццерии получают всю функциональность PizzaStore бесплатно. Им остается субклассировать PizzaStore и предоставить метод cratePizza(), реализующий их местный стиль приготовления пиццы. В нашей модели поддерживаются три основных региональных стиля.

Нью-йоркский стиль приготовления пиццы:

createPizza() возвращает объект Pizza, а субкласс несет полную ответственность за создаваемый конкретный экземпляр Pizza.

Класс NYPizzaStore расширяет PizzaStore, поэтому он наследует метод orderPizza()*.

```
public class NYPizzaStore extends PizzaStore {
    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
       } else if (item.equals("clam")) {
            return new NYStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        } else return null;
    }
}
```

Необходимо реализовать метод createPizza(), так как в PizzaStore он объявлен абстрактным.

Здесь создаются конкретные классы. Для каждого типа пиццы мы создаем реализацию в нью-йоркском стиле.

* Метод orderPizza() суперкласса понятия не имеет, какой из типов пиццы мы создаем; он знает лишь то, что пиццу можно приготовить, выпечь, нарезать и упаковать!

Когда мы закончим работу над субклассами PizzaStore, можно будет с чистой совестью заказать пиццу или две. Но сначала переверните страницу и попробуйте самостоятельно построить региональные классы пиццерий для Чикаго и Калифорнии.

Возьми в руку карандаш

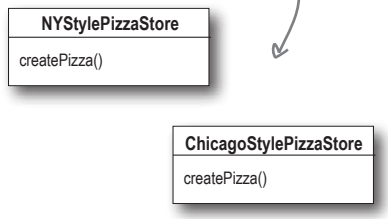
Класс NYPizzaStore уже готов; еще два класса, и бизнес можно будет запускать! Запишите в этой врезке реализации PizzaStore для Чикаго и Калифорнии:

Объявление Фабричного Метода

Всего пара преобразований в PizzaStore — и мы перешли от объекта, создающего экземпляры конкретных классов, к набору классов, решающих ту же задачу. Давайте присмотримся повнимательнее:

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    protected abstract Pizza createPizza(String type);

    // Другие методы
}
```

Субклассы PizzaStore создают экземпляры объектов за нас при вызове createPizza().



Вся ответственность за создание экземпляров Pizza перемещена в метод, действующий как фабрика.

Код под увеличительным стеклом

Фабричный метод отвечает за создание объектов и инкапсулирует эту операцию в субклассе. Таким образом клиентский код в суперклассе отделяется от кода создания объекта в субклассе.

Фабричный метод может быть параметризован для выбора между несколькими разновидностями продукта.

```
abstract Product factoryMethod(String type)
```

Фабричный метод объявляется абстрактным, чтобы субклассы предоставили реализацию создания объектов.

Фабричный метод возвращает некий тип Product, обычно используемый методами, определенными в суперклассе.

Фабричный метод изолирует клиента (код суперкласса — такой, как orderPizza()) от информации о конкретном типе создаваемого продукта.

Заказ пиццы с использованием Фабричного Метода

Как происходит оформление заказа?

1. Для начала Джоэлу и Итану понадобятся экземпляры субклассов PizzaStore. Джоэл создает экземпляр ChicagoPizzaStore, а Итан — экземпляр NYPizzaStore.

2. Имея экземпляр PizzaStore, Итан и Джоэл вызывают метод orderPizza() и передают ему тип нужной пиццы (вегетарианская, с сыром и т. д.).

3. Для создания объекта вызывается метод createPizza(), который определяется в двух субклассах: NYPizzaStore и ChicagoPizzaStore. В нашем определении NYPizzaStore создает экземпляр нью-йоркской пиццы, а ChicagoPizzaStore — экземпляр чикагской пиццы. В любом случае методу orderPizza() возвращается экземпляр Pizza.

4. Метод orderPizza() не знает, какая разновидность пиццы была создана, но знает, что это именно пицца. Соответственно, он готовит, выпекает, нарезает и упаковывает пиццу для Итана и Джоэла.

Что происходит в процессе заказа...

1 **Проследим за заказом Итана. Сначала создается объект NYPizzaStore:**

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

Создание экземпляра NYPizzaStore.

2 **Пиццерия построена, теперь можно принять заказ:**

```
nyPizzaStore.orderPizza("cheese");
```

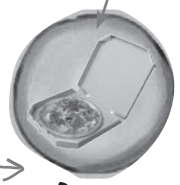
Метод orderPizza() вызывается для экземпляра nyPizzaStore (выполняется метод, определенный в PizzaStore).

createPizza("cheese")

3 **Метод orderPizza() вызывает метод createPizza():**

```
Pizza pizza = createPizza("cheese");
```

Напоминаем: фабричный метод createPizza() реализуется в субклассе. В данном случае он возвращает нью-йоркскую пиццу с сыром.



4 **В итоге мы получаем сырую пиццу, и метод orderPizza() завершает ее приготовление:**

```
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
```

Все эти методы определяются в конкретном объекте, который возвращается фабричным методом createPizza(), определяемым в NYPizzaStore.

Метод orderPizza() получает объект Pizza, но не знает его конкретный субкласс.

Не хватает только одного: ПИЦЦЫ!

Без пиццы наше заведение вряд ли будет популярным. Пора заняться ее реализацией:

Начнем с абстрактного класса Pizza, который станет суперклассом для конкретных классов пиццы.

```
public abstract class Pizza {
    String name;
    String dough;
    String sauce;
    ArrayList<String> toppings = new ArrayList<String>();

    void prepare() {
        System.out.println("Preparing " + name);
        System.out.println("Tossing dough...");
        System.out.println("Adding sauce...");
        System.out.println("Adding toppings: ");
        for (String topping : toppings) {
            System.out.println("   " + topping);
        }
    }

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }

    void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }

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

Каждый объект Pizza содержит название, тип основы, тип соуса и набор добавок.

Абстрактный класс предоставляет реализации по умолчанию для основных методов.

Приготовление пиццы состоит из нескольких шагов, выполняемых в определенной последовательности.

НАПОМИНАЕМ: директивы import и package в листингах не приводятся. Полный исходный код примеров можно загрузить на сайте wickedlysmart (адрес приведен в Введении).

Остается определить конкретные субклассы... Как насчет определения нью-йоркской и чикагской пиццы с сыром?

```
public class NYStyleCheesePizza extends Pizza {
    public NYStyleCheesePizza() {
        name = "NY Style Sauce and Cheese Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";

        toppings.add("Grated Reggiano Cheese");
    }
}
```

Нью-йоркская пицца готовится с соусом «марина-ра» на тонкой основе.

Одна добавка: сыр «реджано»!

```
public class ChicagoStyleCheesePizza extends Pizza {
    public ChicagoStyleCheesePizza() {
        name = "Chicago Style Deep Dish Cheese Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";

        toppings.add("Shredded Mozzarella Cheese");
    }

    void cut() {
        System.out.println("Cutting the pizza into square slices");
    }
}
```

Чикагская пицца использует томатный соус и готовится на толстой основе.

В чикагскую пиццу кладут много сыра «моцарелла»!

Чикагская пицца также переопределяет метод cut(): она нарезается не клиньями, а квадратами.

Сколько можно ждать, несите пиццу!

```
public class PizzaTestDrive {

    public static void main(String[] args) {
        PizzaStore nyStore = new NYPizzaStore();
        PizzaStore chicagoStore = new ChicagoPizzaStore();

        Pizza pizza = nyStore.orderPizza("cheese");
        System.out.println("Ethan ordered a " + pizza.getName() + "\n");

        pizza = chicagoStore.orderPizza("cheese");
        System.out.println("Joel ordered a " + pizza.getName() + "\n");
    }
}
```

Сначала создаем два объекта пиццерий.

Затем используем один из них для выполнения заказа Итана.

А другой — для заказа Джоэла.

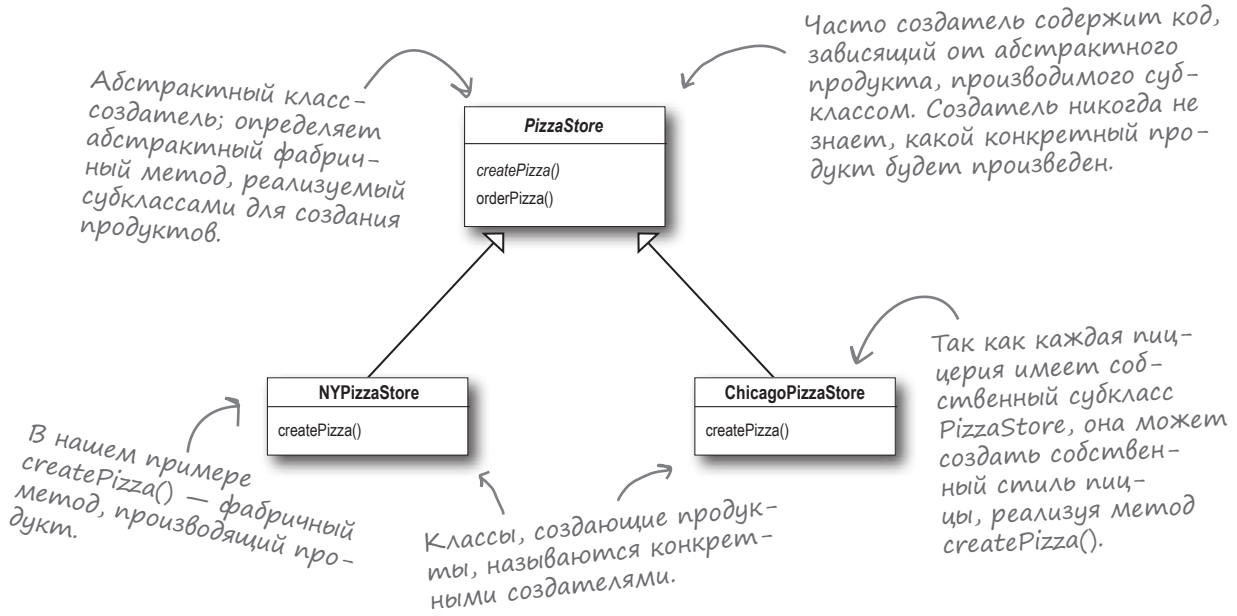
```
File  Edit  Window  Help  YouWantMootzOnThatPizza?

%java PizzaTestDrive

Preparing NY Style Sauce and Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Grated Regiano cheese
Bake for 25 minutes at 350
Cutting the pizza into diagonal slices
Place pizza in official PizzaStore box
Ethan ordered a NY Style Sauce and Cheese Pizza

Preparing Chicago Style Deep Dish Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Shredded Mozzarella Cheese
Bake for 25 minutes at 350
Cutting the pizza into square slices
Place pizza in official PizzaStore box
Joel ordered a Chicago Style Deep Dish Cheese Pizza
```

Обе пиццы готовятся со всеми добавками, выпекаются, нарезаются и упаковываются. Суперклассу не нужно знать подробности — субкласс решает все проблемы, просто создавая правильный экземпляр.

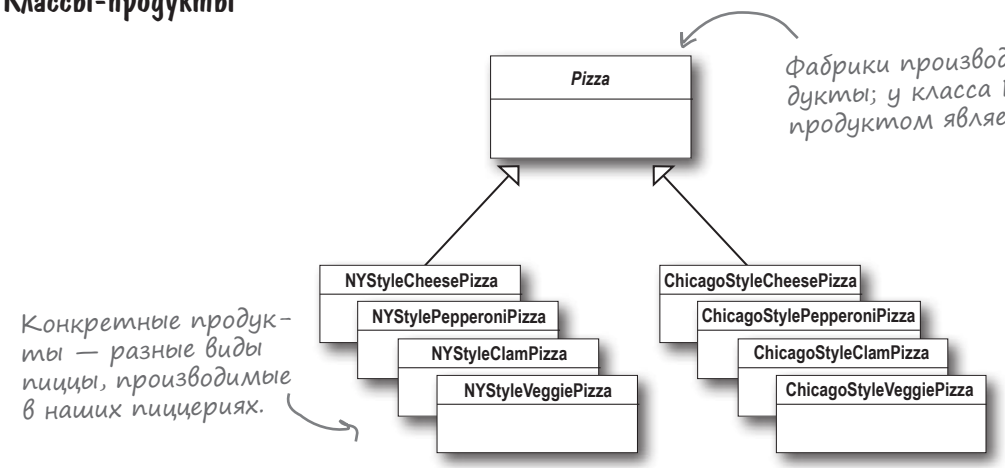
Пора познакомиться с паттерном Фабричный Метод

Все фабричные паттерны инкапсулируют операции создания объектов. Паттерн Фабричный Метод позволяет субклассам решить, какой объект следует создать. На следующих диаграммах классов представлены основные участники этого паттерна:

Классы-создатели



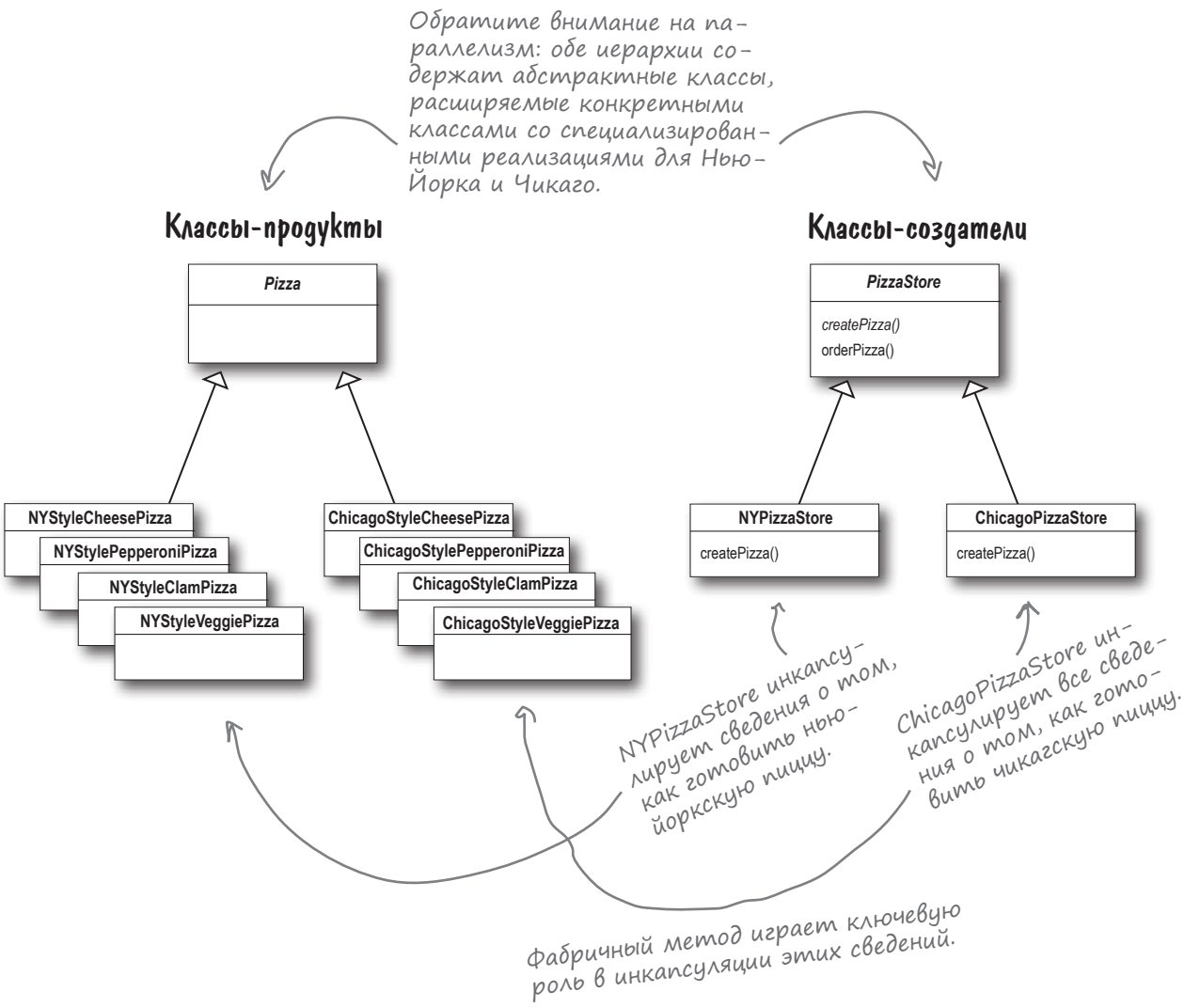
Классы-продукты



Другая точка зрения: параллельные иерархии классов

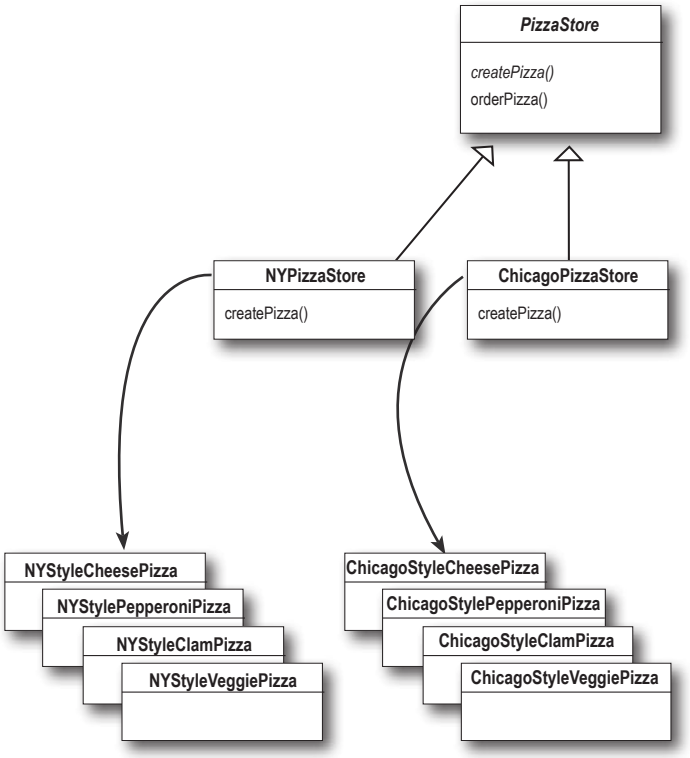
Мы уже видели, как метод orderPizza() объединяется с фабричным методом. Другая инфраструктурная точка зрения на данный паттерн связана со способом инкапсуляции информации о продукте в создателях.

Рассмотрим две параллельные иерархии классов:



Головоломка

Нам нужна еще одна разновидность пиццы для этих безумных калифорнийцев («безумных» в *хорошем* смысле слова, конечно). Нарисуйте еще один параллельный набор классов, необходимых для поддержки нового региона – Калифорнии – в PizzaStore.



Рисуйте здесь...

Теперь запишите пять самых *странных* ингредиентов для пиццы... И можете открывать бизнес по выпечке пиццы в Калифорнии!

Определение паттерна Фабричный Метод

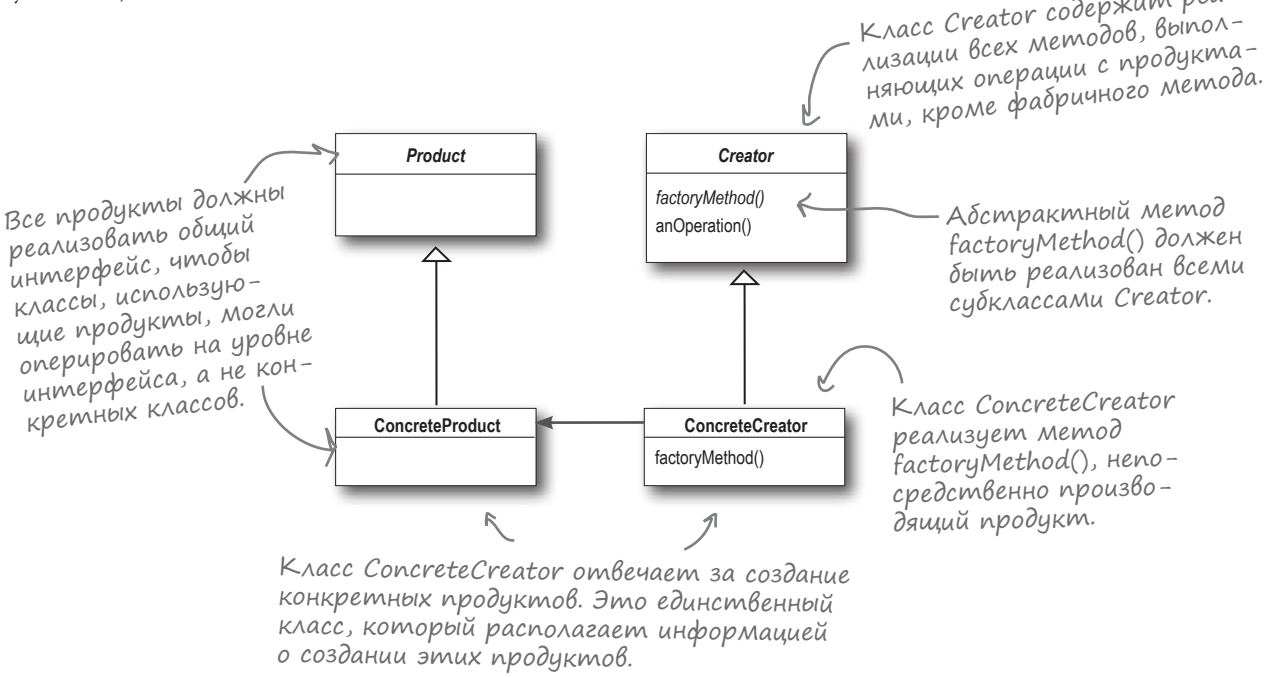
Пора привести официальное определение паттерна Фабричный Метод:

> **Паттерн Фабричный Метод** определяет интерфейс создания объекта, но позволяет субклассам выбрать класс создаваемого экземпляра. Таким образом, Фабричный Метод делегирует операцию создания экземпляра субклассам.

Паттерн Фабричный Метод, как все остальные разновидности фабрик, предоставляет способ инкапсуляции создания экземпляров конкретных типов. Так, из приведенной ниже диаграммы классов видно, что абстрактный класс Creator предоставляет интерфейс к методу создания объектов, также называемому «фабричным методом». Остальные методы, реализуемые в абстрактном классе Creator, работают с продуктами, созданными фабричным методом. Только субклассы фактически реализуют фабричный метод и создают продукты.

Не верите — спросите сами... Но сейчас вы уже разбираетесь в этой теме лучше, чем они!

Разработчики часто говорят, что Фабричный Метод позволяет субклассам выбрать тип создаваемого экземпляра. Имеется в виду не то, что паттерн позволяет субклассам принимать самостоятельное решение во время выполнения, а то, что класс-создатель не обладает информацией о фактическом типе создаваемых продуктов (последний определяется исключительно типом используемого субкласса).



Часто Задаваемые Вопросы

В: Зачем использовать Фабричный Метод при одном классе ConcreteCreator?

О: Паттерн пригодится и в том случае, если в вашей иерархии всего один конкретный создатель — он отделяет реализацию продукта от его использования. Если позднее добавятся другие продукты или изменится реализация продукта, это не отразится на работе класса-создателя.

В: Правильно ли сказать, что в реализации пиццерий для Нью-Йорка и Чикаго из нашего примера применяется паттерн Простая Фабрика?

О: Они похожи, но используются по-разному. Хотя реализация каждой конкретной пиццерии имеет много общего с SimplePizzaFactory, помните, что конкретные пиццерии расширяют класс, в котором createPizza() определяется как абстрактный метод. Каждая пиццерия сама определяет поведение метода createPizza(). В паттерне Простая Фабрика фабрика представляет собой объект, объединяемый с PizzaStore посредством композиции.

В: Фабричный метод и класс-создатель всегда должны быть абстрактными?

О: Нет, Фабричный Метод по умолчанию, может создавать некий конкретный продукт. Это позволит вам создавать продукты даже при отсутствии субклассов у класса-создателя.

В: Каждая пиццерия может выпускать до четырех видов пиццы в зависимости от полученного параметра. Все конкретные создатели обязательно производят несколько продуктов?

О: Мы реализовали *параметризованный* фабричный метод, который может производить разные объекты в зависимости от значения полученного параметра. Фабрика также может производить только один вид объектов; обе версии паттерна вполне допустимы.

В: Ваши параметризованные типы небезопасны. При вызове передается простая строка! А если вместо «ClamPizza» будет передана строка «CalmPizza»?

О: Безусловно, вы правы — произойдет ошибка времени выполнения. Существуют различные приемы, обеспечивающие возможность обнаружения ошибок на стадии компиляции: например, создание объектов, представляющих виды параметров, использование статических констант или *перечислений*.

В: Я плохо понимаю различия между Простой Фабрикой и Фабричным Методом. Они очень похожи, разве что в Фабричном Методе класс, возвращающий объекты, реализован в виде субкласса. Можете объяснить?

О: Субклассы действительно похожи на Простую Фабрику, но Простая Фабрика обладает узкой специализацией, а Фабричный Метод ведет к созданию инфраструктуры, в которой реализация выбирается субклассами. Например, метод orderPizza() в Фабричном Методе создает общую инфраструктуру для создания объектов. Субклассируя PizzaStore, вы выбираете состав пиццы, возвращаемой методом orderPizza(). Простая Фабрика инкапсулирует создание объектов, но она лишена гибкости Фабричного Метода в изменении создаваемых продуктов.

Учитель и Ученик...

Учитель: *Скажи, как идет твое обучение?*

Ученик: *Учитель, я продолжаю изучать тему «инкапсуляции переменных аспектов».*

Учитель: *Продолжай...*

Ученик: *Я узнал, что код создания объектов тоже можно инкапсулировать. Код, создающий экземпляры конкретных классов, подвержен частым изменениям. Я узнал о «фабриках», которые помогают инкапсулировать создание экземпляров.*

Учитель: *И какую пользу приносят эти «фабрики»?*

Ученик: *Большую, о Учитель. Размещение кода создания в одном объекте или методе позволяет избежать дублирования кода и упрощает сопровождение. Кроме того, клиент зависит только от интерфейсов, а не от конкретных классов, необходимых для создания объектов. Программирование на уровне интерфейса делает код более гибким и упрощает его возможное расширение.*

Учитель: *Воистину твои познания растут. Может, у тебя есть какие-нибудь вопросы?*

Ученик: *Учитель, я знаю, что инкапсуляция создания объектов улучшает степень абстракции и отделяет клиентский код от фактических реализаций. Но мой фабричный код все равно должен использовать конкретные классы. Нет ли в этом самообмана?*

Учитель: *Создание объектов — суровая действительность; без создания объектов не напишешь ни одной Java-программы. Но вооружившись этим знанием, мы изолируем код создания объектов — подобно тому, как мы изолируем овец в загоне, где о них можно заботиться и ухаживать. Если же разрешить овцам свободно бегать за оградой, мы никогда не сможем настричь с них шерсти.*

Ученик: *Учитель, я вижу в этом свет Истины.*

Учитель: *Я не сомневался. А теперь иди и медитируй на зависимостях между объектами.*

PizzaStore с сильными зависимостями

Возьми в руку карандаш

Представьте, что вы никогда не слышали о ОО-фабриках. Ниже приведена версия PizzaStore, в которой фабрики не используются; подсчитайте количество конкретных объектов пиццы, от которых зависит этот класс. А если добавить пиццу в калифорнийском стиле, от скольких объектов он будет зависеть тогда?

```
public class DependentPizzaStore {

    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new NYStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new ChicagoStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new ChicagoStylePepperoniPizza();
            }
        } else {
            System.out.println("Error: invalid type of pizza");
            return null;
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

Все пиццы в нью-йоркском стиле.

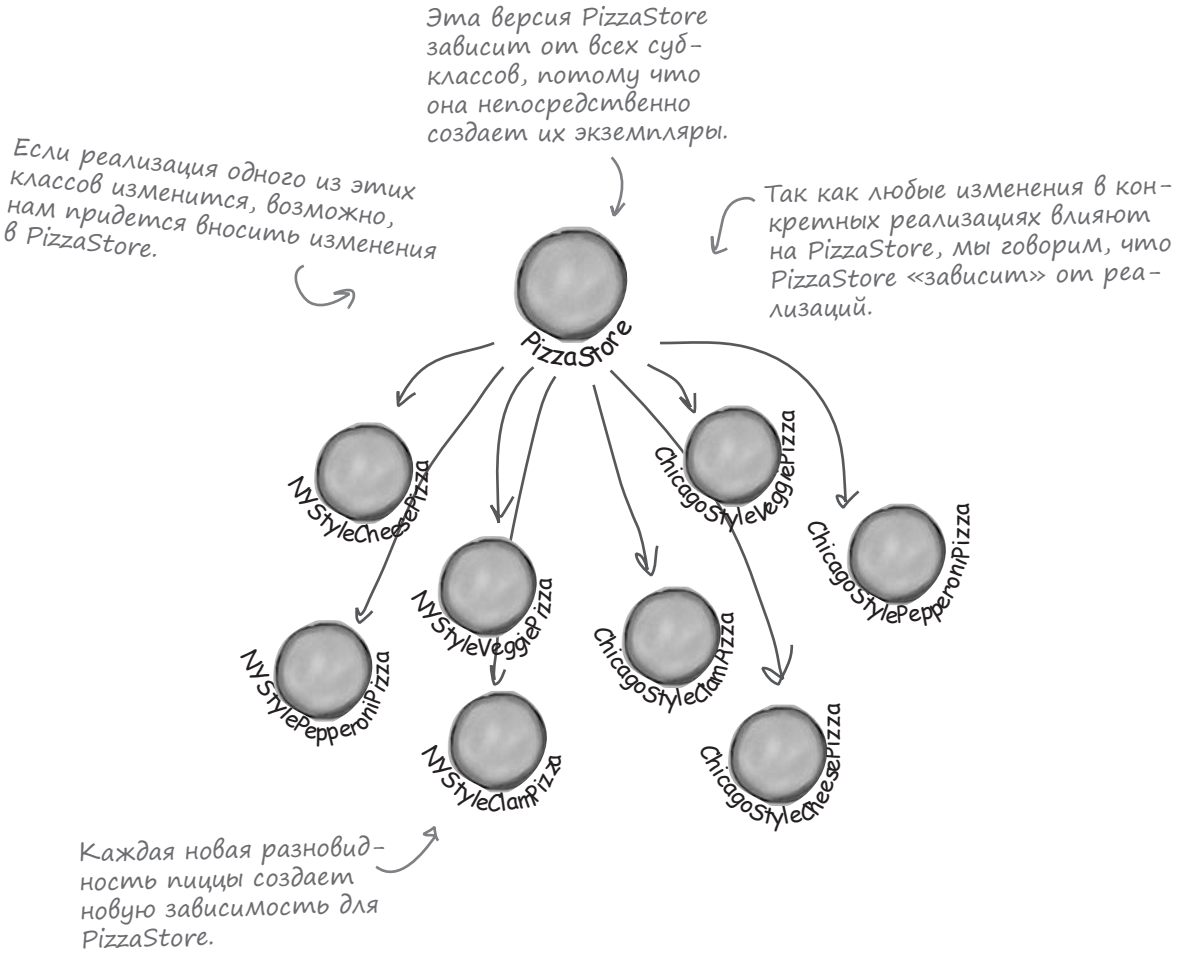
Все пиццы в чикагском стиле.

Впишите свои ответы: _____ число _____ с калифорнийской пиццей

Зависимости между объектами

При непосредственном создании экземпляра объекта возникает зависимость от его конкретного класса. Взгляните на реализацию PizzaStore с предыдущей страницы — все объекты пиццы создаются самим классом PizzaStore.

Если нарисовать диаграмму, изображающую эту версию PizzaStore и все объекты, от которых она зависит, результат будет выглядеть примерно так:



Принцип инверсии зависимостей

Вполне очевидно, что сокращение зависимостей от конкретных классов — явление положительное. Более того, этот факт закреплен в одном из принципов ОО-проектирования с красивым, впечатляющим названием *принцип инверсии зависимостей*.

Еще одна фраза, которая наверняка произведет впечатление на начальство!

Он формулируется так:

Принцип проектирования

Код должен зависеть от абстракций, а не от конкретных классов.

На первый взгляд этот принцип сильно напоминает принцип «Программируйте на уровне интерфейсов, а не на уровне реализаций», верно? Да, они похожи; однако принцип инверсии зависимостей предъявляет еще более жесткие требования к абстракции. Он требует, чтобы высокоуровневые компоненты не зависели от низкоуровневых компонентов; вместо этого *и те, и другие* должны зависеть от абстракций.

«Высокоуровневым» компонентом называется класс, поведение которого определяется в контексте других, «низкоуровневых» компонентов.

Например, PizzaStore является высокоуровневым компонентом, потому что он работает с разными объектами пиццы — готовит их, выпекает, нарезает и упаковывает. Объекты пиццы при этом являются низкоуровневыми компонентами.

Но что это значит?

Давайте еще раз взглянем на диаграмму с предыдущей страницы. PizzaStore — наш «высокоуровневый» компонент, а реализации пицц — «низкоуровневые» компоненты. Очевидно, PizzaStore зависит от конкретных классов, представляющих виды пиццы.

Принцип указывает, что в своем коде мы должны зависеть от абстракций, а не от конкретных классов. Это относится как к высокоуровневым, так и к низкоуровневым модулям.

Но как это сделать? Давайте подумаем, как применить этот принцип к Сильно Зависимой реализации PizzaStore...

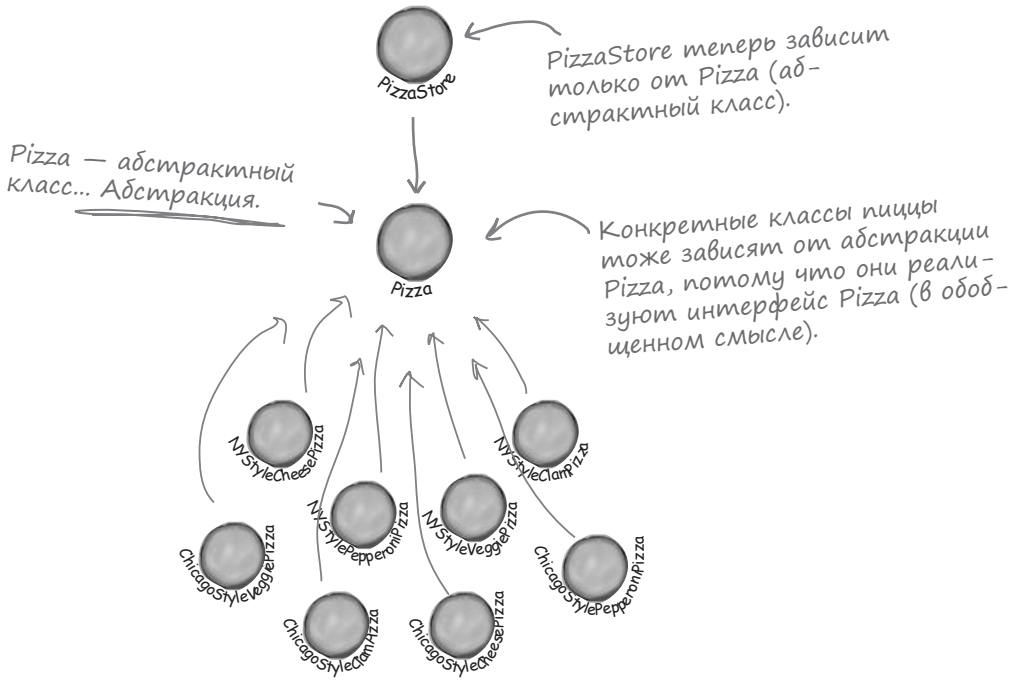
Применение принципа

Главный недостаток Сильно Зависимой версии PizzaStore заключается в том, что она зависит от всех классов пиццы из-за явного создания экземпляров конкретных типов в методе orderPizza().

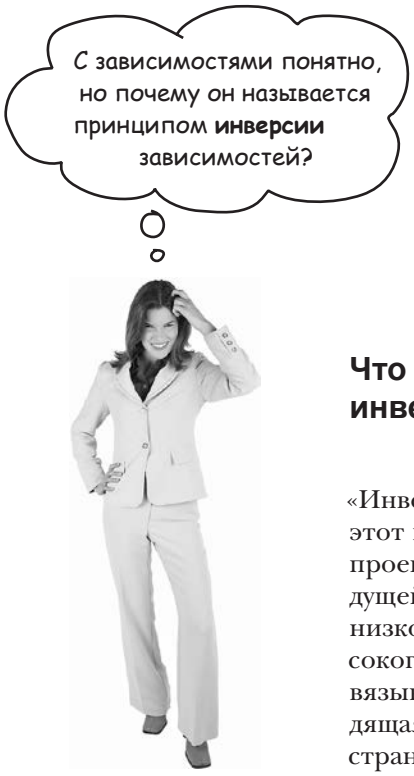
Хотя мы и создали абстракцию Pizza, в коде кодируются конкретные типы пиццы, поэтому абстракция особой пользы не приносит.

Как вынести создание экземпляров из метода orderPizza()? Для этого и нужен Фабричный Метод.

После применения Фабричного Метода диаграмма выглядит так:



Нетрудно заметить, что после применения Фабричного Метода высокоуровневый компонент PizzaStore и низкоуровневые компоненты зависят от Pizza, то есть от абстракции. Фабричныый Метод — не единственный, но один из самых мощных приемов, обеспечивающих соблюдение принципа инверсии зависимостей.

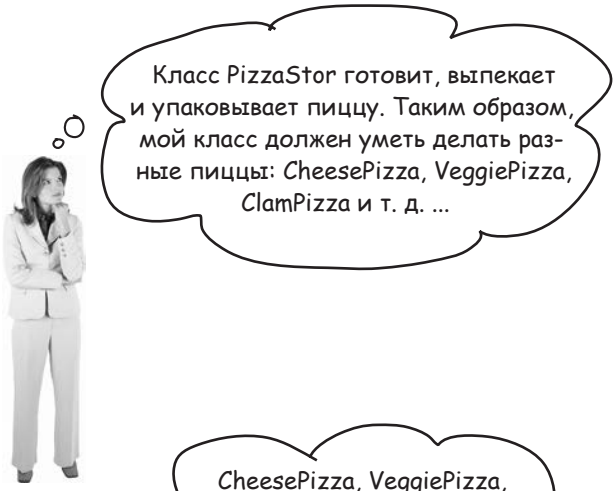


Что именно инвертируется в принципе инверсии зависимостей?

«Инверсия» в названии принципа объясняется тем, что этот принцип инвертирует традиционный подход к ОО-проектированию. Взгляните на диаграмму на предыдущей странице и обратите внимание на зависимость низкоуровневых компонентов от абстракции более высокого уровня. Высокоуровневый компонент тоже привязывается к той же абстракции. Таким образом, нисходящая диаграмма зависимостей, нарисованная нами пару страниц назад, инвертировалась — и высокоуровневые, и низкоуровневые модули теперь зависят от абстракции.

А теперь проанализируем типичный подход к процессу проектирования и посмотрим, как этот принцип влияет на мышление проектировщика...

Инверсия мышления...



Вы занимаетесь реализацией PizzaStore. Какая мысль первой приходит вам в голову?

Верно, вы начинаете с верха иерархии, и опускаетесь к конкретным классам. Но, как мы уже знаем, пиццерия не должна располагать информацией о конкретных типах пиццы, потому что это создает зависимость от конкретных классов!

А теперь «инвертируем» направление мысли... Вместо того, чтобы начинать сверху, начнем с разных конкретных видов пиццы и подумаем, что из них можно абстрагировать.

Верно! Мы приходим к абстракции *Pizza*. А теперь можно вернуться к проектированию PizzaStore.

Но для этого нам придется воспользоваться фабрикой, чтобы исключить конкретные классы из PizzaStore. После этого разные конкретные типы пиццы будут зависеть только от абстракции, как и класс пиццерии. Мы взяли архитектуру, в которой работа пиццерии зависит от конкретных классов, и инвертировали эти зависимости (вместе с направлением мышления).

Несколько советов по применению принципа...

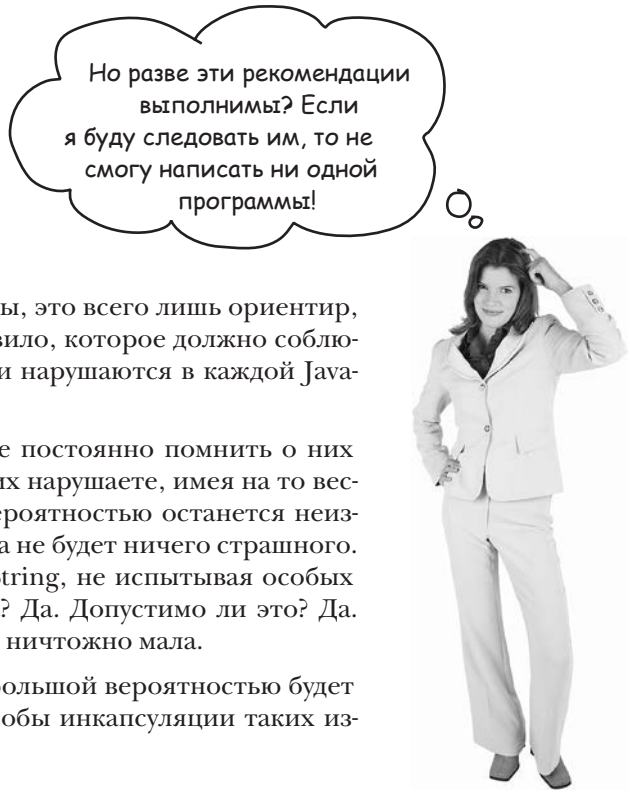
Следующие рекомендации помогут вам избежать нарушения принципа инверсии зависимостей в своих архитектурах:

При использовании new сохраняется ссылка на конкретный класс. Используйте фабрику!

- Ссылки на конкретные классы не должны храниться в переменных.
- В архитектуре не должно быть классов, производных от конкретных классов.
- Методы не должны переопределять методы, реализованные в каких-либо из его базовых классов.

Наследование от конкретного класса создает зависимость от него. Определяйте классы производными от абстракций — интерфейсов или абстрактных классов.

Если вы переопределяете реализованный метод, значит, базовый класс был плохой абстракцией. Методы, реализованные в базовом классе, должны использоваться всеми субклассами.



Вы абсолютно правы! Как и многие наши принципы, это всего лишь ориентир, к которому следует стремиться, а не железное правило, которое должно соблюдаться постоянно. Понятно, что эти рекомендации нарушаются в каждой Java-программе!

Но если вы запомните эти рекомендации и будете постоянно помнить о них в ходе проектирования, вы будете знать, когда вы их нарушаете, имея на то веские причины. Например, если класс с большой вероятностью останется неизменным, в создании экземпляра конкретного класса не будет ничего страшного. В конце концов, мы постоянно создаем объекты String, не испытывая особых опасений. Является ли это нарушением принципа? Да. Допустимо ли это? Да. Почему? Потому что вероятность изменения String ничтожно мала.

С другой стороны, если написанный вами класс с большой вероятностью будет изменяться, в вашем распоряжении хорошие способы инкапсуляции таких изменений, например Фабричный Метод.

Вернемся в пиццерию...

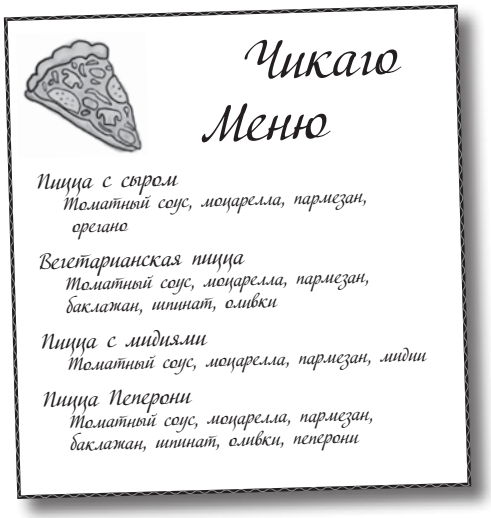
Архитектура PizzaStore постепенно формируется: мы создаем гибкую инфраструктуру, которая хорошо соответствует принципам проектирования.

Ключом к успеху вашей пиццерии всегда были свежие, качественные ингредиенты. Однако вы обнаружили, что новые пиццерии следуют всем *процедурам*, но некоторые из них используют второсортные ингредиенты для снижения затрат и повышения прибыли. С этим надо что-то делать, потому что в долгосрочной перспективе такая «экономия» только повредит бренду!

Единые стандарты качества ингредиентов

Как обеспечить использование только качественных ингредиентов? Мы создадим фабрику, которая производит их и поставляет вашим пиццериям!

У этой идеи только один недостаток: пиццерии находятся в разных географических регионах, и то, что называется «томатным соусом» в Нью-Йорке, в Чикаго называется совершенно иначе. Таким образом, в Нью-Йорк будет поставляться один набор ингредиентов, а в Чикаго — совершенно другой. Давайте присмотримся поближе:

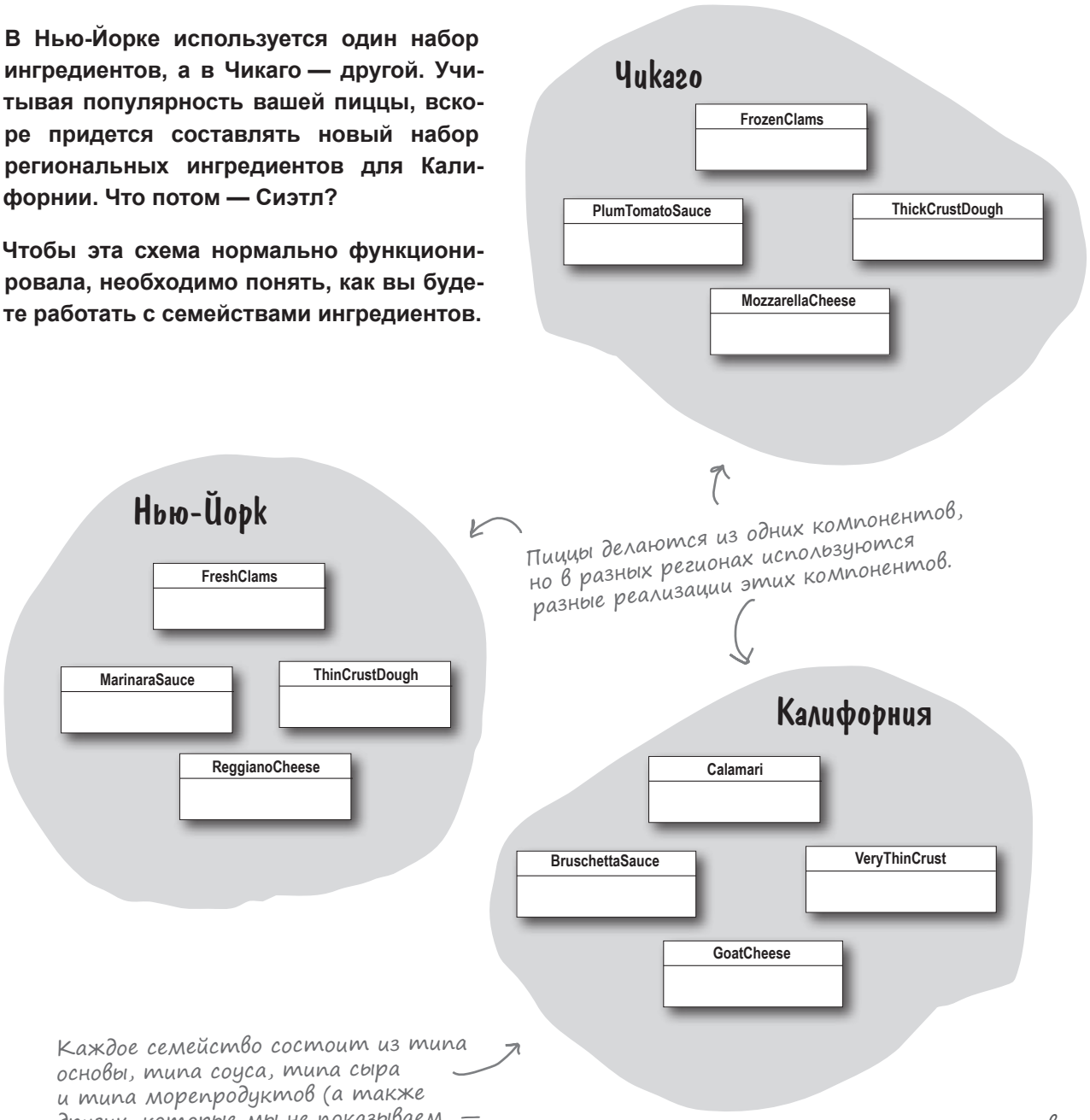


Одни и те же наборы продуктов (основа, соус, сыр, овощи, мясо), но с разными реализациями в зависимости от региона.

Семейства ингредиентов

В Нью-Йорке используется один набор ингредиентов, а в Чикаго — другой. Учитывая популярность вашей пиццы, вскоре придется составлять новый набор региональных ингредиентов для Калифорнии. Что потом — Сиэтл?

Чтобы эта схема нормально функционировала, необходимо понять, как вы будете работать с семействами ингредиентов.



Построение фабрик ингредиентов

Мы собираемся построить фабрику для создания ингредиентов; фабрика будет нести ответственность за создание каждого ингредиента. Другими словами, фабрика будет создавать основу, соус, сыр и т. д. Вскоре вы увидите, как мы будем решать проблему региональных различий.

Начнем с определения интерфейса фабрики, которая будет создавать все наши ингредиенты:

```
public interface PizzaIngredientFactory {

    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();

}
```

Для каждого ингредиента в интерфейсе определяется метод create.

Множество новых классов, по одному на каждый ингредиент.

Если бы в каждом экземпляре фабрики должны были присутствовать некие общие «механизмы», также можно было бы создать абстрактный класс...

Вот что мы собираемся сделать:

1. Построить фабрику для каждого региона. Для этого мы создадим субкласс PizzaIngredientFactory, реализующий все методы create.

2. Реализовать набор классов ингредиентов, которые будут использоваться с фабрикой, – ReggianoCheese, RedPeppers, ThickCrustDough и т. д. Там, где это возможно, эти классы могут использоваться совместно несколькими регионами.

3. Затем все новые классы необходимо связать воедино. Для этого мы определим новые фабрики ингредиентов в коде PizzaStore.

Построение фабрики ингредиентов для Нью-Йорка

Перед вами реализация фабрики ингредиентов для Нью-Йорка. Эта фабрика специализируется на соусе «маринара», сыре «реджиано» и свежих мидиях...

Нью-Йоркская фабрика ингредиентов реализует общий интерфейс всех фабрик ингредиентов

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {

    public Dough createDough() {
        return new ThinCrustDough();
    }

    public Sauce createSauce() {
        return new MarinaraSauce();
    }

    public Cheese createCheese() {
        return new ReggianoCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new Garlic(), new Onion(), new Mushroom(), new RedPepper() };
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }
    public Clams createClam() {
        return new FreshClams();
    }
}
```

Для каждого ингредиента в семействе создается его версия для Нью-Йорка.

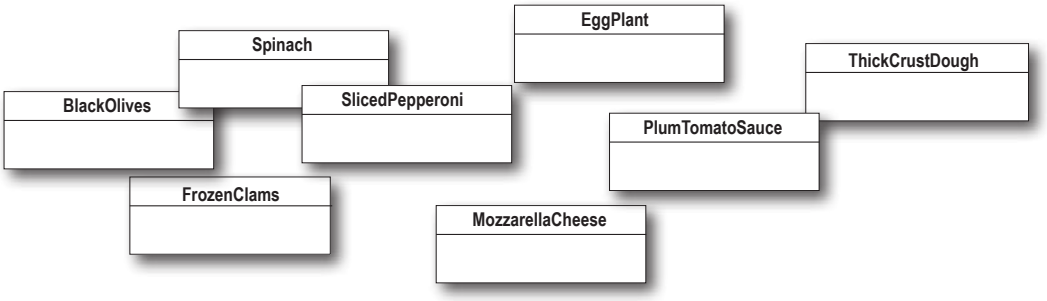
Содержимое массива жестко фиксировано. Возможны и менее тривиальные решения, но они не имеют отношения к изучению паттерна, поэтому был выбран простой вариант.

Нарезанные пеперони используются и в Нью-Йорке, и в Чикаго.

Нью-Йорк находится на побережье; в нем используются свежие мидии. Чикаго придется довольствоваться мороженными.

Возьми в руку карандаш

Напишите фабрику ингредиентов ChicagoPizzaIngredientFactory. Используйте в своей реализации классы, приведенные ниже:



Перерабатываем классы пиццы...

Все наши фабрики запущены и готовы к выпуску качественных ингредиентов; осталось переработать классы пиццы, чтобы они использовали только ингредиенты, произведенные фабриками. Начнем с абстрактного класса Pizza:

```
public abstract class Pizza {
    String name;
    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;

    abstract void prepare();

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }

    void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }

    void setName(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }

    public String toString() {
        // Код вывода описания пиццы
    }
}
```

Каждый объект пиццы содержит набор ингредиентов, используемых при ее приготовлении.

Метод prepare стал абстрактным. В нем мы будем собирать ингредиенты, необходимые для приготовления пиццы. Которые, разумеется, будут производиться фабрикой ингредиентов.

Другие методы остаются неизменными (кроме prepare).

Переработка классов пиццы продолжается...

Теперь, когда у нас имеется абстрактный класс Pizza, можно переходить к созданию классов нью-йоркской и чикагской пиццы — только на этот раз они будут получать свои ингредиенты прямо с фабрики. Времена, когда пиццерии могли экономить на качестве, прошли!

При написании кода Фабричного Метода у нас были классы NYCheesePizza и ChicagoCheesePizza. Присмотревшись к этим классам, мы видим, что они различаются только использованием региональных ингредиентов, а сама пицца остается неизменной (основа + соус + сыр). То же относится и к другим пиццам: вегетарианской, с мидиями и т. д. Все они готовятся по единой схеме, но с разными ингредиентами.

Итак, на самом деле нам не нужны два класса для каждой пиццы; фабрика ингредиентов справится с региональными различиями за нас. Например, класс пиццы с сыром выглядит так:

В ходе приготовления пиццы нам понадобится фабрика, поставляющая ингредиенты. Соответственно, конструктору каждого класса пиццы передается объект фабрики, ссылка на который сохраняется в переменной экземпляра.

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

Самое главное!

Метод prepare() готовит пиццу с сыром. Когда ему требуется очередной ингредиент, он запрашивает его у фабрики.

Код под увеличительным стеклом

Код Pizza использует фабрику для производства ингредиентов, используемых в пицце. Производимые ингредиенты определяются фабрикой. Для класса Pizza различия несущественны; он умеет готовить пиццу из обобщенных ингредиентов. Он изолирован от различий в региональных ингредиентах, и мы можем легко использовать его с фабриками для любых других регионов.

```
sauce = ingredientFactory.createSauce();
```

В переменной экземпляра сохраняется ссылка на конкретный соус данной пиццы.

Класс Pizza может использовать любую фабрику ингредиентов.

Метод createSauce() возвращает соус, используемый в данном регионе. Скажем, для Нью-Йорка это будет соус «маринара».

Также рассмотрим класс ClamPizza:

```
public class ClamPizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public ClamPizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
        clam = ingredientFactory.createClam();
    }
}
```

ClamPizza тоже сохраняет фабрику ингредиентов.

Чтобы создать пиццу с мидиями, метод prepare получает ингредиенты от локальной фабрики.

Если это нью-йоркская фабрика, мидии будут свежие, а если чикагская — мороженые.

Возвращаемся к пиццериям

Работа почти закончена; остается вернуться к классам пиццерий и позаботиться о том, чтобы они использовали правильные объекты Pizza. Также необходимо передать им ссылку на региональную фабрику ингредиентов:

```
public class NYPizzaStore extends PizzaStore {

    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
            new NYPizzaIngredientFactory();

        if (item.equals("cheese")) {

            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("New York Style Cheese Pizza");

        } else if (item.equals("veggie")) {

            pizza = new VeggiePizza(ingredientFactory);
            pizza.setName("New York Style Veggie Pizza");

        } else if (item.equals("clam")) {

            pizza = new ClamPizza(ingredientFactory);
            pizza.setName("New York Style Clam Pizza");

        } else if (item.equals("pepperoni")) {
            pizza = new PepperoniPizza(ingredientFactory);
            pizza.setName("New York Style Pepperoni Pizza");

        }
        return pizza;
    }
}
```

Фабрика производит ингредиенты для всех пицц в нью-йоркском стиле.

Теперь при создании каждой пиццы задается фабрика, которая должна использоваться для производства ее ингредиентов.

Вернитесь к предыдущей странице и обязательно разберитесь в том, как взаимодействуют классы пиццы и фабрики!

Для каждого вида пиццы мы создаем новый экземпляр Pizza и передаем ему фабрику, необходимую для производства ингредиентов.

МОЗГОВОЙ ШТУРМ

Сравните новую версию метода createPizza() с версией из реализации Фабричного Метода, приведенной ранее в этой главе.

Чего мы добились?

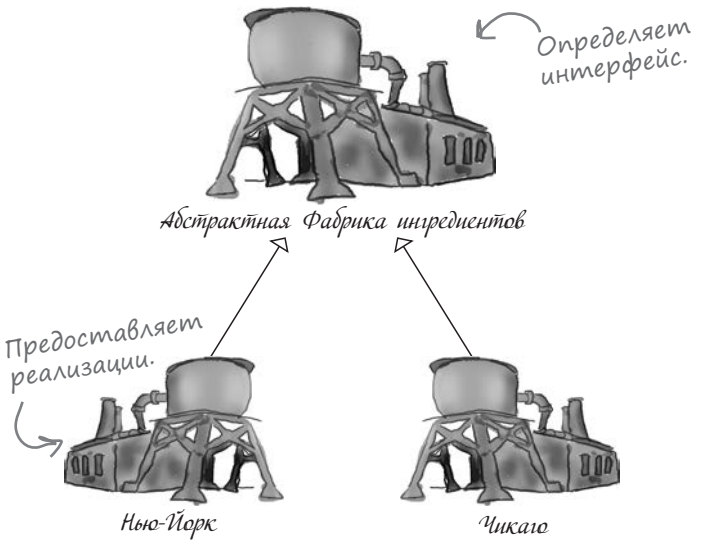
Мы внесли в программный код серию последовательных изменений; чего именно мы добились?

Мы реализовали механизм создания семейств ингредиентов для пиццы; для этого был введен новый тип фабрики, называемый Абстрактной Фабрикой.

Абстрактная Фабрика определяет интерфейс создания семейства продуктов. Использование этого интерфейса отделяет код от фабрики, создающей продукты. Таким образом создаются разнообразные фабрики, производящие продукты для разных контекстов: разных регионов, операционных систем или вариантов оформления.

Отделение кода от фактически используемых продуктов позволяет динамически переключать фабрики для изменения поведения (например, замены томатного соуса соусом «маринара»).

Абстрактная Фабрика определяет интерфейс для семейства продуктов. Что такое «семейство»? В нашем примере это ингредиенты для приготовления пиццы: основа, соус, сыр и т. д.



На основе Абстрактной Фабрики создаются одна или несколько конкретных фабрик, производящих одинаковые продукты, но с разными реализациями.

Затем мы пишем свой код так, чтобы для создания продуктов в нем использовались фабрики. Передавая разные фабрики, мы получаем разные реализации продуктов, но клиентский код при этом остается неизменным.

Итан и Джоэл требуют продолжения...

За сценой 

Итан и Джоэл в восторге от пиццы из Объектвиля! Но им и невдомек, что при выполнении их заказов используются новые фабрики ингредиентов. Вот, что происходит, когда они делают заказ...

Первая стадия обработки заказа не изменилась. Давайте еще раз проследим за выполнением заказа Итана:

1 **Сначала создается объект NYPizzaStore:**

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

Создание экземпляра NYPizzaStore.

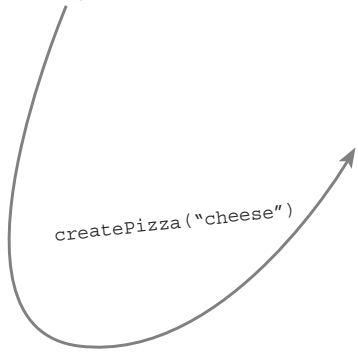
2 **Пиццерия построена, можно принимать заказ:**

```
nyPizzaStore.orderPizza("cheese");
```

Метод orderPizza() вызывается экземпляром nyPizzaStore

3 **Метод orderPizza() сначала вызывает метод createPizza():**

```
Pizza pizza  = createPizza("cheese");
```



Дальше процедура изменяется, потому что мы используем фабрику ингредиентов

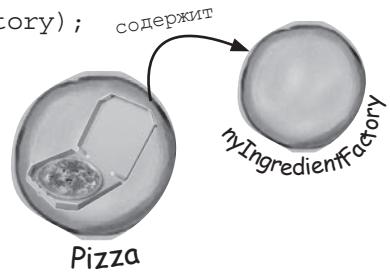
За сценой

4 **Вызывается метод createPizza(). Именно здесь вступает в дело наша фабрика ингредиентов:**

Фабрика ингредиентов выбирается и создается в PizzaStore, а затем передается конструктору каждой пиццы.

```
Pizza pizza = new CheesePizza(nyIngredientFactory);
```

Создает экземпляр Pizza, содержащий фабрику ингредиентов для Нью-Йорка.



5 **Теперь пиццу необходимо приготовить. При вызове метода prepare() фабрика получает запрос на производство ингредиентов:**

prepare()

```
void prepare() {
    dough = factory.createDough();
    sauce = factory.createSauce();
    cheese = factory.createCheese();
}
```

Тонкая основа

Маринара

Реджиано

Для пиццы Итана используется нью-йоркская фабрика ингредиентов.

6 **Пицца подготовлена. Метод orderPizza() выпекает, нарезает и упаковывает ее.**

Определение паттерна Абстрактная Фабрика

В нашу коллекцию паттернов добавляется очередной фабричный паттерн, предназначенный для создания семейств продуктов. Давайте обратимся к официальному определению этого паттерна:

> **Паттерн Абстрактная Фабрика** предоставляет интерфейс создания семейств взаимосвязанных или взаимозависимых объектов без указания их конкретных классов.

Как было показано ранее, благодаря паттерну Абстрактная Фабрика клиент может использовать абстрактный интерфейс для создания логически связанных продуктов, не располагая информацией о конкретных создаваемых продуктах. Таким образом, клиент отделяется от подробностей конкретного продукта. Следующая диаграмма классов показывает, как работает эта схема:

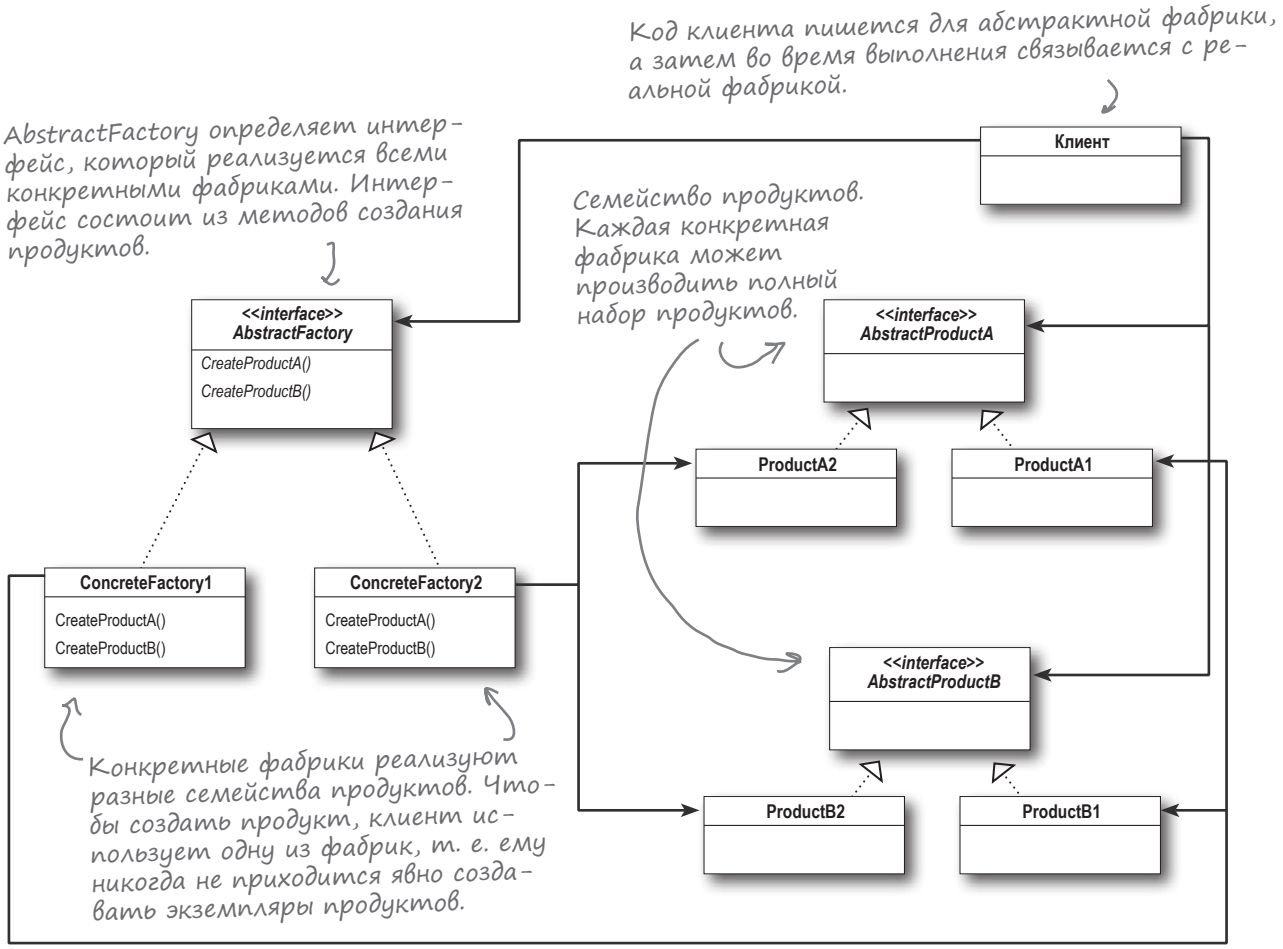
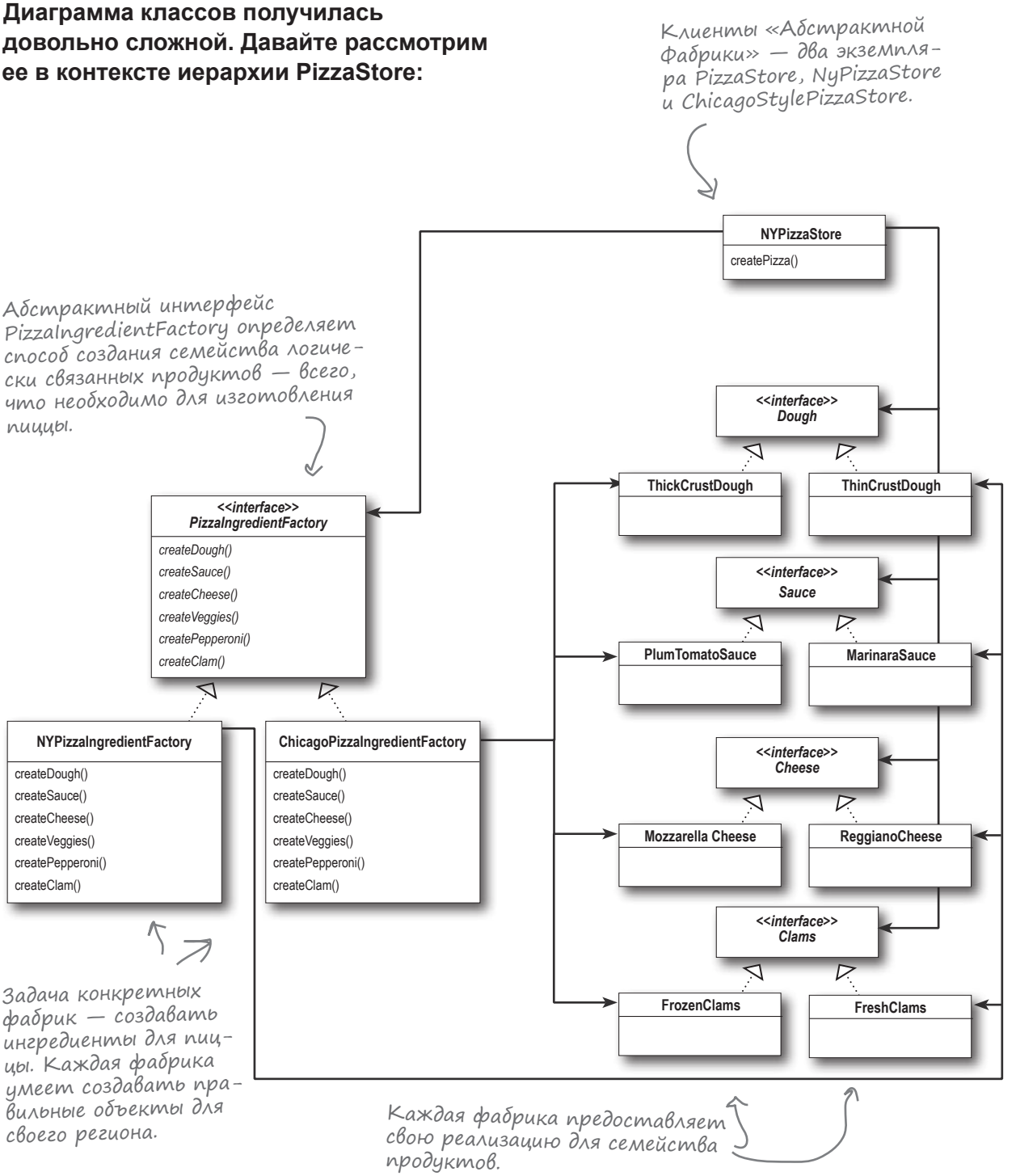


Диаграмма классов получилась довольно сложной. Давайте рассмотрим ее в контексте иерархии PizzaStore:



В Абстрактной Фабрике прячется Фабричный Метод?

Хорошее наблюдение! Да, методы Абстрактной Фабрики часто реализуются как фабричные методы. И это вполне логично, не правда ли? Задача Абстрактной Фабрики – определить интерфейс для создания набора продуктов. Каждый метод этого интерфейса отвечает за создание конкретного продукта, и мы реализуем субкласс Абстрактной Фабрики, который предоставляет эти реализации. Таким образом, фабричные методы являются естественным способом реализации методов продуктов в абстрактных фабриках.

ПАТТЕРНЫ ДЛЯ ВСЕХ

Интервью недели:
Фабричный Метод и Абстрактная Фабрика друг о друге

HeadFirst: Ого, интервью с двумя паттернами сразу! Такое у нас впервые.

Фабричный Метод: Да, хотя я не уверен, что мне нравится, когда меня смешивают с Абстрактной Фабрикой. Да, мы оба являемся фабричными паттернами, но это не значит, что мы не заслужили отдельных интервью.

HeadFirst: Не переживайте, мы хотели побеседовать с вами одновременно, чтобы наши читатели раз и навсегда поняли, кто есть кто. Между вами существует некоторое сходство, и люди вас иногда путают.

Абстрактная Фабрика: Да, это правда. Мы оба предназначены для отделения приложений от конкретики реализаций, но делаем это по-разному. Так что я могу понять, почему нас путают.

Фабричный Метод: И все равно меня это раздражает. В конце концов, для создания я использую классы, а Абстрактная Фабрика – объекты. Ничего общего!

HeadFirst: А можно чуть подробнее?

Фабричный Метод: Конечно. И я, и Абстрактная Фабрика создаем объекты — это наша работа. Но я использую наследование...

Абстрактная Фабрика: ...а я — композицию.

Фабричный Метод: Точно. Значит, для создания объектов Фабричным Методом необходимо расширить класс и переопределить фабричный метод.

HeadFirst: И что делает этот фабричный метод?

Фабричный Метод: Создает объекты, конечно! То есть вся суть паттерна Фабричный Метод заключается в использовании субкласса, который создает объекты за вас. Клиенту достаточно знать абстрактный тип, который они используют, а субкласс имеет дело с конкретными типами. Другими словами, я отделяю клиентов от конкретных типов.

Абстрактная Фабрика: И я тоже, но по-другому.

HeadFirst: Слушаем Абстрактную Фабрику... Вы что-то говорили о композиции?

Абстрактная Фабрика: Я предоставляю абстрактный тип для создания семейств продуктов. Субклассы этого типа определяют способ создания продуктов. Чтобы использовать фабрику, вы создаете экземпляр и передаете его коду, написанному для абстрактного типа. Таким образом, как и с Фабричным Методом, мои клиенты отделяются от конкретных продуктов.

HeadFirst: Выходит, другое преимущество заключается в группировке связанных продуктов.

Абстрактная Фабрика: Вот именно.

HeadFirst: А что происходит при расширении набора этих продуктов — скажем, при добавлении? Разве оно не требует изменения интерфейса?

Абстрактная Фабрика: Верно, мой интерфейс изменяется при добавлении новых продуктов, но...

Фабричный Метод: (*презрительно фыркает*)

Абстрактная Фабрика: Как это понимать?

Фабричный Метод: Да ладно, это серьезная проблема! При изменении интерфейса приходится изменять интерфейс каждого субкласса!

Абстрактная Фабрика: Да, но меня используют для создания целых семейств продуктов, поэтому мне нужен большой интерфейс. Ты создаешь только один продукт, поэтому тебе хватает одного метода.

HeadFirst: Говорят, вы часто используете фабричные методы для реализации конкретных фабрик?

Абстрактная Фабрика: Да, не отрицаю, мои конкретные фабрики часто реализуют фабричный метод для создания своих продуктов. В моем случае они используются только для создания продуктов...

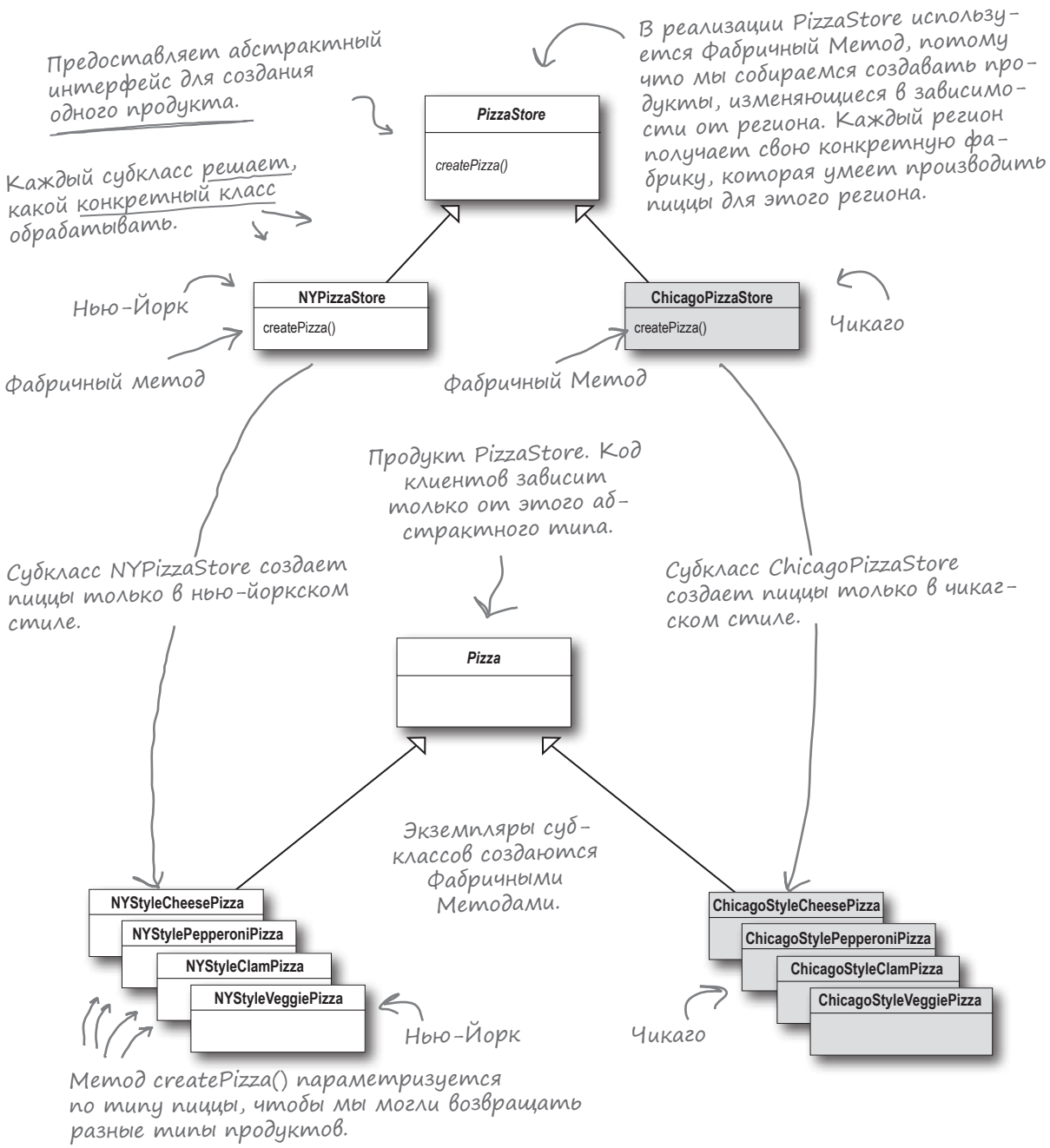
Фабричный Метод: ...а я обычно реализую в абстрактном создателе код, который использует конкретные типы, созданные субклассами.

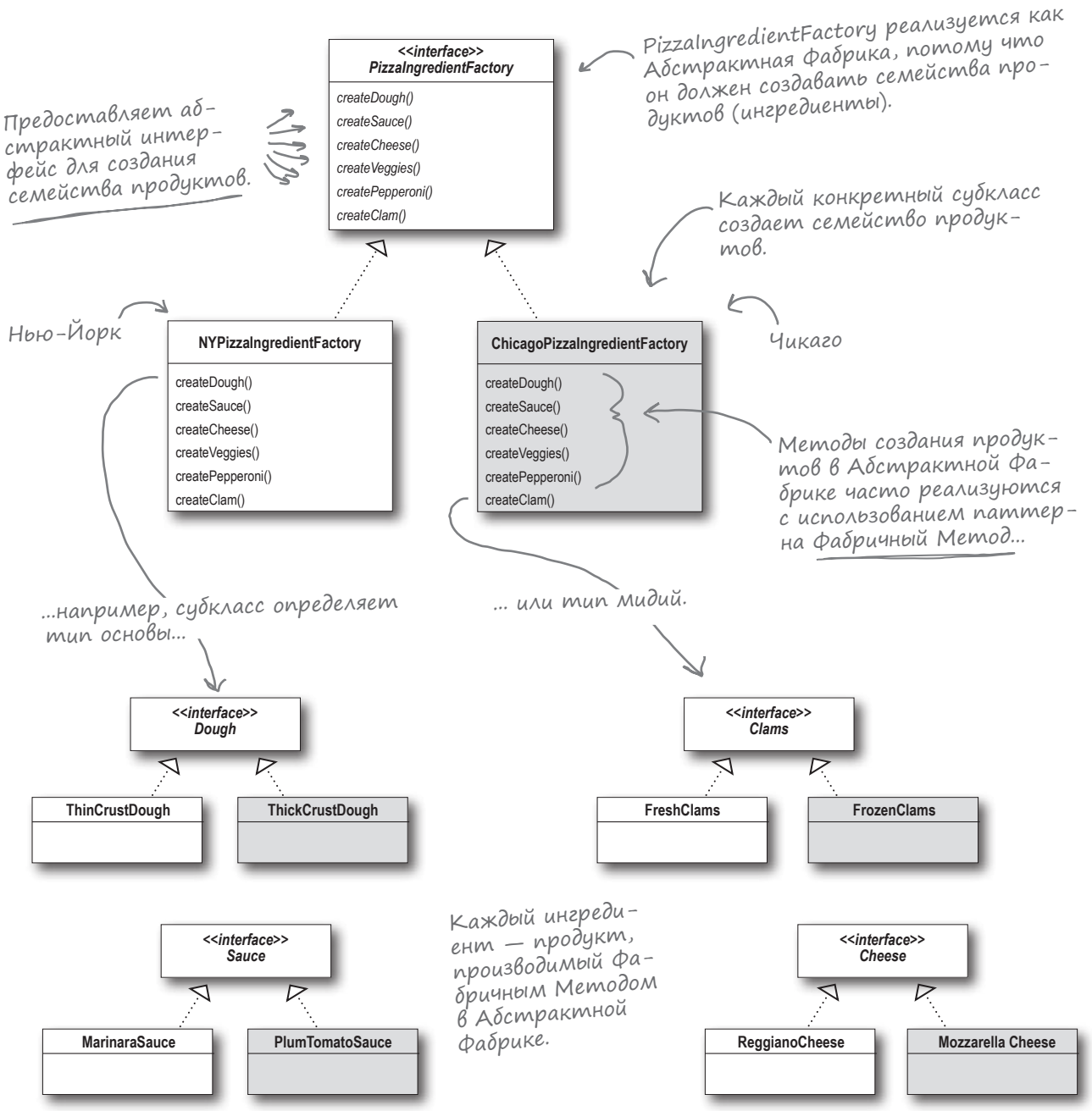
HeadFirst: Похоже, вы оба отлично справляетесь со своим делом. Вы оба инкапсулируете создание объектов, чтобы обеспечить слабую связанность приложений и снизить зависимость от реализаций, а это всегда хорошо независимо от выбора паттерна. Пара слов на прощание?

Абстрактная Фабрика: Спасибо. Используйте меня при создании семейств продуктов, когда вы должны обеспечить логическую согласованность создаваемых объектов.

Фабричный Метод: А я буду полезен для отделения клиентского кода от создания экземпляров конкретных классов и в тех ситуациях, когда точный состав всех конкретных классов неизвестен заранее. Субклассируйте меня и реализуйте мой фабричный метод!

Сравнение паттернов Фабричный Метод и Абстрактная Фабрика

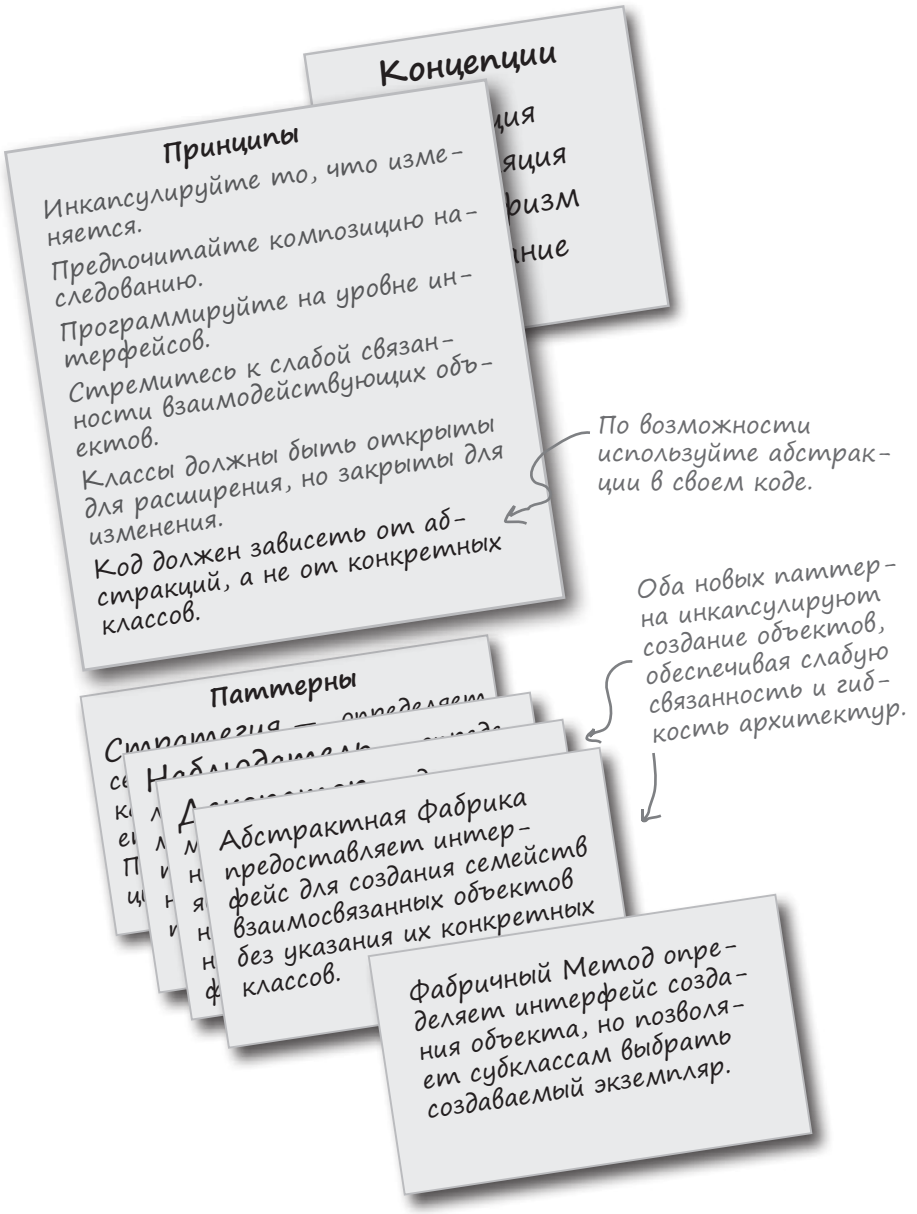




Субклассы продуктов образуют параллельные наборы семейств. В данном примере это семейства ингредиентов для Нью-Йорка и Чикаго.

Новые инструменты

В этой главе в вашем ОО-инструментарии появились два новых инструмента: Фабричный Метод и Абстрактная Фабрика. Оба паттерна инкапсулируют создание объектов и помогают изолировать код от операций с конкретными типами.



КЛЮЧЕВЫЕ МОМЕНТЫ

- Все фабрики инкапсулируют создание объектов.
- Простая Фабрика, хотя и не является полноценным паттерном, обеспечивает простой механизм изоляции клиентов от конкретных классов.
- Фабричный Метод основан на наследовании: создание объектов делегируется субклассам, реализующим фабричный метод для создания объектов.
- Абстрактная Фабрика основана на композиции: создание объектов реализуется в методе, доступ к которому осуществляется через интерфейс фабрики.
- Все фабричные паттерны обеспечивают слабую связанность за счет сокращения зависимости приложения от конкретных классов.
- Задача Фабричного Метода — перемещение создания экземпляров в субклассы.
- Задача Абстрактной Фабрики — создание семейств взаимосвязанных объектов без зависимости от их конкретных классов.

Возьми в руку карандаш
Решение

Класс NYPizzaStore уже готов; еще два класса, и бизнес можно будет запускать! Запишите в этой врезке реализации PizzaStore для Чикаго и Калифорнии:

Обе пиццерии почти не отличаются от нью-йоркской... Просто они создают другие виды пиццы.

```
public class ChicagoPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new ChicagoStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new ChicagoStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new ChicagoStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new ChicagoStylePepperoniPizza();
        } else return null;
    }
}
```

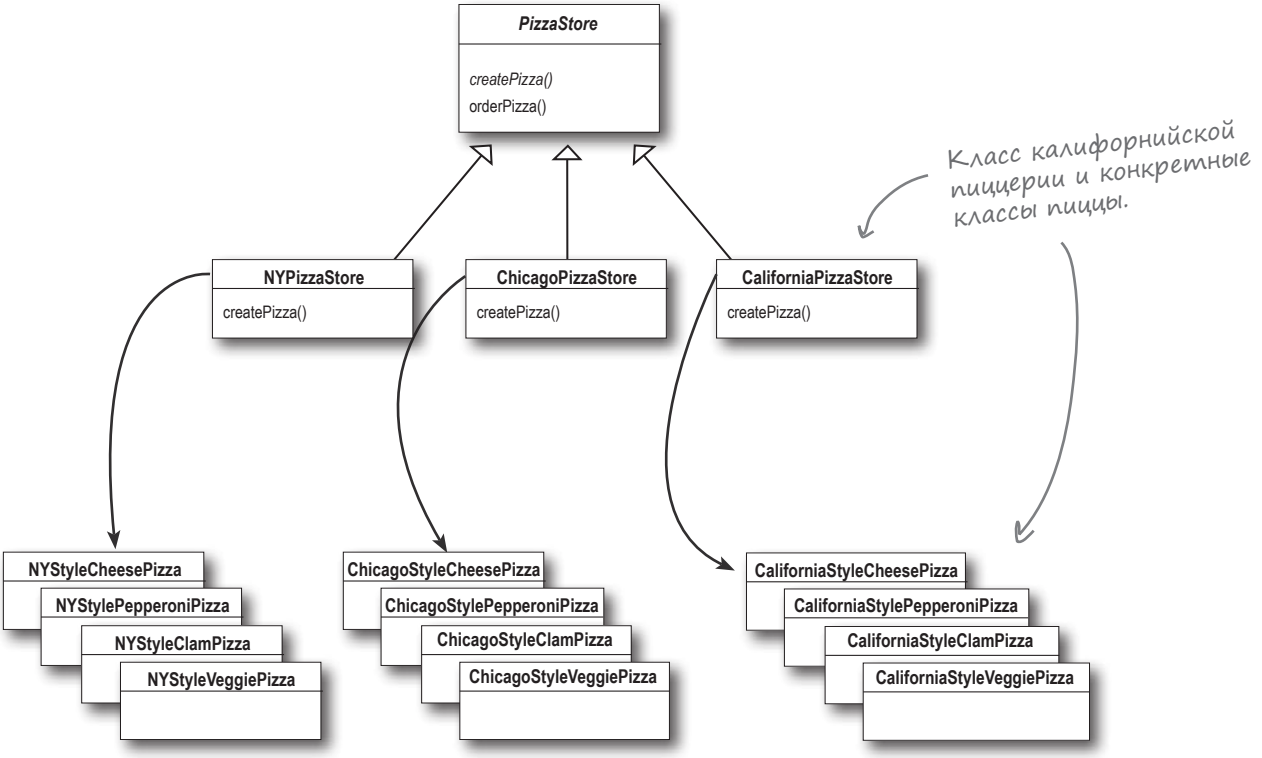
Для чикагской пиццерии необходимо создавать пиццу в чикагском стиле...

```
public class CaliforniaPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new CaliforniaStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new CaliforniaStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new CaliforniaStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new CaliforniaStylePepperoniPizza();
        } else return null;
    }
}
```

...а для калифорнийской — пиццу в калифорнийском стиле.

Решение головоломки

Нам нужна еще одна разновидность пиццы для этих безумных калифорнийцев («безумных» в *хорошем* смысле слова, конечно). Нарисуйте еще один параллельный набор классов, необходимых для поддержки нового региона – Калифорнии – в PizzaStore.



Теперь запишите пять самых *странных* ингредиентов для пиццы... И можете открывать бизнес по выпечке пиццы в Калифорнии!

Наши предложения...

- Картофельное пюре с жареным чесноком
- Соус «барбекю»
- Артишоки
- M&M's
- Арахис

Очень зависимый PizzaStore

Возьми в руку карандаш
Решение

Представьте, что вы никогда не слышали о ОО-фабриках. Ниже приведена версия PizzaStore, в которой фабрики не используются; подсчитайте количество конкретных объектов пиццы, от которых зависит этот класс. А если добавить пиццу в калифорнийском стиле, от скольких объектов он будет зависеть тогда?

```
public class DependentPizzaStore {

    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new NYStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new ChicagoStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new ChicagoStylePepperoniPizza();
            }
        } else {
            System.out.println("Error: invalid type of pizza");
            return null;
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

Обработка всех видов нью-йоркской пиццы.

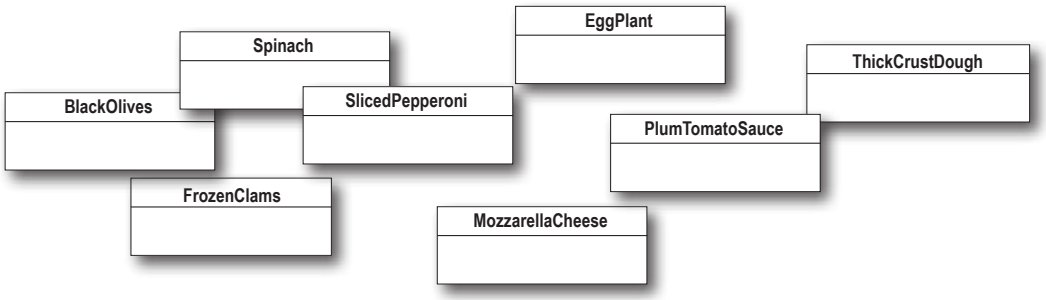
Обработка всех видов чикагской пиццы.

Впишите свои ответы: 8 *число* 12 *с калифорнийской пиццей.*

Возьми в руку карандаш
Решение

Напишите фабрику ингредиентов ChicagoPizzaIngredientFactory. Используйте в своей реализации классы, приведенные ниже:

```
public class ChicagoPizzaIngredientFactory
    implements PizzaIngredientFactory
{
    public Dough createDough() {
        return new ThickCrustDough();
    }

    public Sauce createSauce() {
        return new PlumTomatoSauce();
    }

    public Cheese createCheese() {
        return new MozzarellaCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new BlackOlives(),
                              new Spinach(),
                              new Eggplant() };
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }

    public Clams createClam() {
        return new FrozenClams();
    }
}
```



Уникальные объекты

Паттерн *Одиночка* направлен на *создание уникальных объектов, существующих только в одном экземпляре*. Из всех паттернов Одиночка имеет самую простую диаграмму классов; собственно, вся диаграмма состоит из одного-единственного класса! Однако не стоит расслабляться; несмотря на всю его простоту с точки зрения архитектуры классов, в его реализации кроется немало ловушек. Так что пристегните ремни!

Программист: И в чем польза такого паттерна?

Гуру: Существует много объектов, которые нужны нам только в одном экземпляре: пулы программных потоков, кэши, диалоговые окна, объекты ведения журнала, объекты драйверов устройств... Более того, для многих типов таких объектов попытка создания более одного экземпляра приведет к всевозможным проблемам — некорректному поведению программы, лишним затратам ресурсов или нелогичным результатам.

Программист: Хорошо, некоторые классы действительно должны создаваться только в одном экземпляре. Но писать об этом целую главу? Разве нельзя, например, воспользоваться глобальной переменной? А в Java можно было добиться желаемого с помощью статической переменной.

Гуру: Паттерн Одиночка во многих отношениях представляет собой **схему**, которая гарантирует, что для заданного класса может быть создан один и только один объект. Если кто-нибудь придумает более удачное решение, мир о нем услышит; а пока паттерн Одиночка, как и все паттерны, представляет собой проверенный временем механизм создания единственного объекта. Кроме того, Одиночка, как и глобальная переменная, предоставляет глобальную точку доступа к данным, но без ее недостатков.

Программист: Каких недостатков?

Гуру: Простейший пример: если объект присваивается глобальной переменной, он может быть создан в начале работы приложения. Верно? А если этот объект расходует много ресурсов, но никогда не будет использоваться приложением? Как вы увидите, паттерн Одиночка позволяет создавать объекты в тот момент, когда в них появится необходимость.

Программист: И все равно не вижу ничего особенно сложного.

Гуру: Для того, кто хорошо разбирается в статических переменных и методах, а также модификаторах доступа, — ничего сложного нет. Но как бы то ни было, интересно посмотреть, как работает паттерн Одиночка, и при всей простоте его довольно непросто реализовать. Как бы вы предотвратили создание повторных экземпляров? Задача отнюдь не тривиальная, не правда ли?

Вопросы и ответы

Маленькое упражнение в стиле сократовских диалогов

Как создать один объект?

```
new MyObject();
```

А если другой объект захочет создать еще один экземпляр MyObject? Сможет ли он снова вызвать new для MyObject?

Да, конечно.

Значит, если у нас есть класс, мы всегда можем создать один или несколько экземпляров этого класса?

Да. Но только если это открытый класс.

А если нет?

А если это не открытый класс, то его экземпляры могут создаваться только классами того же пакета. Но они все равно могут создать несколько экземпляров.

Хмм, интересно.

А вы знаете, что можно сделать так?

```
public MyClass {

   private MyClass() {}

}
```

Нет, никогда не задумывался. Но такое определение выглядит разумно и ничего не нарушает.

И что это значит?

Вероятно, это класс, экземпляры которого не могут создаваться, потому что конструктор объявлен приватным.

Хоть КАКОЙ-НИБУДЬ объект может вызывать приватный конструктор?

Хмм, он может вызываться только из кода MyClass. Но какая от этого польза?

Почему?

Потому что для вызова нужно иметь экземпляр класса, а я не могу создать экземпляр, потому что он не может быть создан другим классом. Классическая проблема «курицы и яйца».

Понятно.

А что можно сказать об этом фрагменте?

```
public MyClass {

   public static MyClass getInstance() {
   }
}
```

Класс MyClass содержит статический метод, который можно вызвать так:

```
MyClass.getInstance();
```

Почему вместо имени объекта используется MyClass?

Метод getInstance() является статическим, то есть методом КЛАССА. При вызове статического метода необходимо указывать имя класса.

Очень интересно.

А *теперь* я могу создать экземпляр MyClass?

```
public MyClass {

    private MyClass() {}

    public static MyClass getInstance() {
        return new MyClass();
    }
}
```

Безусловно.

Как должны создаваться экземпляры в клиентском коде?

```
MyClass.getInstance();
```

Можете ли вы дописать код, чтобы он всегда создавал не более ОДНОГО экземпляра MyClass?

Да, пожалуй...

(*Ответ на следующей странице)*

Классическая реализация паттерна Одиночка

Класс MyClass переименован в Singleton.

Статическая переменная для хранения единственного экземпляра.

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // Другие методы
}
```

Приватный конструктор; только Singleton может создавать экземпляры этого класса!

Метод getInstance() создает и возвращает экземпляр.

Как и всякий другой класс, Singleton содержит другие переменные и методы экземпляров.

Будьте осторожны!

Если вы бегло просматриваете книгу, не торопитесь использовать этот код. Как будет показано позднее в этой главе, он нуждается в доработке.

Код под увеличительным стеклом

uniqueInstance содержит ЕДИНСТВЕННЫЙ экземпляр; не забудьте, что это статическая переменная.

Если uniqueInstance содержит null, значит, экземпляр еще не создан...

...тогда мы создаем экземпляр Singleton приватным конструктором и присваиваем его uniqueInstance.

```
if (uniqueInstance == null) {
    uniqueInstance = new MyClass();
}
return uniqueInstance;
```

Если uniqueInstance уже содержит значение, сразу переходим к команде return.

К моменту выполнения этой команды экземпляр уже создан — возвращаем его.

Паттерны для всех

Интервью недели: Признания Одиночки

HeadFirst: Сегодня вниманию слушателей предлагается интервью с объектом Одиночкой. Может, немного расскажете о себе?

Одиночка: Я единственный и неповторимый!

HeadFirst: Неужели?

Одиночка: Да. Я создан на основе паттерна Одиночка, а это гарантирует, что в любой момент времени существует только один экземпляр моего класса.

HeadFirst: Разве это не расточительно? Кто-то тратит время на создание класса, а потом по этому описанию создается всего один объект?

Одиночка: Вовсе нет! Во многих случаях существование одиночных объектов оправдано. Предположим, в объекте хранятся настройки реестра. Если кто-то создаст несколько экземпляров такого объекта, это может привести к хаосу. Одиночные объекты гарантируют, что все компоненты вашего приложения будут использовать один и тот же глобальный ресурс.

HeadFirst: Продолжайте...

Одиночка: О, я отлично подхожу для подобных задач. Одиночество имеет свои преимущества. Меня часто используют для управления пулами ресурсов — скажем, подключений или программных потоков.

HeadFirst: И все-таки — быть всегда одним? Вам не скучно?

Одиночка: Так как мне никто не помогает, скучать некогда. Но разработчикам не помешало бы получше изучить меня — во многих программах возникают ошибки, когда в системе создаются дополнительные экземпляры объектов, о которых их создатель и не подозревает.

HeadFirst: Но как можно быть уверенным в том, что вы существуете в одном экземпляре? Разве любой желающий не сможет создать новый экземпляр оператором new?

Одиночка: Нет! Я абсолютно уникален.

HeadFirst: Разработчики торжественно клянутся не создавать более одного экземпляра?

Одиночка: Нет, конечно. Возможно, это дело личное, но... у меня нет открытого конструктора.

HeadFirst: НЕТ ОТКРЫТОГО КОНСТРУКТОРА?! Ох, извините... неужели нет?

Одиночка: Вот именно. Мой конструктор объявлен приватным..

HeadFirst: И как работает эта схема? Как ВООБЩЕ можно создать экземпляр?

Одиночка: Чтобы получить объект, вы не создаете его экземпляр, а *запрашиваете* его. Мой класс содержит статический метод с именем getInstance(). Вызовите его, и вы получите экземпляр, готовый к работе. Может оказаться, что к этому моменту я уже существую и обслуживаю другие объекты.

HeadFirst: Похоже, в этой схеме многое остается скрытым от посторонних глаз! Спасибо, что приняли наше приглашение. Надеемся, наша беседа скоро продолжится!

Шоколадная фабрика

Всем известно, что на всех современных шоколадных фабриках используются нагреватели с компьютерным управлением. Смесь шоколада и молока в таком нагревателе доводится до кипения и передается на следующий этап изготовления шоколадных батончиков. Ниже приведен код класса, управляющего Choc-O-Holic — мощным высокопроизводительным нагревателем для шоколада. Просмотрите код; вы заметите, что автор постарался сделать все возможное, чтобы избежать некоторых неприятностей — скажем, слива холодной смеси, переполнения или нагревания пустой емкости!

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    public  ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // Заполнение нагревателя молочно-шоколадной смесью
        }
    }

    public void drain() {
        if (!isEmpty() && isBoiled()) {
            // Слить нагретое молоко и шоколад
            empty = true;
        }
    }

    public void boil() {
        if (!isEmpty() && !isBoiled()) {
            // Довести содержимое до кипения
            boiled = true;
        }
    }

    public boolean isEmpty() {
        return empty;
    }

    public boolean isBoiled() {
        return boiled;
    }
}
```

Код выполняется только при пустом нагревателе!

Перед наполнением мы проверяем, что нагреватель пуст, а после наполнения устанавливаем флаги empty и boiled.

Чтобы слить содержимое, мы проверяем, что нагреватель не пуст, а смесь доведена до кипения. После слива флагу empty снова присваивается true.

Чтобы вскипятить смесь, мы проверяем, что нагреватель полон, но еще не нагрет. После нагревания флагу boiled присваивается true.

МОЗГОВОЙ ШТУРМ

Автор кода неплохо позаботился о том, чтобы избежать любых проблем. Но если в системе вдруг появятся два экземпляра ChocolateBoiler, возможны самые неприятные последствия.

Что может произойти, если приложение создаст два и более экземпляра ChocolateBoiler?

Возьми в руку карандаш

Сможете ли вы усовершенствовать класс ChocolateBoiler, преобразовав его в синглетную форму (то есть с единственным экземпляром)?

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    [                                                    ]

    [            ] ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    [                                                    ]
    [                                                    ]
    [                                                    ]

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // Заполнение нагревателя молочно-шоколадной смесью
        }
    }
    // Остальной код ChocolateBoiler...
}
```

Определение паттерна Одиночка

Итак, вы познакомились с классической реализацией паттерна Одиночка. Сейчас можно устроиться поудобнее, съесть шоколадку и перейти к рассмотрению некоторых нюансов паттерна Одиночка.

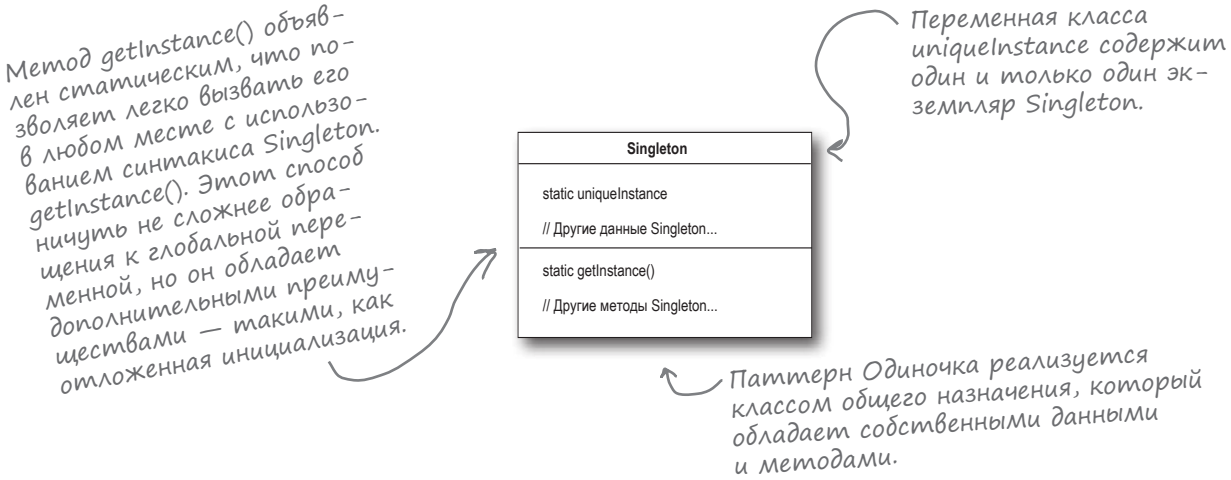
Начнем с формального определения паттерна:

> **Паттерн Одиночка** гарантирует, что класс имеет только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру.

Пока ничего особенного. Но давайте присмотримся повнимательнее:

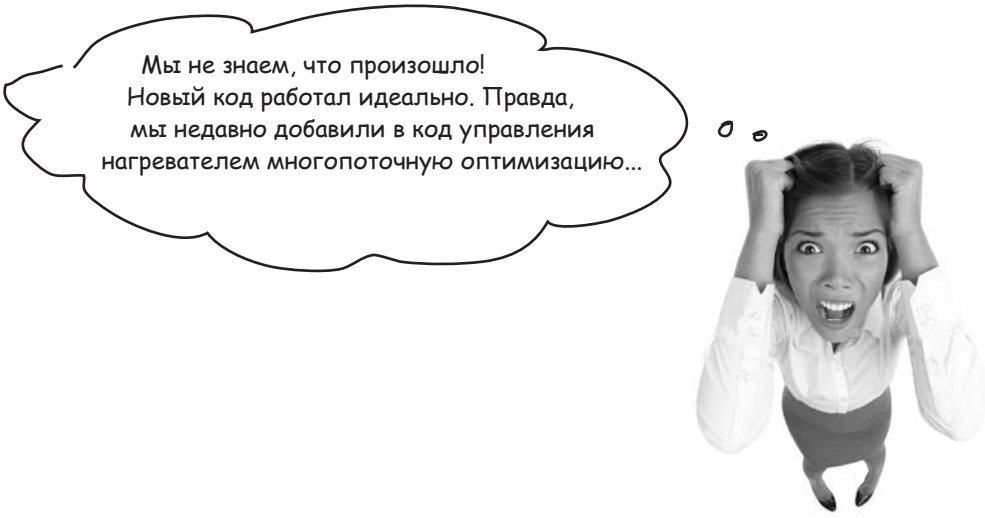
- Что здесь по сути происходит? Мы берем существующий класс и разрешаем ему создать только один экземпляр. Кроме того, мы запрещаем любым другим классам произвольно создавать новые экземпляры этого класса. Чтобы получить экземпляр, необходимо обратиться с запросом к самому классу.
- Кроме того, паттерн предоставляет глобальную точку доступа к экземпляру: обратившись с запросом к классу в любой точке программы, вы получите ссылку на единственный экземпляр. Как было показано выше, возможно отложенное создание экземпляра, что особенно важно для объектов, создание которых сопряжено с большими затратами ресурсов.

Обратимся к диаграмме классов:



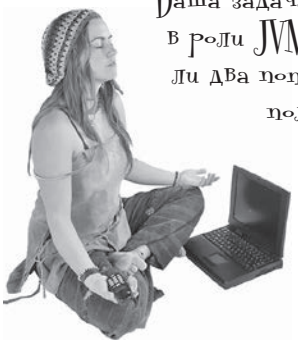
Кажется, у нас проблемы...

Похоже, программа управления нагревателем нас подвела; хотя мы усовершенствовали код и перешли на классическую реализацию паттерна Одиночка, метод fill() класса ChocolateBoiler почему-то начал заполнять емкость, которая уже была заполнена! 500 галлонов молока (и шоколада) пролилось на пол. Что произошло?!



Могло ли введение программных потоков привести к катастрофе? Ведь после того, как переменной uniqueInstance будет присвоен единственный экземпляр ChocolateBoiler, все вызовы getInstance() должны возвращать один и тот же экземпляр? Разве нет?

Представьте, что вы — JVM...



Есть два программных потока, в каждом из которых выполняется этот код. Ваша задача — представить себя в роли JVM и определить, не могут ли два потока в какой-то ситуации получить два разных объекта нагревателя. Подсказка: проанализируйте последовательность операций в методе getInstance() и значение uniqueInstance, и подумайте, могут ли они перекрываться. Магниты с фрагментами кода помогут вам в поиске возможной последовательности действий, в результате которой программа может получить два объекта ChocolateBoiler.

```
ChocolateBoiler boiler =
        ChocolateBoiler.getInstance();
fill();
boil();
drain();
```

Прежде чем читать дальше, проверьте свой ответ на с. 217!

```
public static ChocolateBoiler
        getInstance() {
    if (uniqueInstance == null) {
        uniqueInstance =
            new ChocolateBoiler();
    }
    return uniqueInstance;
}
```

Первый поток	Второй поток	Значение uniqueInstance

Решение проблемы многопоточного доступа

Наши многопоточные проблемы решаются почти тривиально: метод getInstance() объявляется синхронизированным:

```
public class Singleton {
    private static Singleton uniqueInstance;

    // Другие переменные экземпляра

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // Другие методы
```

Включая в объявление getInstance() ключевое слово synchronized, мы заставляем каждый поток дождаться своей очереди для входа в него. Иначе говоря, два потока не смогут войти в метод одновременно.

В точку! На самом деле ситуация еще серьезнее: синхронизация актуальна только при первом вызове этого метода. Иначе говоря, после того, как переменной uniqueInstance будет присвоен экземпляр синглетного класса, необходимость в дальнейшей синхронизации этого метода отпадает. После первого выполнения синхронизация только приводит к лишним затратам ресурсов!

Можно ли усовершенствовать многопоточную реализацию?

Безусловно, в большинстве Java-приложений мы должны позаботиться о том, чтобы паттерн работал в многопоточном коде. Но синхронизация метода getInstance() приводит к значительным затратам ресурсов. Что же делать?

Есть несколько вариантов.

1. Ничего не делать, если производительность getInstance() не критична для вашего приложения

Да, вы не ошиблись: если вызов getInstance() не приводит к заметному ухудшению быстродействия, не обращайте внимания. Синхронизация getInstance() — простое и эффективное решение. Только помните, что синхронизация метода может замедлить его выполнение в сто и более раз. Если метод getInstance() применяется в интенсивно используемой части приложения, возможно, вам стоит пересмотреть свое решение.

2. Создавайте экземпляр заранее

Если приложение всегда создает и использует экземпляр синглетного класса или затраты за создание не столь существенны, экземпляр можно создать заранее:

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

Экземпляр Singleton создается в статическом инициализаторе. Потоковая безопасность этого кода гарантирована!

Экземпляр уже существует, просто возвращаем его.

При таком подходе мы доверяем JVM создание уникального экземпляра Singleton при загрузке класса. JVM гарантирует, что экземпляр будет создан до того, как какой-либо поток сможет обратиться к статической переменной uniqueInstance.

3. Воспользуйтесь «условной блокировкой», чтобы свести к минимуму использование синхронизации в getInstance()

Сначала проверьте, создается ли новый экземпляр, и если нет – ТОГДА синхронизируйте фрагмент кода. В этом случае синхронизация будет выполняться только при первом вызове (что нам, собственно, и требовалось).

Давайте проверим код:

```
public class Singleton {
    private volatile* static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

Проверяем экземпляр; если он не существует — входим в блок synchronized.

Синхронизация выполняется только при первом вызове!

Внутри блока повторяем проверку, и если значение все еще равно null — создаем экземпляр.

* *Ключевое слово volatile гарантирует, что параллельные программные потоки будут правильно работать с переменной uniqueInstance при ее инициализации экземпляром Singleton.*

Если производительность выполнения getInstance() критична, то этот способ реализации кардинально ускорит выполнение метода.

Будьте осторожны!

Условная блокировка не работает в Java 1.4 и более ранних версиях!

К сожалению, многие реализации ключевого слова volatile в Java 1.4 и более ранних версиях допускают неверную синхронизацию условной блокировки. При необходимости использования JVM-версии, отличной от Java 5, рассмотрите другие способы реализации паттерна Одиночка.

Тем временем на шоколадной фабрике...

Пока мы разбирались с проблемами многопоточности, нагреватель уже почистили, и он снова готов к работе. Но сначала необходимо исправить допущенную ошибку. Имеется несколько решений, каждое из которых обладает своими достоинствами и недостатками; какое из них следует применить?

Возьми в руку карандаш

Опишите пригодность каждого решения для решения проблемы многопоточности в коде ChocolateBoiler:

Синхронизация метода getInstance():

Раннее создание экземпляра:

Условная блокировка:

Поздравляем!

К этому моменту все проблемы шоколадной фабрики решены. Какое бы из многопоточных решений вы ни применили, нагреватель работает нормально, а новые сбои исключены. Поздравляем! В этой главе мы не только разлили 500 галлонов горячего шоколада, но и рассмотрели все потенциальные проблемы паттерна Одиночка.

Часто Задаваемые Вопросы

В: Для такого простого паттерна, состоящего из одного класса, у Одиночки многовато проблем.

О: А мы вас предупреждали заранее! Пусть проблемы вас не смущают; возможно, правильная реализация потребует некоторых усилий, после чтения этой главы вы достаточно хорошо разбираетесь во всех тонкостях и сможете использовать этот паттерн для ограничения количества создаваемых экземпляров.

В: А почему не создать класс, у которого все методы и переменные определены как статические? Разве это не приведет к тому же результату?

О: Да, если ваш класс автономен и не участвует в сложных инициализациях. Но из-за особенностей статической инициализации в Java (прежде всего с участием нескольких классов) в этом сценарии возможны коварные, трудноуловимые ошибки, связанные с порядком инициализации. Если у вас нет веских причин для выбора именно такой реализации, лучше придерживаться традиционной объектной парадигмы.

В: А как насчет загрузчиков классов? Говорят, существует вероятность того, что два загрузчика классов получат собственный экземпляр синглетного класса.

О: Да, это правда, так как каждый загрузчик классов определяет пространство имен. Если вы используете два и более загрузчика, вы сможете загрузить один класс многократно (по одному разу для каждого загрузчика). Если класс будет синглетным, то мы получим несколько экземпляров синглетного объекта. Будьте внимательны! Одно из возможных решений — самостоятельное назначение загрузчика.

В: Меня всегда учили, что класс должен делать что-то одно. Совмещение классом двух функций считается признаком плохого ОО-проектирования. Разве паттерн Одиночка не нарушает это правило?

Расслабьтесь

Слухи об опасности уборщика мусора сильно преувеличены

До выхода Java 1.2 ошибка в уборщике мусора допускала преждевременное уничтожение синглетных объектов при отсутствии глобальных ссылок на них. Другими словами, вы создавали объект Singleton, и если единственная ссылка на него хранилась только в самом объекте Singleton — он «освобождался» и мог быть уничтожен уборщиком мусора. Это приводило к хитроумным ошибкам, потому что после «освобождения» следующий вызов getInstance() создавал новенький объект Singleton. Во многих приложениях это приводило к загадочным последствиям: состояние само собой заново инициализировалось исходными значениями, происходил сброс сетевых подключений и т. д.

В Java 1.2 эта ошибка была исправлена, и глобальная ссылка перестала быть обязательной. Но если вы по какой-то причине продолжаете использовать JVM-версии до 1.2, помните об этой проблеме.

О: Вы говорите о «принципе одной обязанности»? Да, вы правы. Синглетный класс отвечает не только за управление своим экземпляром (и предоставление глобального доступа к нему), но и за выполнение своих основных функций в приложении. Таким образом, можно сказать, что он имеет не одну, а две обязанности. Однако все управление экземпляром осуществляется отдельной подсистемой класса; безусловно, это упрощает общую архитектуру. Кроме того, многие разработчики хорошо знают широко распространенный паттерн Одиночка. Тем не менее некоторые разработчики считают, что функциональность паттерна Одиночка следует абстрагировать за пределами класса.

В: Я хотел субклассировать свой синглетный класс, но у меня возникли проблемы. Можно ли субклассировать такие классы?

О: Одна из проблем с субклассированием синглетных классов связана с приватностью конструктора. Класс с приватным конструктором расширить нельзя. Следовательно, прежде всего конструктор придется изменить, чтобы он был открытым или защищенным. Но тогда класс перестает быть синглетным, потому что другие классы смогут создавать его экземпляры.

Изменение конструктора также создает другую проблему. Реализация паттерна Одиночка основана на статической переменной, поэтому при прямолинейном субклассировании все производные классы будут совместно использовать одну переменную экземпляра. Вероятно, это не то, что вам нужно, поэтому при субклассировании придется реализовать в базовом классе некое подобие системы управления доступом.

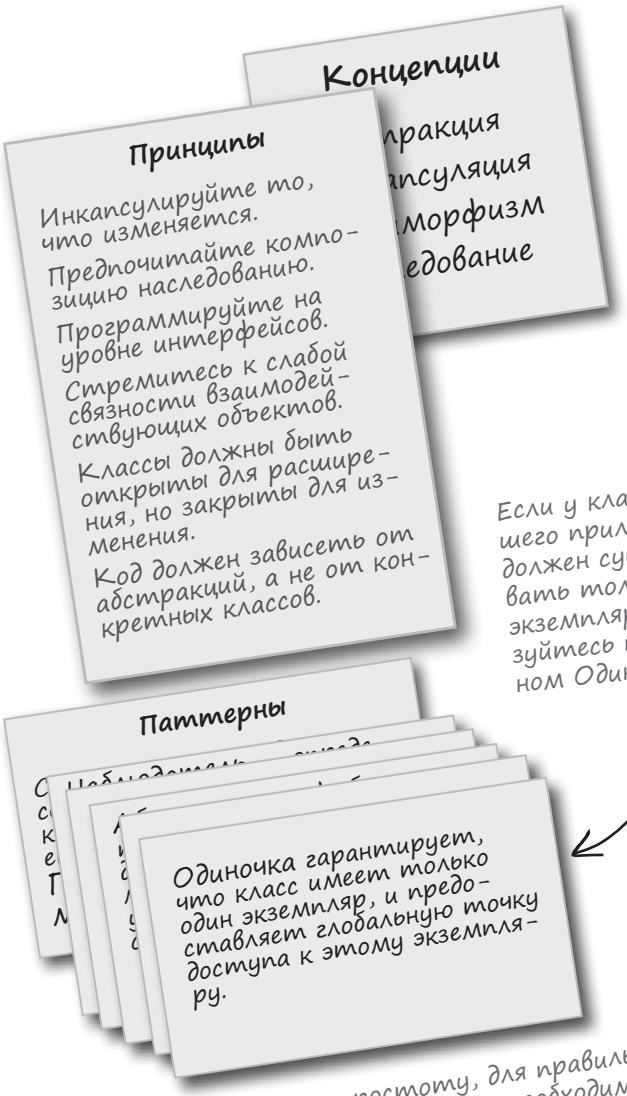
Прежде чем браться за реализацию такой схемы, спросите себя, какую пользу вам принесет субклассирование синглетного класса. Как и многие паттерны, этот не рассчитан на библиотечное использование, а его поддержка элементарно добавляется в любой существующий класс. Если в вашем приложении используется большое количество синглетных классов, возможно, вам стоит пересмотреть архитектуру.

В: Я все еще не понимаю, почему глобальные переменные хуже паттерна Одиночка.

О: В Java глобальные переменные фактически представляют собой статические ссылки на объекты. У подобного использования глобальных переменных имеется пара недостатков. Один уже упоминался ранее: немедленное создание экземпляра вместо отложенного. Но мы должны помнить о предназначении этого паттерна: он должен обеспечить существование только одного экземпляра класса и глобальный доступ к нему. Глобальная переменная обеспечивает второе, но не первое. Кроме того, глобальные переменные вынуждают разработчиков загрязнять пространство имен множеством глобальных ссылок на мелкие объекты. Синглетным классам этот недостаток не присущ, хотя возможны и другие злоупотребления.

Новые инструменты

Паттерн Одиночка определяет альтернативный способ создания объектов — в данном случае уникальных объектов.



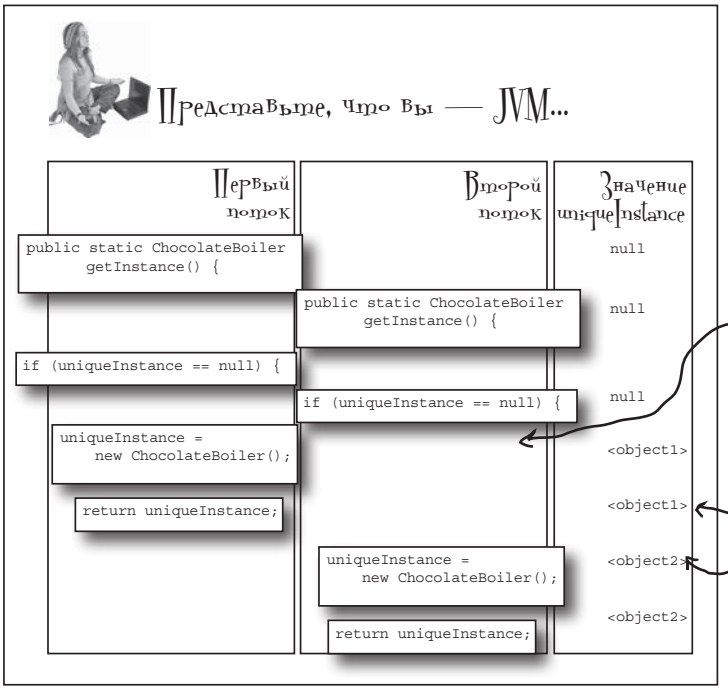
Если у класса вашего приложения должен существовать только один экземпляр, воспользуйтесь паттерном Одиночка.

Несмотря на внешнюю простоту, для правильной реализации паттерна Одиночка необходимо учесть многие нюансы. Впрочем, после прочтения этой главы вы сможете свободно использовать его в своем коде.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Паттерн Одиночка гарантирует, что в приложении существует не более одного экземпляра данного класса.
- Паттерн Одиночка также предоставляет глобальную точку доступа к этому экземпляру.
- Реализация паттерна Одиночка на языке Java использует приватный конструктор и статический метод в сочетании со статической переменной.
- Проанализируйте ограничения по производительности и затратам ресурсов, тщательно выберите реализацию Одиночки для многопоточного приложения.
- Будьте внимательны с условной блокировкой; до Java 2 версии 5 она была небезопасной.
- Будьте внимательны при использовании загрузчиков классов; они могут привести к созданию нескольких экземпляров, а это противоречит основной цели паттерна.
- Если вы используете JVM версии ранее 1.2, создайте реестр синглетных объектов, чтобы предотвратить их уничтожение уборщиком мусора.

Ответы к упражнениям



Возьми в руку карандаш

Решение

Сможете ли вы усовершенствовать класс ChocolateBoiler, преобразовав его в синглетную форму (то есть с единственным экземпляром)?

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    private static ChocolateBoiler uniqueInstance;

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public static ChocolateBoiler getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new ChocolateBoiler();
        }
        return uniqueInstance;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // Заполнение нагревателя молочно-шоколадной смесью
        }
    }
    // Остальной код ChocolateBoiler...
}
```

Возьми в руку карандаш
Решение

Опишите пригодность каждого решения для решения проблемы многопоточности в коде ChocolateBoiler:

Синхронизация метода getInstance():

Простое и заведомо рабочее решение. Хорошо подходит для нашей задачи, в которой нет проблем с быстродействием.

Ранняя инициализация:

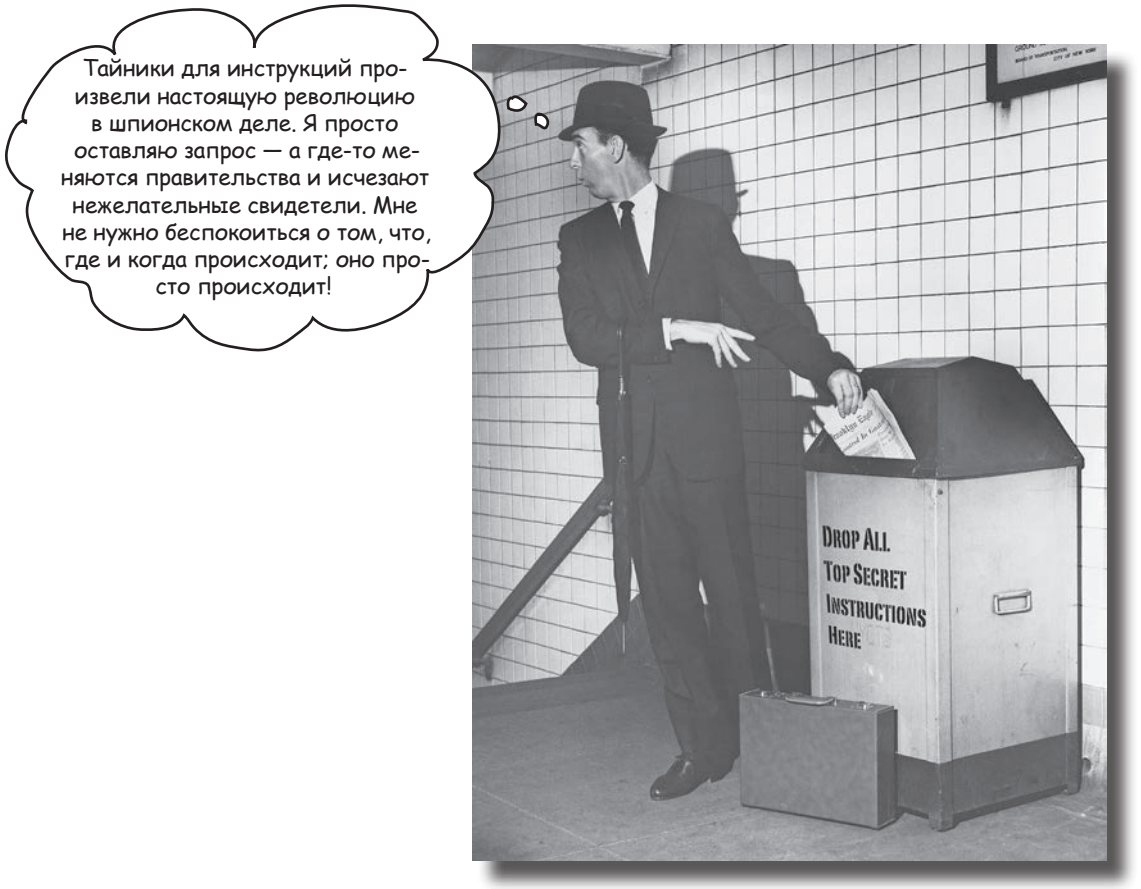
Экземпляр нагревателя в нашем приложении необходим всегда, и ранняя инициализация не создаст проблем.

Решение работает так же хорошо, как и синхронизация метода, хотя возможно, оно будет менее очевидным для разработчика, знакомого со стандартным паттерном.

Условная блокировка:

При отсутствии ограничений по производительности условная блокировка выглядит как перебор. Также не забудьте убедиться в том, что вы используете как минимум Java 5.

Инкапсуляция вызова



В этой главе мы выходим на новый уровень инкапсуляции — на этот раз будут инкапсулироваться вызовы методов. Да, все верно — вызывающему объекту не нужно беспокоиться о том, как будут выполняться его запросы. Он просто использует инкапсулированный метод для решения своей задачи. Инкапсуляция позволяет решать и такие нетривиальные задачи, как регистрация или отмена вызовов.

Home Automation or Bust, Inc.
1221 Industrial Avenue, Suite 2000
Future City, IL 62914

Недавно я получил от Джонни Харрикейна, исполнительного директора Weather-O-Rama, демо-версию и краткое описание их новой метеостанции. Архитектура продукта произвела на меня такое впечатление, что я решил предложить вашей фирме разработку API для нашего нового Пульта Домашней Автоматизации. Ваши услуги будут щедро оплачены акциями нашей фирмы.

Прилагаю прототип нашего революционного устройства. Пульт имеет семь программируемых ячеек (каждая из которых связывается с отдельным домашним устройством) и соответствующую кнопку «вкл/выкл» для каждой ячейки. Кроме того, устройство оснащено кнопкой глобальной отмены.

Также прилагается диск с набором классов Java, созданных разными фирмами-разработчиками для управления всевозможными домашними устройствами: светильниками, вентиляторами, ваннами-джакузи, акустическим оборудованием и т. д.

Ваша задача — создать API для программирования пульта, чтобы каждая ячейка могла быть настроена на управление устройством или группой устройств. Также следует учесть, что пульт должен поддерживать как текущий набор устройств, так и все устройства, которые могут быть добавлены в будущем.

После знакомства с работой, выполненной вами для метеостанции Weather-O-Rama, мы не сомневаемся, что вы отлично справитесь с разработкой программного обеспечения для пульта!

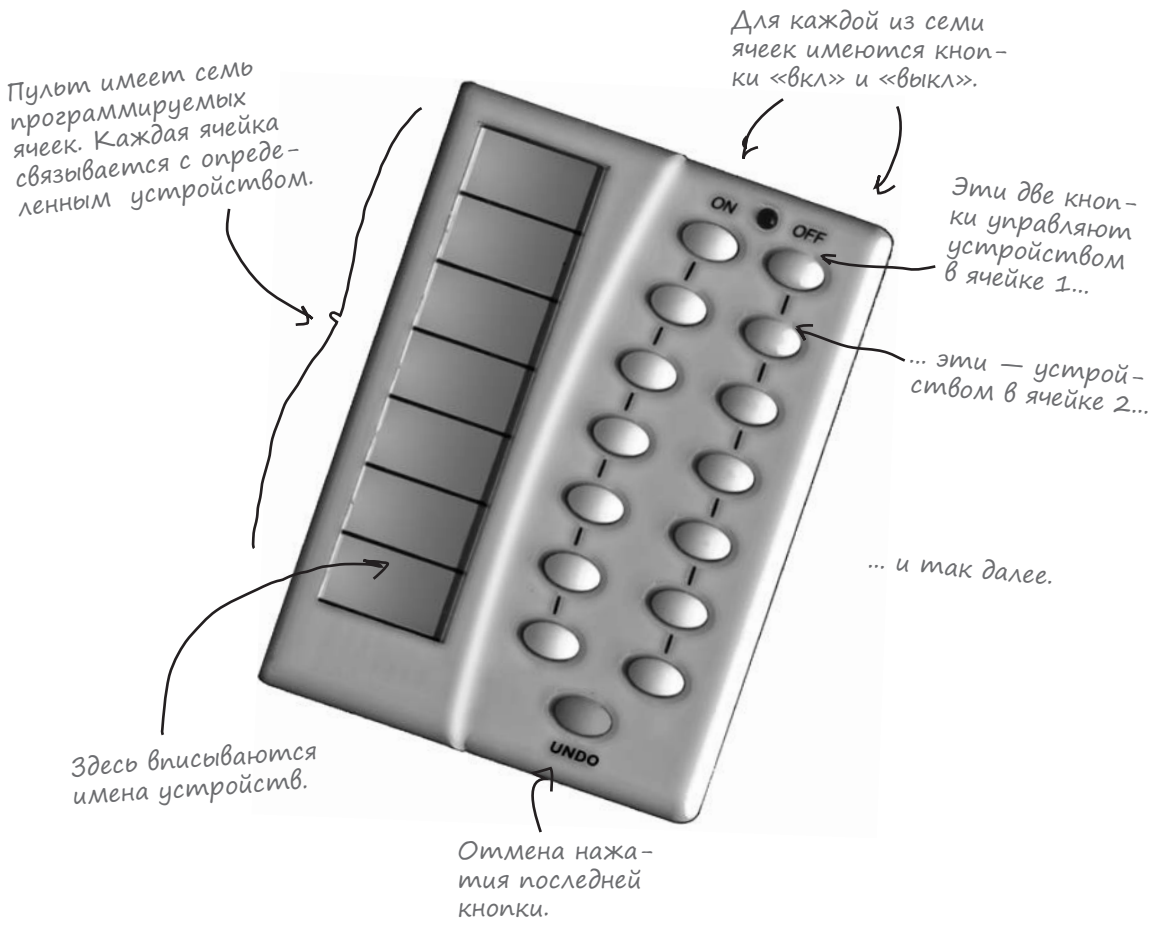
Надеемся на успешное сотрудничество.

Искренне ваш,

Билл «X-10» Томпсон, исполнительный директор

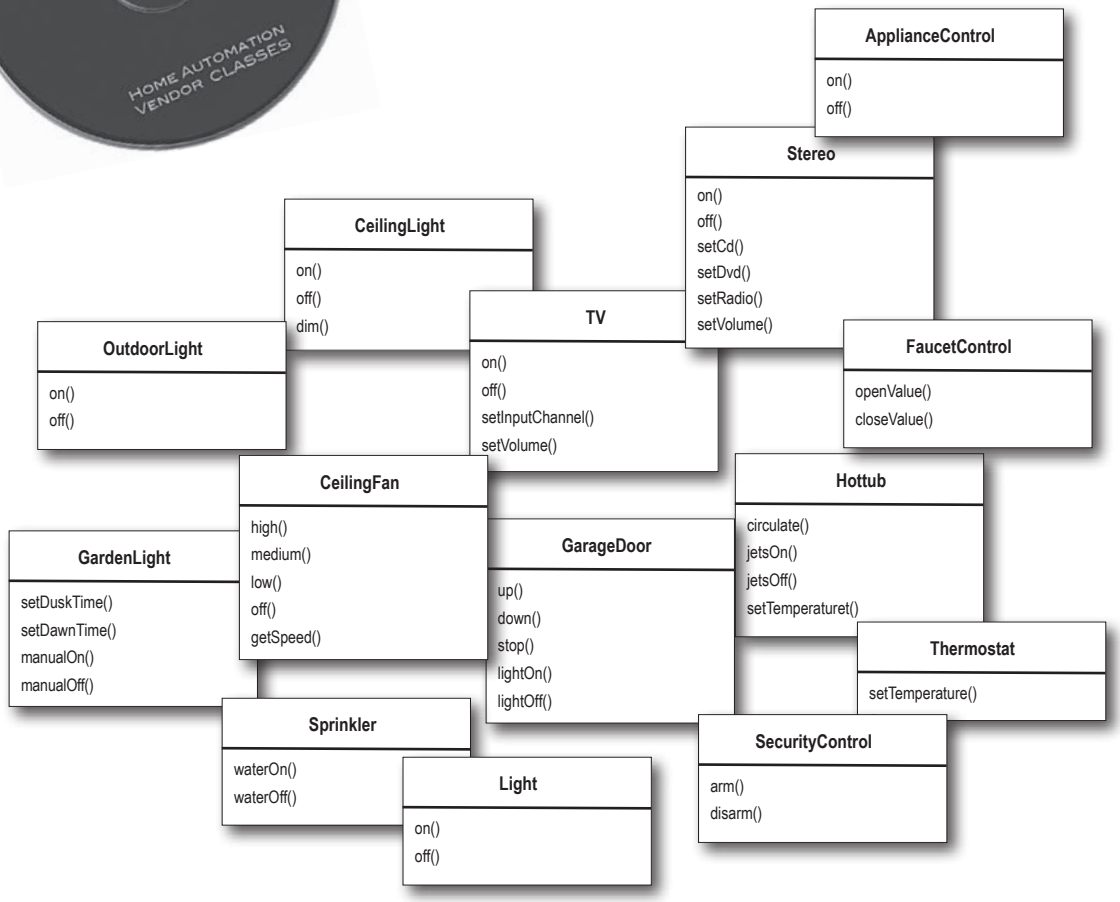
HOME AUTOMATION
VENDOR CLASSES

Посмотрим, чего от нас хотят...



Классы управления устройствами

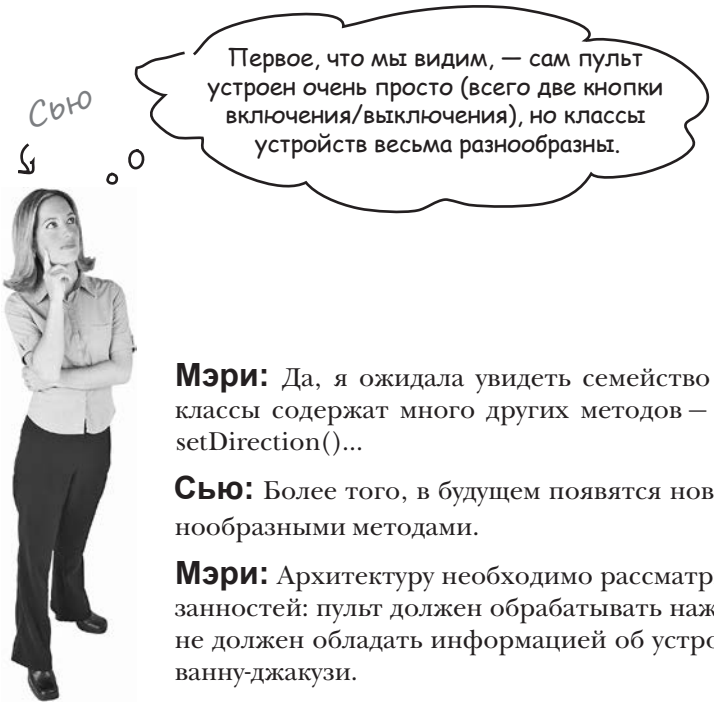
Рассмотрим классы управления устройствами с полученного диска — они дают представление об интерфейсах объектов, которыми должен управлять пульт.



Как видите, набор классов довольно обширный, а до появления унифицированного интерфейса еще очень далеко. Более того, в будущем наверняка появятся новые классы. Проектирование API для пульта в таких условиях — задача далеко не тривиальная.

Разговор в офисе

Ваши коллеги уже обсуждают, каким должен быть API...



Мэри: Да, я ожидала увидеть семейство классов с методами on() и off(), но классы содержат много других методов — dim(), setTemperature(), setVolume(), setDirection()...

Сью: Более того, в будущем появятся новые классы устройств с еще более разнообразными методами.

Мэри: Архитектуру необходимо рассматривать с точки зрения разделения обязанностей: пульт должен обрабатывать нажатия кнопок и выдавать запросы. Он не должен обладать информацией об устройствах, скажем, о том, как включить ванну-джакузи.

Сью: Разумно. Но если пульт умеет выдавать только обобщенные запросы, как спроектировать выполнение им разных операций: включения света, открытия гаражной двери и т. д.?

Мэри: Пока не знаю, но пульт в любом случае не должен привязываться к конкретной реализации классов устройств.

Сью: Что ты имеешь в виду?

Мэри: Программа не должна состоять из цепочки условных команд вида «if slot1 == Light, then light.on(), else if slot1 = Hottub then hottub.jetsOn()». Это признак плохой архитектуры.

Сью: Согласна. Ведь при появлении нового класса устройства нам неизбежно придется изменять код, а это повышает риск ошибок и создает дополнительную работу!

Мэри: Вот как? Расскажи подробнее.

Джо: Паттерн Команда отделяет сторону, выдающую запрос, от объекта, фактически выполняющего операцию. В нашем примере запрос поступает от пульта, а объектом, выполняющим операцию, будет экземпляр одного из классов устройств.

Сью: Но как такое возможно? Как разделить их? В конце концов, когда я нажимаю кнопку, пульт должен включить свет.

Джо: Для этого в архитектуру приложения вводятся «объекты команд». Объект команды инкапсулирует запрос на выполнение некой операции (скажем, включение света) с конкретным объектом (допустим, с осветительной системой). Если для каждой кнопки в приложении хранится свой объект команды, при ее нажатии мы обращаемся к объекту команды с запросом на выполнение операции. Сам пульт понятия не имеет, что это за операция, — он знает только, как взаимодействовать с нужным объектом для выполнения операции. Получается, что пульт полностью отделен от объекта осветительной системы!

Сью: Да, это похоже на то, что нам нужно.

Мэри: А я пока плохо представляю, как работает этот паттерн. Давайте посмотрим, правильно ли я понимаю: используя этот паттерн, мы можем создать API, в котором объекты команд связываются с определенными ячейками, благодаря чему код пульта остается очень простым. При этом выполнение операции инкапсулируется в том объекте, который эту операцию должен выполнять.

Джо: Да, вроде так. И еще мне кажется, что паттерн поможет реализовать функцию отмены, хотя я пока не размышлял об этом.

Мэри: Выглядит заманчиво, но, по-моему, чтобы действительно хорошо понять суть этого паттерна, мне придется изрядно потрудиться.

Сью: И мне тоже.

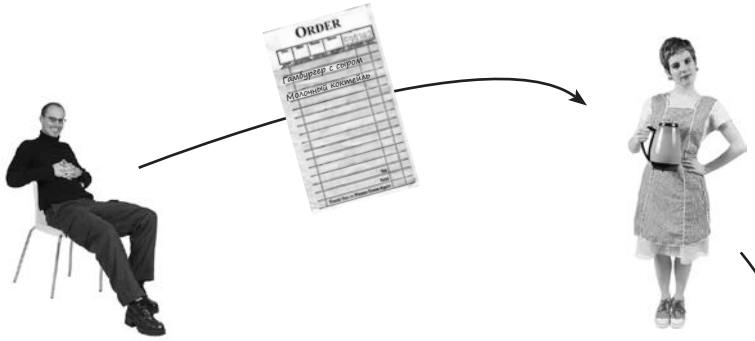
А тем временем в кафе..,
или Краткое введение в паттерн Команда

Паттерн Команда довольно трудно понять по описанию. Но не огорчайтесь, у нас есть помощники: помните кафе из главы 1? С тех пор, как мы навещали Элис и Фло, прошло немало времени, но у нас есть веские причины, чтобы вернуться туда (помимо еды и дружеского общения): кафе поможет нам лучше понять паттерн Команда.

Давайте проанализируем взаимодействия между посетителем, официанткой, заказами и поваром. Так вы лучше поймете суть взаимодействий объектов, задействованных в паттерне Команда. А потом мы сможем вплотную заняться проектированием API для пульта управления домашней техникой.

Итак, в кафе Объективиля...

Все мы отлично знаем, как работает кафе:



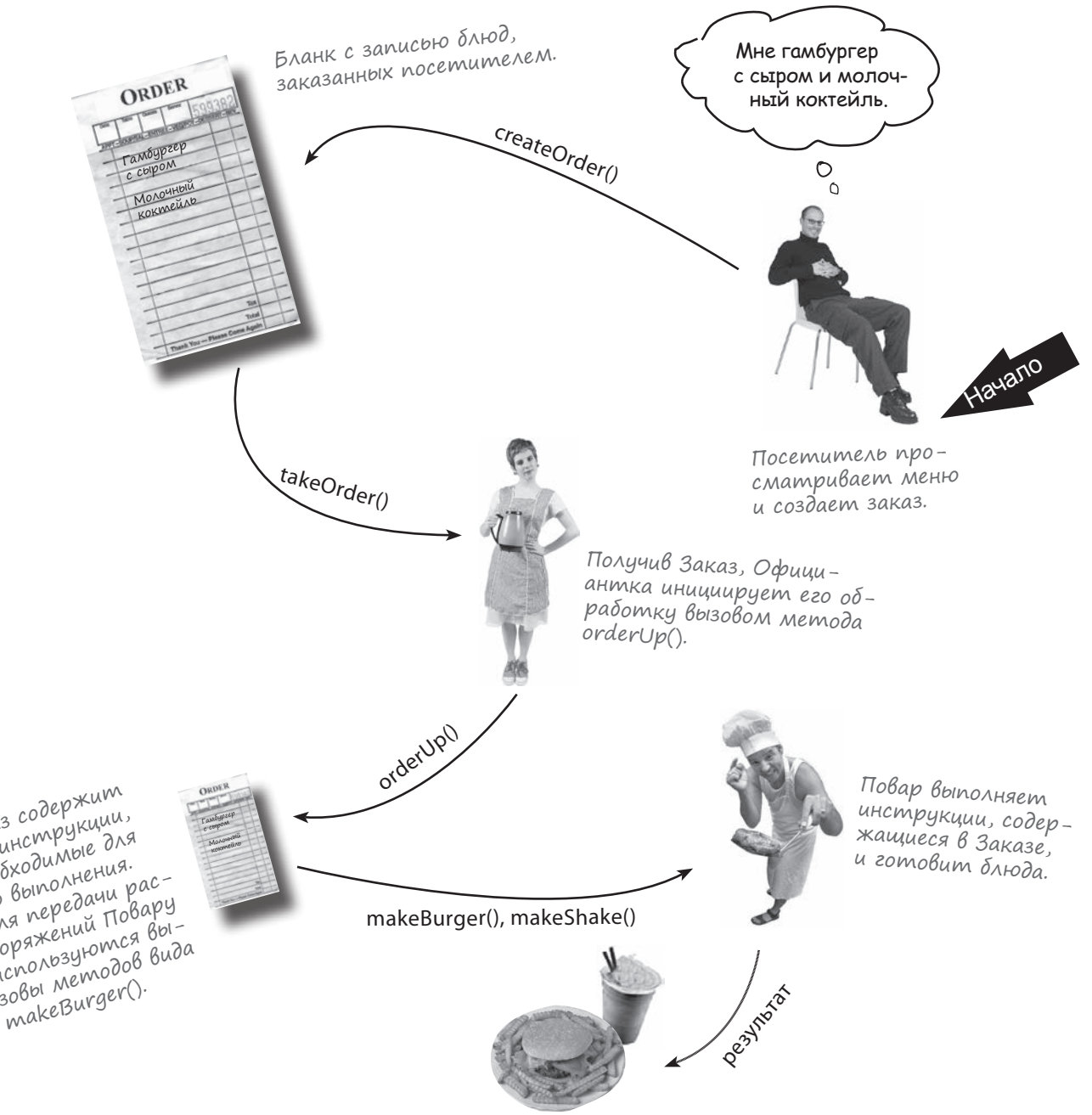
1 **Посетитель** передает **Официантке** свой **Заказ.**

2 **Официантка** получает **Заказ**, кладет его на стойку и говорит: «У нас заказ!»

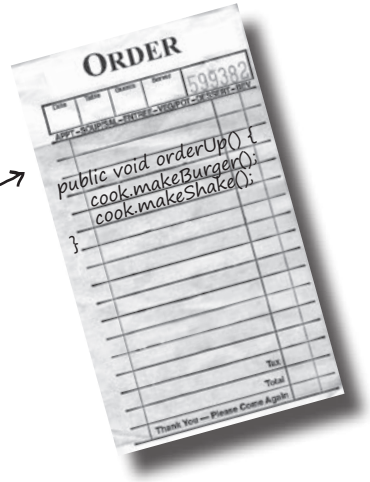
3 **Повар** готовит блюда, входящие в ваш **Заказ**.

Рассмотрим взаимодействия чуть более подробно...

...а так как кафе находится в Объективиле, мы используем для их описания объекты и методы!



Роли и обязанности в кафе Объективиля



Бланк заказа инкапсулирует запрос на приготовление блюд.

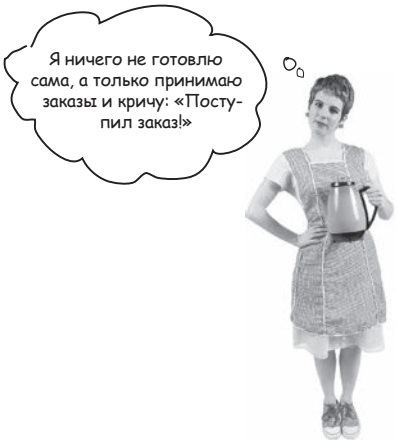
Будем рассматривать бланк Заказа как объект запроса на приготовление еды. Как и любой другой объект, он может передаваться — от Официантки на стойку или следующей Официантке, которая сменяет первую. Его интерфейс состоит из единственного метода orderUp(), инкапсулирующего все действия, необходимые для приготовления. Кроме того, Заказ содержит ссылку на объект, который должен готовить блюда. Инкапсуляция заключается в том, что Официантку не интересует содержимое заказа и то, кто будет его выполнять; она только кладет заказ на стойку и сообщает: «Поступил заказ!»

Наверное, в реальной жизни официантка следит за тем, что записано на Бланке и кто выполняет заказ, но мы-то находимся в Объективиле!

Задача Официантки — получить Заказ и вызвать его метод orderUp().

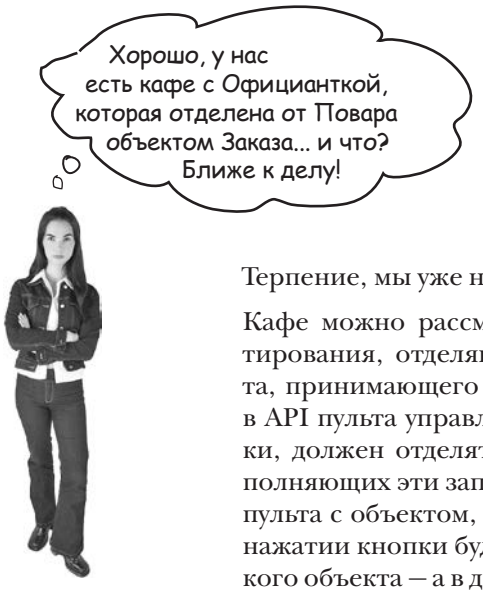
Работа Официантки проста: она получает Заказ и продолжает обслуживать посетителей, пока не вернется к стойке и не вызовет метод orderUp() для выполнения Заказа. Как упоминалось ранее, Официантка не беспокоится о том, что содержится в Заказе и кто его будет выполнять. Ей известно лишь то, что у Заказа есть метод orderUp(), который необходимо вызвать для выполнения операции.

В течение рабочего дня метод takeOrder() класса Официантки вызывается с множеством разных заказов от разных посетителей, но это Официантку не смущает. Она знает, что все Заказы поддерживают метод orderUp(), который необходимо вызвать для приготовления блюд.



Повар располагает всей информацией, необходимой для приготовления блюд.

Повар — тот объект, который умеет выполнять заказы. После того, как Официантка вызовет метод orderUp(), за дело берется Повар — он реализует все методы, необходимые для создания блюд. Обратите внимание: Официантка и Повар ничем не связаны. Вся информация о заказанных блюдах инкапсулирована в Заказе; Официантка только вызывает метод для каждого Заказа, чтобы он был выполнен. Повар получает свои инструкции из бланка Заказа; ему никогда не приходится взаимодействовать с Официанткой напрямую.



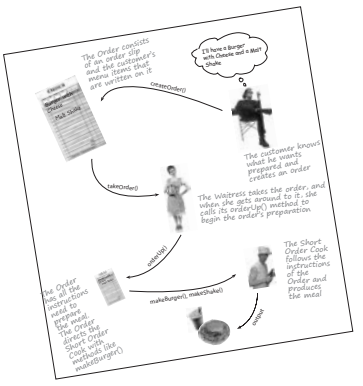
Терпение, мы уже недалеко от цели...

Кафе можно рассматривать как модель паттерна ОО-проектирования, отделяющего объект-источник запроса от объекта, принимающего и выполняющего эти запросы. Например, в API пульта управления код, вызываемый при нажатии кнопки, должен отделяться от объектов конкретных классов, выполняющих эти запросы. Почему бы не связать каждую ячейку пульта с объектом, аналогичным объекту заказа из кафе? При нажатии кнопки будет вызываться аналог метода orderUp() такого объекта — а в доме будет включаться свет, причем пульт не будет ничего знать о том, как он включается и какие объекты задействованы в выполнении операции.

А теперь давайте немного сменим тему и перейдем от кафе к паттерну Команда...

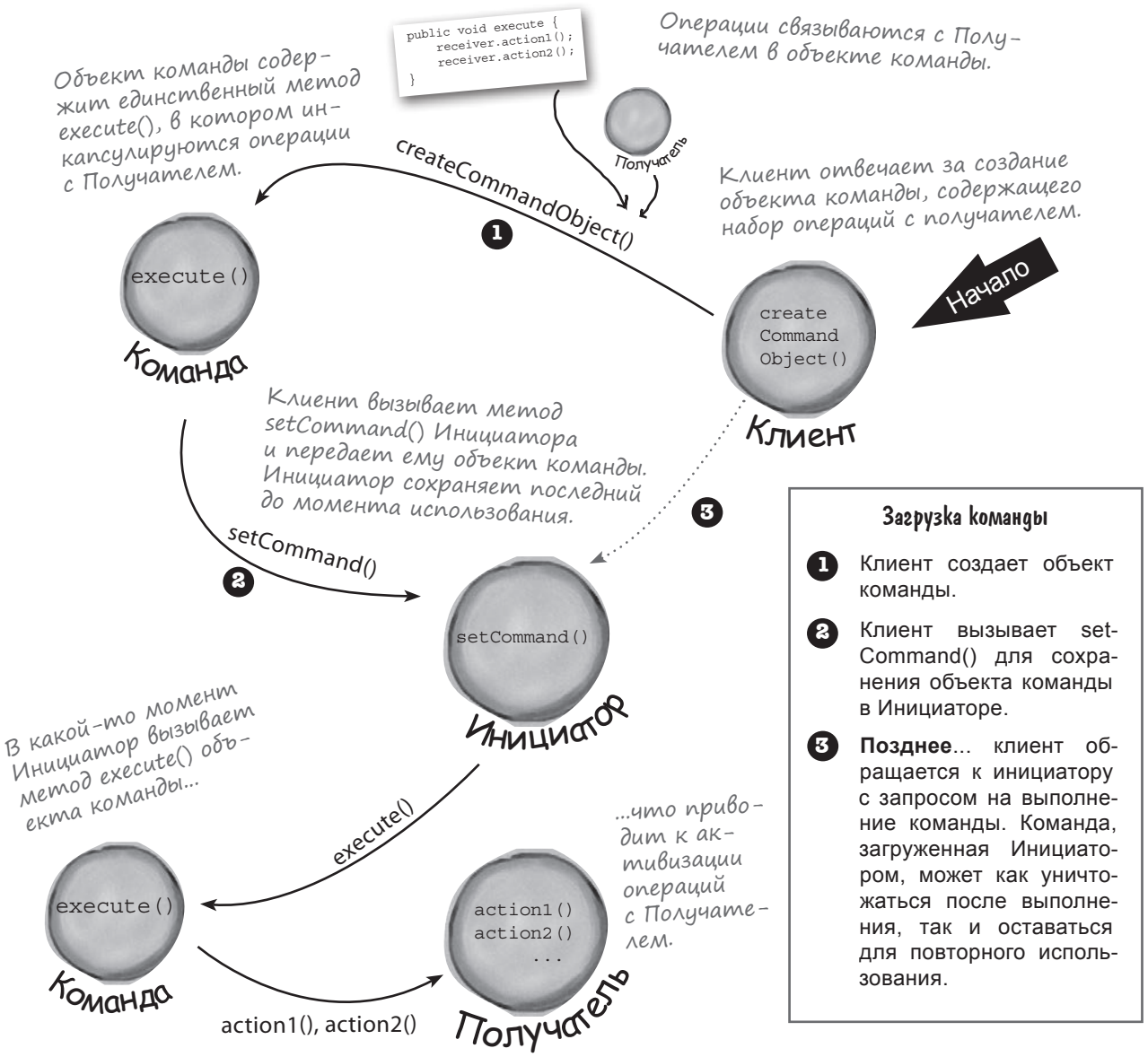
МОЗГОВОЙ ШТУРМ

Прежде чем двигаться дальше, хорошенько изучите приведенную пару страниц назад диаграмму с ролями и обязанностями, пока вы не начнете хорошо разбираться в объектах и взаимоотношениях. А когда это будет сделано — беритесь за паттерн Команда!



От кафе к паттерну Команда

Мы провели в кафе Объективиля достаточно времени, чтобы изучить всех действующих лиц и их обязанности. А теперь диаграмма выполнения заказа будет переработана для паттерна Команда. Вы увидите, что участники остались прежними; изменились только имена!



Загрузка команды

1. Клиент создает объект команды.
2. Клиент вызывает setCommand() для сохранения объекта команды в Инициаторе.
3. **Позднее**... клиент обращается к инициатору с запросом на выполнение команды. Команда, загруженная Инициатором, может как уничтожаться после выполнения, так и оставаться для повторного использования.

Сопоставьте роли и методы из примера с кафе с соответствующими ролями и методами паттерна Команда.

Кафе	Паттерн Команда
Официантка	Команда
Повар	execute()
orderUp()	Клиент
Заказ	Инициатор
Посетитель	Получатель
takeOrder()	setCommand()

Наш первый объект команды

Не пора ли создать первый объект команды? Хотя API для пульта управления домашней техникой еще не спроектирован, построение реализации «снизу вверх» может нам помочь...

Реализация интерфейса Command

Начнем по порядку: все объекты команд реализуют единый интерфейс, который состоит всего из одного метода. В примере с кафе мы назвали этот метод orderUp(), но чаще встречается стандартное имя execute().

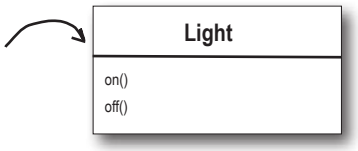
Интерфейс Command выглядит так:

```
public interface Command {
    public void execute();
}
```

Очень простой интерфейс: всего один метод execute().

Реализация команды для включения света

Допустим, вы хотите реализовать команду для включения света. Обратившись к описаниям классов устройств, мы видим, что класс Light содержит два метода: on() и off(). Реализация команды выглядит примерно так:



Класс команды должен реализовать интерфейс Command.

```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }
}
```

В переменной light конструктору передается конкретный объект, которым будет управлять команда (допустим, освещение в гостиной). При вызове execute получателем запроса будет объект light.

Метод execute вызывает метод on() объекта-получателя (то есть осветительной системы).

Итак, у нас имеется класс LightOnCommand. Давайте найдем ему практическое применение...

Использование объекта команды

Упростим исходную задачу: допустим, пульт оснащен всего одной кнопкой и имеет всего одну ячейку для хранения управляемого устройства:

```
public class SimpleRemoteControl {
    Command slot;

    public SimpleRemoteControl() {}

    public void setCommand(Command command) {
        slot = command;
    }

    public void buttonWasPressed() {
        slot.execute();
    }
}
```

Всего одна ячейка для хранения команды (и одно управляемое устройство).

Метод для назначения команды. Может вызываться повторно, если клиент кода захочет изменить поведение кнопки.

Метод, вызываемый при нажатии кнопки. Мы просто берем объект команды, связанный с текущей ячейкой, и вызываем его метод execute().

Создание простого теста

Следующий фрагмент кода поможет нам в тестировании упрощенной версии пульта. Обратите внимание на соответствие между отдельными строками и блоками диаграммы паттерна Команда:

```
public class RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        LightOnCommand lightOn = new LightOnCommand(light);

        remote.setCommand(lightOn);
        remote.buttonWasPressed();
    }
}
```

Клиент в терминологии паттерна.

Объект remote — Инициатор; ему будет передаваться объект команды.

Создание объекта Light, который будет Получателем запроса.

Создание команды с указанием Получателя.

Команда передается Инициатору.

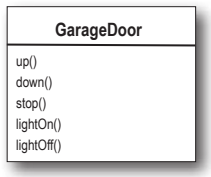
Имитируем нажатие кнопки.

Результат выполнения тестового кода.

```
File  Edit  Window  Help  DinerFoodYum
%java RemoteControlTest
Light is On
%
```

Возьми в руку карандаш

А теперь вы самостоятельно реализуете класс GarageDoorOpenCommand. Для начала вам понадобится диаграмма класса GarageDoor.




```
public class GarageDoorOpenCommand
        implements Command {

}
```

Ваш код

Ваш класс готов — какой результат выведет следующий код? (Подсказка: метод up() класса GarageDoor выводит сообщение «Garage Door is Open».)

```
public class RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        GarageDoor garageDoor = new GarageDoor();
        LightOnCommand lightOn = new LightOnCommand(light);
        GarageDoorOpenCommand garageOpen =
            new GarageDoorOpenCommand(garageDoor);


        remote.setCommand(lightOn);
        remote.buttonWasPressed();
        remote.setCommand(garageOpen);
        remote.buttonWasPressed();
    }
}
```

Здесь впишите свои результаты.

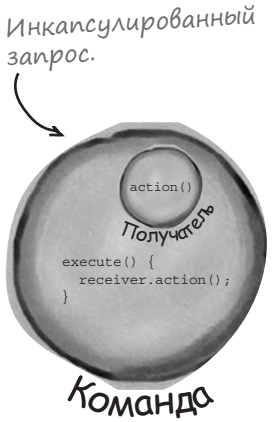
```
File Edit Window Help GreenEggs&Ham

%java RemoteControlTest
```

Определение паттерна Команда

Итак, мы хорошо провели время в кафе Объективиля, частично реализовали API пульта, а попутно составили довольно хорошее представление о том, как организовано взаимодействие классов и объектов в паттерне Команда. Теперь мы определим паттерн Команда и разберемся с подробностями.

Начнем с официального определения:

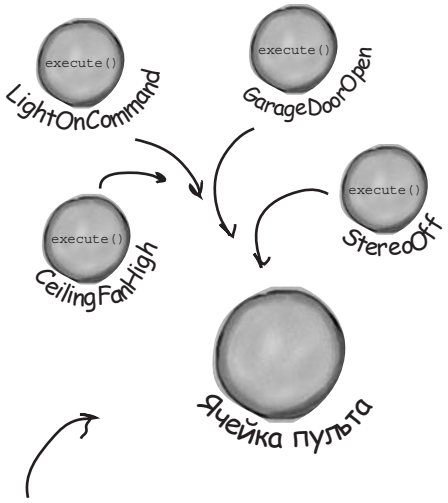
> **Паттерн Команда** инкапсулирует запрос в виде объекта, делая возможной параметризацию клиентских объектов с другими запросами, организацию очереди или регистрацию запросов, а также поддержку отмены операций.



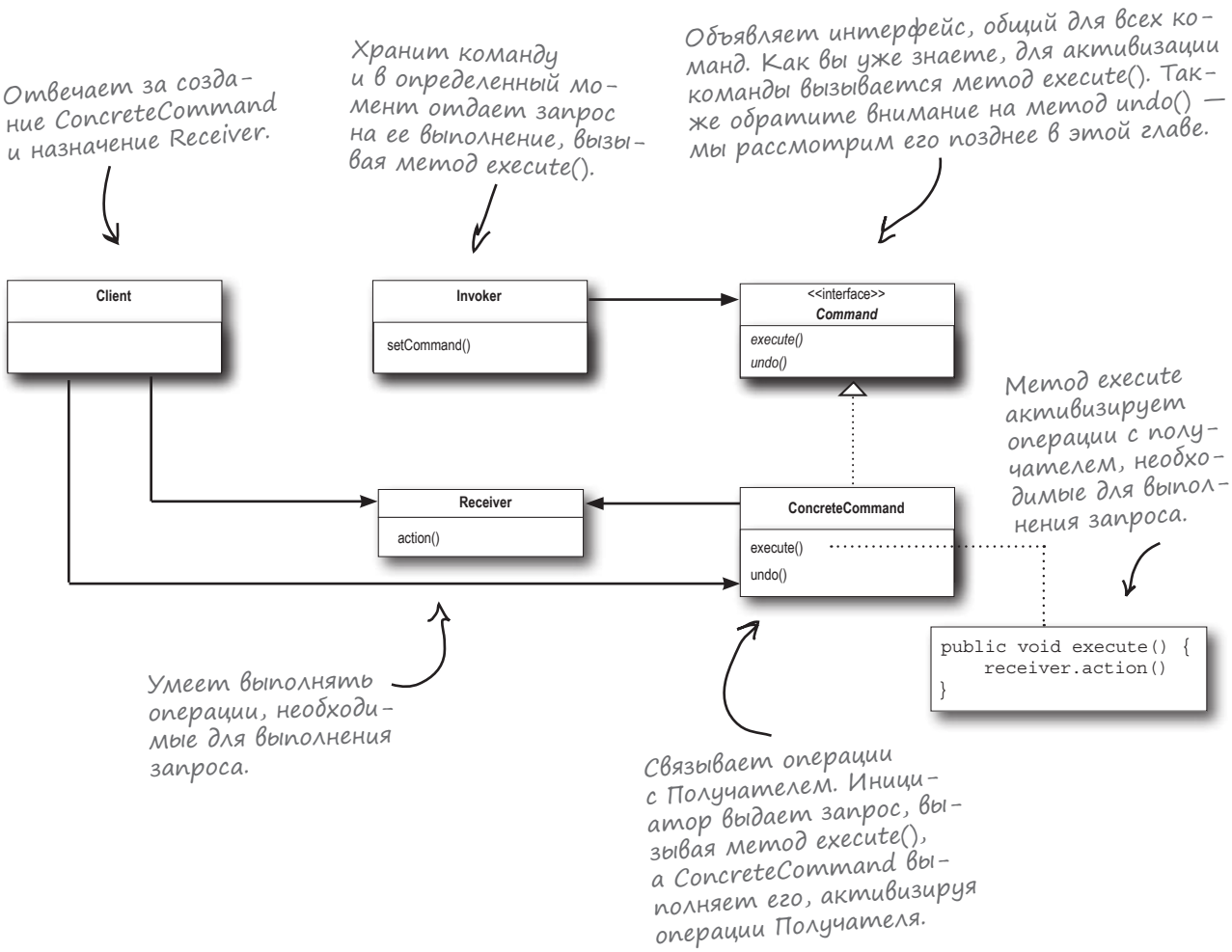
Объект команды *инкапсулирует запрос* посредством привязки набора операций к конкретному получателю. Для этого информация об операции и получателе «упаковывается» в объекте с единственным методом execute(). При вызове метод execute() выполняет операцию с данным получателем. Внешние объекты не знают, какие именно операции выполняются, и с каким получателем; они только знают, что при вызове метода execute() их запрос будет выполнен.

Мы уже рассмотрели пару примеров *параметризации объектов* в командах. В кафе Официантка параметризовалась разными заказами. В упрощенном примере с пультом ячейка сначала связывалась с командой включения света, а потом эта команда заменялась командой открытия двери гаража. Для ячейки пульта (как и для Официантки) совершенно неважно, какой объект команды с ней связан (при условии, что он реализует необходимый интерфейс команды).

С использованием команд для реализации *очередей и регистрации, а также поддержки отмены* мы пока не сталкивались. Не беспокойтесь, это весьма тривиальные расширения базового паттерна Команда. А разобравшись с основной схемой паттерна, вы легко освоите паттерн Метакоманда — механизм создания макропоследовательностей для выполнения серий команд.

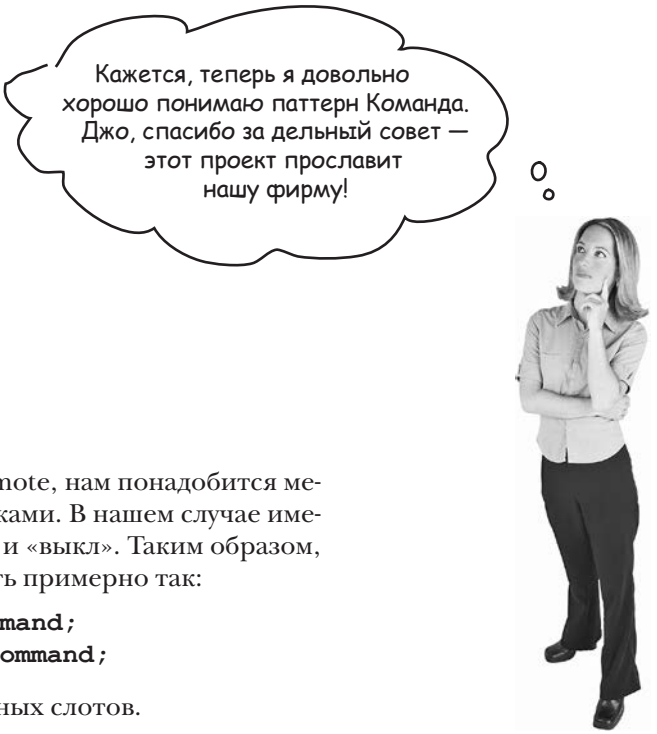


Определение паттерна Команда: диаграмма классов



МОЗГОВОЙ ШТУРМ

Каким образом архитектура паттерна Команда способствует отделению инициатора от получателя запроса?



Мэри: С чего начнем?

Сью: Как и в примере SimpleRemote, нам понадобится механизм связывания команд с ячейками. В нашем случае имеются семь ячеек с кнопками «вкл» и «выкл». Таким образом, назначение команд будет выглядеть примерно так:

```
onCommands[0] = onCommand;
offCommands[0] = offCommand;
```

и т. д. для каждого из семи командных слотов.

Мэри: А как пульт будет отличать освещение в гостиной от освещения на кухне?

Сью: Никак, в этом-то и дело! Пульт не знает ничего, кроме того, что при нажатии кнопки необходимо вызвать execute() для соответствующего объекта команды.

Мэри: Да, это понятно, но я говорю о реализации. Как нам убедиться, что активизируемые с пульта объекты будут включать и выключать правильные устройства?

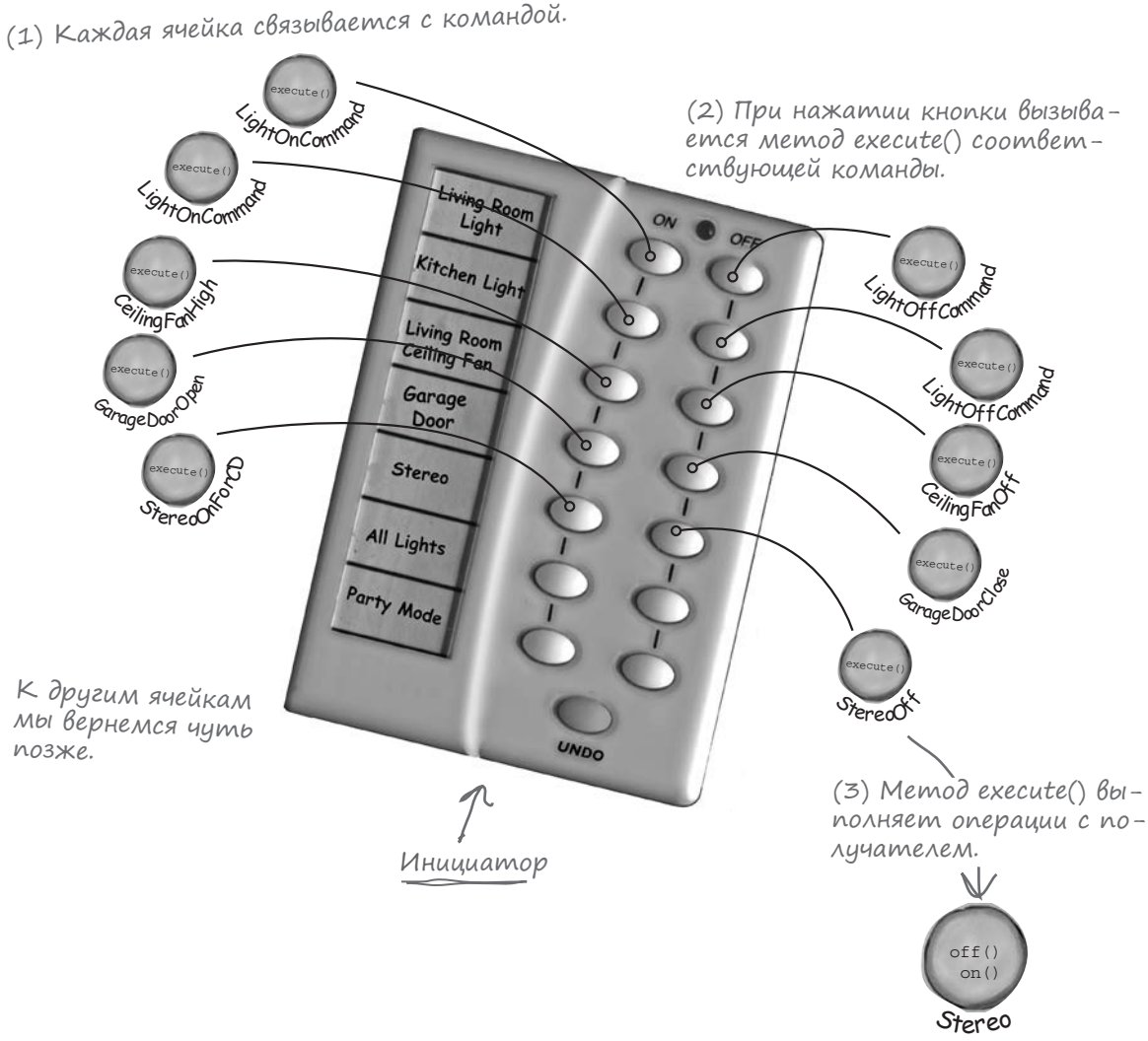
Сью: При создании команд, связываемых с пультом, мы создаем разные команды для осветительных систем в гостиной и на кухне. Напомню, что получатель запроса определяется в команде, в которой он инкапсулируется. Таким образом, в момент нажатия кнопки совершенно неважно, к какой осветительной системе он относится (все необходимое происходит автоматически сразу же после вызова execute()).

Мэри: Кажется, поняла. Давай займемся программированием. Думаю, все прояснится само собой!

Сью: Отличная идея. Давай попробуем...

Связывание команд с ячейками

Мы собираемся связать каждую ячейку пульта с командой. Таким образом, пульту отводится роль *инициатора*. При нажатии кнопки вызывается метод execute() соответствующей команды, что приводит к выполнению операции с получателем (осветительной системой, кондиционером и т. д.).



Реализация пульта

```
public class RemoteControl {
    Command[] onCommands;
    Command[] offCommands;

    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        Command noCommand = new NoCommand();
        for (int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
    }

    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
    }

    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
    }

    public String toString() {
        StringBuffer stringBuff = new StringBuffer();
        stringBuff.append("\n------ Remote Control -------\n");
        for (int i = 0; i < onCommands.length; i++) {
            stringBuff.append("[slot " + i + "] " + onCommands[i].getClass().getName()
                + "    " + offCommands[i].getClass().getName() + "\n");
        }
        return stringBuff.toString();
    }
}
```

В этой версии пульт будет поддерживать все семь команд «вкл/выкл», которые будут храниться в соответствующих массивах.

Конструктор создает экземпляры команд и инициализирует массивы onCommands и offCommands.

Метод setCommand() получает ячейку и команды включения/выключения для этой ячейки. Команды сохраняются в массивах для последующего использования.

При нажатии кнопки «вкл» или «выкл» пульт вызывает соответствующий метод: onButtonWasPushed() или offButtonWasPushed().

Переопределенный метод toString() выводит все ячейки с соответствующими командами. Мы воспользуемся им при тестировании пульта.

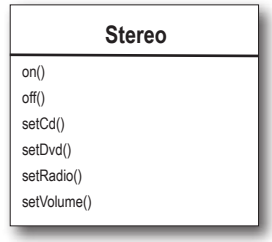
Реализация команд

Мы уже поэкспериментировали с реализацией команды LightOnCommand для версии SimpleRemoteControl. Готовый код будет отлично работать и в новой версии. Реализация команд выключения выглядит практически так же:

```
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.off();
    }
}
```

Команда LightOffCommand почти неотличима от LightOnCommand, если не считать того, что получатель связывается с другой операцией: методом off().

Попробуем что-то более интересное; как насчет команд включения/выключения для стереосистемы? С выключением все просто: объект Stereo связывается с методом off() команды StereoOffCommand. С включением дело обстоит сложнее; предположим, мы хотим написать команду StereoOnWithCDCommand...




```
public class StereoOnWithCDCommand implements Command {
    Stereo stereo;

    public StereoOnWithCDCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    public void execute() {
        stereo.on();
        stereo.setCD();
        stereo.setVolume(11);
    }
}
```

По аналогии с LightOnCommand передается экземпляр стереосистемы, который сохраняется в локальной переменной экземпляра.

Для выполнения этого запроса необходимо вызвать три операции со стереосистемой: включить ее, установить режим воспроизведения CD и установить громкость на уровне 11. Почему именно 11?.. Но ведь 11 лучше 10, верно?

Пока неплохо. Просмотрите остальные классы устройств. Несомненно, к этому моменту вы сможете самостоятельно реализовать остальные классы команд.

Проверяем пульт в деле

Программирование пульта практически завершено, остается лишь провести тестирование и написать документацию с описанием API. Это должно произвести впечатление на заказчика, не правда ли? Нам удалось разработать архитектуру, гибкую и простую в сопровождении, для которой фирмы-разработчики в будущем смогут легко создавать новые классы устройств.

Пора переходить к тестированию кода!

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControl remoteControl = new RemoteControl();

        Light livingRoomLight = new Light("Living Room");
        Light kitchenLight = new Light("Kitchen");
        CeilingFan ceilingFan= new CeilingFan("Living Room");
        GarageDoor garageDoor = new GarageDoor("");
        Stereo stereo = new Stereo("Living Room");

        LightOnCommand livingRoomLightOn =
                new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff =
                new LightOffCommand(livingRoomLight);
        LightOnCommand kitchenLightOn =
                new LightOnCommand(kitchenLight);
        LightOffCommand kitchenLightOff =
                new LightOffCommand(kitchenLight);

        CeilingFanOnCommand ceilingFanOn =
                new CeilingFanOnCommand(ceilingFan);
        CeilingFanOffCommand ceilingFanOff =
                new CeilingFanOffCommand(ceilingFan);

        GarageDoorUpCommand garageDoorUp =
                new GarageDoorUpCommand(garageDoor);
        GarageDoorDownCommand garageDoorDown =
                new GarageDoorDownCommand(garageDoor);

        StereoOnWithCDCommand stereoOnWithCD =
                new StereoOnWithCDCommand(stereo);
        StereoOffCommand  stereoOff =
                new StereoOffCommand(stereo);
```

Создание всех устройств.

Создание команд для управления освещением.

Создание команд для управления потолочным вентилятором.

Создание команд для управления дверью гаража.

Создание команд для управления стереосистемой.

```
        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);
        remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);
        remoteControl.setCommand(2, ceilingFanOn, ceilingFanOff);
        remoteControl.setCommand(3, stereoOnWithCD, stereoOff);

        System.out.println(remoteControl);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(1);
        remoteControl.offButtonWasPushed(1);
        remoteControl.onButtonWasPushed(2);
        remoteControl.offButtonWasPushed(2);
        remoteControl.onButtonWasPushed(3);
        remoteControl.offButtonWasPushed(3);
    }
}
```

Готовые команды связываются с ячейками пульта.

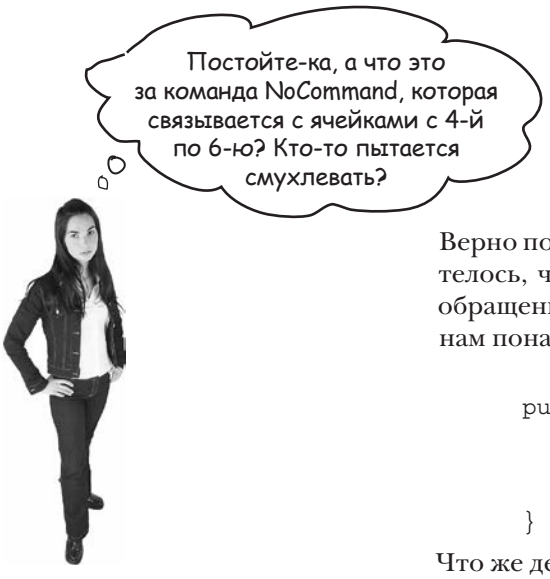
Метод toString() выводит список ячеек и связанных с ними команд.

Пульт готов к проверке! Перебираем все ячейки, и для каждой ячейки имитируем нажатие кнопок «вкл» и «выкл».

Проверяем результаты тестирования...

```
File Edit Window Help CommandsGetThingsDone

% java RemoteLoader
------ Remote Control -------
[slot 0] LightOnCommand             LightOffCommand
[slot 1] LightOnCommand             LightOffCommand
[slot 2] CeilingFanOnCommand        CeilingFanOffCommand
[slot 3] StereoOnWithCDCommand      StereoOffCommand
[slot 4] NoCommand                  NoCommand
[slot 5] NoCommand                  NoCommand
[slot 6] NoCommand                  NoCommand

                    Вкл       Выкл

Living Room light is on
Living Room light is off
Kitchen light is on
Kitchen light is off
Living Room ceiling fan is on high
Living Room ceiling fan is off
Living Room stereo is on
Living Room stereo is set for CD input
Living Room Stereo volume set to 11
Living Room stereo is off

%
```

Наши команды работают! Стоит напомнить, что результаты выполнения команд программируются во внешних классах устройств. Например, при включении освещения соответствующий класс выводит сообщение «Living Room light is on».



Верно подмечено. Мы действительно немного схитрили: нам не хотелось, чтобы код пульта проверял наличие команды при каждом обращении к ячейке. Например, в методе onButtonWasPushed() нам понадобится код следующего вида:

```
public void onButtonWasPushed(int slot) {
    if (onCommands[slot] != null) {
        onCommands[slot].execute();
    }
}
```

Что же делать? Реализовать команду, которая не делает ничего!

```
public class NoCommand implements Command {
    public void execute() { }
}
```

В конструкторе RemoteControl с каждой ячейкой связывается объект NoCommand по умолчанию, и мы знаем, что в каждой ячейке всегда присутствует допустимая команда.

```
Command noCommand = new NoCommand();
for (int i = 0; i < 7; i++) {
    onCommands[i] = noCommand;
    offCommands[i] = noCommand;
}
```

В результатах тестового запуска мы видим ячейки, которые не были связаны с командой, если не считать команды по умолчанию NoCommand, назначенной при создании RemoteControl.

Заслуженный помощник паттернов

Объект NoCommand является примером *пустого* (null) *объекта*. Пустые объекты применяются тогда, когда вернуть «полноценный» объект невозможно, но вам хочется избавить клиента от необходимости проверять **`null`**-ссылки. Так, в нашем примере при отсутствии полноценного объекта, который можно было бы связать с ячейкой пульта, используется суррогатный объект NoCommand с фиктивным методом execute.

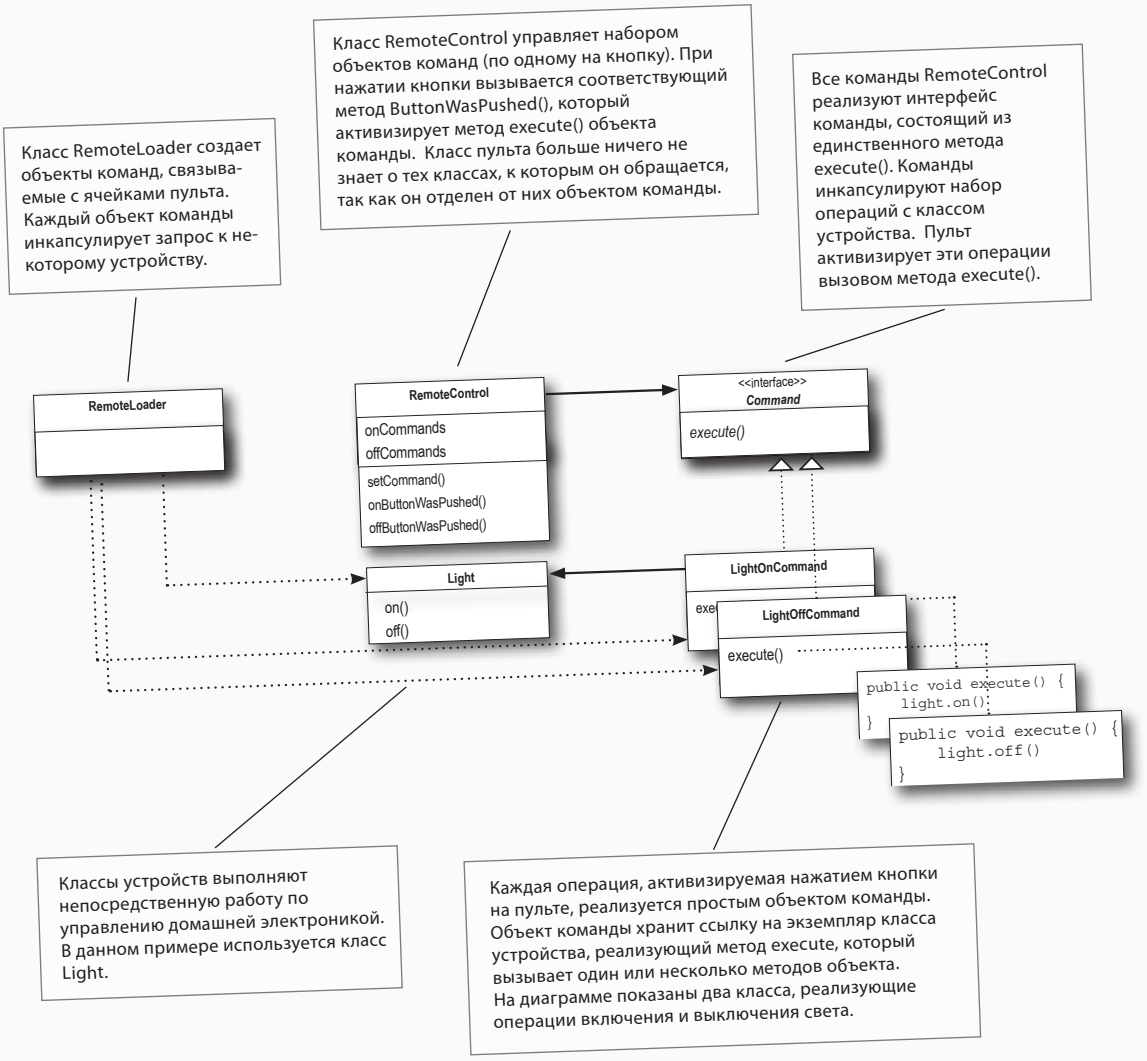
Пустые объекты используются во многих паттернах проектирования, а некоторые авторы даже считают их самостоятельным паттерном.

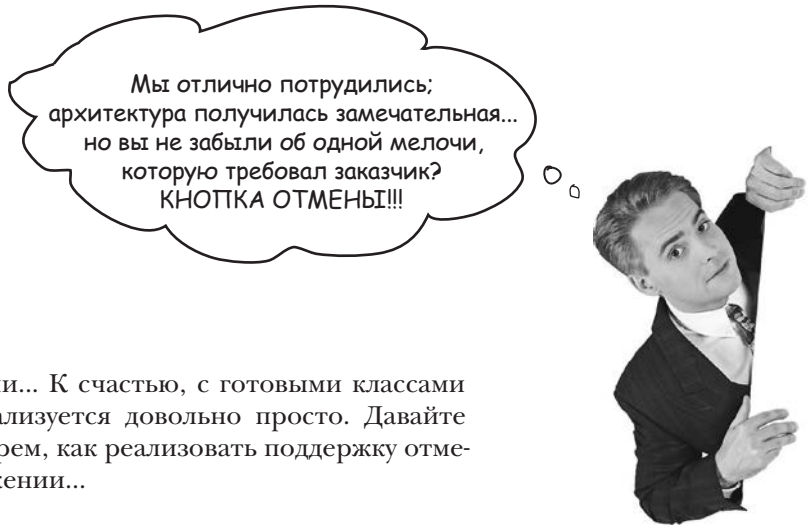
Пора писать документацию...

Архитектура API пульта для фирмы Home Automation or Bust, Inc.

Вашему вниманию представляется следующая архитектура и интерфейс прикладного программирования для пульта управления домашними электронными устройствами. Мы постарались сделать код пульта как можно более простым, чтобы с появлением новых классов в него не приходилось вносить изменения. Для логической изоляции класса пульта от классов устройств был применен паттерн Команда. Мы полагаем, что это приведет к сокращению как затрат на производство пультов, так и к значительному удешевлению сопровождения системы.

Общая схема архитектуры представлена на следующей диаграмме классов:





Стоп! Почти забыли... К счастью, с готовыми классами команд отмена реализуется довольно просто. Давайте шаг за шагом разберем, как реализовать поддержку отмены в нашем приложении...

Что, собственно, нужно сделать?

Нужно реализовать поддержку отмены операций на пульте. Допустим, свет в гостиной выключен, а вы нажимаете на пульте кнопку включения. Разумеется, свет включается. Если теперь нажать кнопку отмены, то последняя операция отменяется — свет выключается. Прежде чем браться за более сложные примеры, разберемся с простейшим случаем:

1. Команды, поддерживающие механизм отмены, должны содержать метод undo(), парный по отношению к методу execute(). Метод undo() отменяет последнюю операцию, выполненную вызовом execute(). Таким образом, перед добавлением функциональности отмены в объекты команд необходимо добавить в интерфейс Command метод undo():

```
public interface Command {
    public void execute();
    public void undo();
}
```

Новый метод undo().

Как видите, ничего сложного.

Теперь мы перейдем к команде включения света и реализуем метод undo().

2 Начнем с класса LightOnCommand: если был вызван метод execute() класса LightOnCommand, значит, последним вызывался метод on(). Чтобы отменить его последствия, метод undo() вызывает противоположный метод off().

```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }

    public void undo() {
        light.off();
    }
}
```

execute() включает свет, поэтому undo() просто выключает его.

Проще простого! На очереди команда LightOffCommand. На этот раз в методе undo() достаточно вызвать метод on() объекта Light.

```
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.off();
    }

    public void undo() {
        light.on();
    }
}
```

Метод undo() снова включает выключенный свет!

Впрочем, это еще не все; нужно включить в класс пульта механизм отслеживания последней нажатой кнопки, а также нажатия кнопки отмены.

❸ Для поддержки отмены достаточно внести в класс RemoteControl несколько незначительных изменений. Мы добавим новую переменную экземпляра для отслеживания последней команды. Далее при нажатии кнопки отмены мы обращаемся к этой команде и вызываем ее метод undo().

```
public class RemoteControlWithUndo {
    Command[] onCommands;
    Command[] offCommands;
    Command undoCommand;

    public RemoteControlWithUndo() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        Command noCommand = new NoCommand();
        for(int i=0;i<7;i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
        undoCommand = noCommand;
    }

    public void setCommand(int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
        undoCommand = onCommands[slot];
    }

    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
        undoCommand = offCommands[slot];
    }

    public void undoButtonWasPushed() {
        undoCommand.undo();
    }

    public String toString() {
        // Код toString...
    }
}
```

Переменная для хранения последней выполненной команды.

В переменную undoCommand изначально также заносится объект NoCommand, чтобы при нажатии кнопки отмены ранее любых других кнопок ничего не происходило.

При нажатии кнопки мы сначала читаем команду и выполняем ее, а затем сохраняем ссылку на нее в переменной undoCommand.

При нажатии кнопки отмены мы вызываем метод undo() команды, хранящейся в переменной undoCommand. Вызов отменяет операцию последней выполненной команды.

Пора протестировать кнопку отмены!

Слегка переработаем тестовую программу, чтобы в ней тестировалась новая функция отмены:

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();

        Light livingRoomLight = new Light("Living Room");

        LightOnCommand livingRoomLightOn =
                new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff =
                new LightOffCommand(livingRoomLight);

        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}
```

Создание объекта Light и новых команд с поддержкой отмены.

Команды размещаются в ячейке 0.

Включение, выключение и отмена.

Выключение, включение и снова отмена.

Результаты тестирования...

```
File Edit  Window  Help  UndoCommandsDefyEntropy
% java RemoteLoader
Light is on
Light is off

------ Remote Control -------
[slot 0] LightOnCommand       LightOffCommand
[slot 1] NoCommand            NoCommand
[slot 2] NoCommand            NoCommand
[slot 3] NoCommand            NoCommand
[slot 4] NoCommand            NoCommand
[slot 5] NoCommand            NoCommand
[slot 6] NoCommand            NoCommand
[undo] LightOffCommand

Light is on

Light is off
Light is on

------ Remote Control -------
[slot 0] LightOnCommand       LightOffCommand
[slot 1] NoCommand            NoCommand
[slot 2] NoCommand            NoCommand
[slot 3] NoCommand            NoCommand
[slot 4] NoCommand            NoCommand
[slot 5] NoCommand            NoCommand
[slot 6] NoCommand            NoCommand
[undo] LightOnCommand

Light is off
```

Включение и выключение.

Команды управления освещением.

Отмена... Метод undo() объекта LightOffCommand снова включает свет.

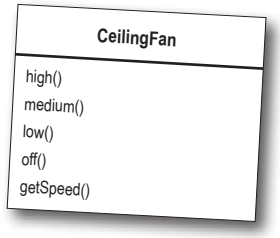
Теперь в undo хранится LightOffCommand — последняя выполненная команда.

Выключаем и снова включаем.

Нажата кнопка отмены, свет снова выключается.

В undo хранится последняя выполнненая команда LightOnCommand.

Реализация отмены с состоянием



Реализация отмены для освещения была поучительной, но слишком тривиальной. Как правило, реализация отмены требует управления состоянием. Рассмотрим более интересный пример — команды управления вентилятором. У вентилятора имеется несколько скоростей вращения, которые задаются соответствующими методами.

Исходный код класса CeilingFan:

```
public class CeilingFan {
    public static final int HIGH = 3;
    public static final int MEDIUM = 2;
    public static final int LOW = 1;
    public static final int OFF = 0;
    String location;
    int speed;

    public CeilingFan(String location) {
        this.location = location;
        speed = OFF;
    }

    public void high() {
        speed = HIGH;
        // Высокая скорость
    }

    public void medium() {
        speed = MEDIUM;
        // Средняя скорость
    }

    public void low() {
        speed = LOW;
        // Низкая скорость
    }

    public void off() {
        speed = OFF;
        // Выключение вентилятора
    }

    public int getSpeed() {
        return speed;
    }
}
```

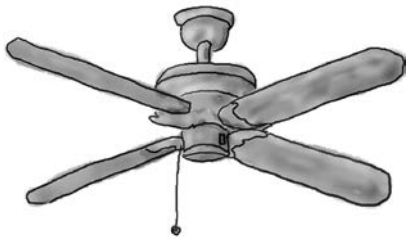
Класс CeilingFan имеет локальную переменную состояния, представляющую скорость вращения вентилятора.

Методы, задающие скорость вращения вентилятора.

Для получения текущей скорости используется метод getSpeed().

Реализация отмены в командах управления вентилятором

Давайте включим поддержку отмены в команды управления вентилятором. Для этого необходимо запомнить предыдущую скорость вращения вентилятора и восстановить сохраненную скорость при вызове метода undo(). Код команды CeilingFanHighCommand:



```
public class CeilingFanHighCommand implements Command {
    CeilingFan ceilingFan;
    int prevSpeed;

    public CeilingFanHighCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }

    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.high();
    }

    public void undo() {
        if (prevSpeed == CeilingFan.HIGH) {
            ceilingFan.high();
        } else if (prevSpeed == CeilingFan.MEDIUM) {
            ceilingFan.medium();
        } else if (prevSpeed == CeilingFan.LOW) {
            ceilingFan.low();
        } else if (prevSpeed == CeilingFan.OFF) {
            ceilingFan.off();
        }
    }
}
```

Добавляем локальную переменную состояния для хранения предыдущей скорости.

В методе execute перед изменением скорости ее предыдущее значение сохраняется для возможной отмены.

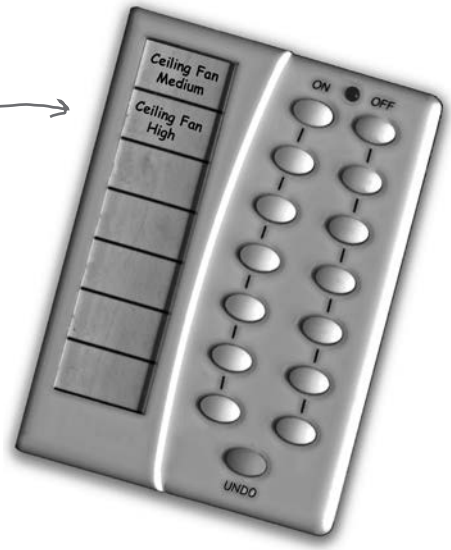
В методе undo() вентилятор возвращается к предыдущей скорости.

МОЗГОВОЙ ШТУРМ

Осталось написать еще три команды управления вентилятором: для низкой скорости, для средней скорости и для выключения. Вы представляете себе их возможную реализацию?

Переходим к тестированию вентилятора

Пора загрузить в пульт новые команды управления вентилятором. Кнопка «вкл» ячейки 0 будет включать вентилятор на средней скорости, а кнопка «вкл» ячейки 1 включает его на высокой скорости. Обе соответствующие кнопки «выкл» просто выключают вентилятор.



Код тестового сценария:

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();

        CeilingFan ceilingFan = new CeilingFan("Living Room");

        CeilingFanMediumCommand ceilingFanMedium =
                new CeilingFanMediumCommand(ceilingFan);
        CeilingFanHighCommand ceilingFanHigh =
                new CeilingFanHighCommand(ceilingFan);
        CeilingFanOffCommand ceilingFanOff =
                new CeilingFanOffCommand(ceilingFan);

        remoteControl.setCommand(0, ceilingFanMedium, ceilingFanOff);
        remoteControl.setCommand(1, ceilingFanHigh, ceilingFanOff);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();

        remoteControl.onButtonWasPushed(1);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}
```

Создаем экземпляры трех команд: для высокой, для средней скорости и для выключения.

Команды включения средней и высокой скорости помещаются в ячейки 0 и 1.

Сначала включаем среднюю скорость.

Потом выключаем вентилятор.

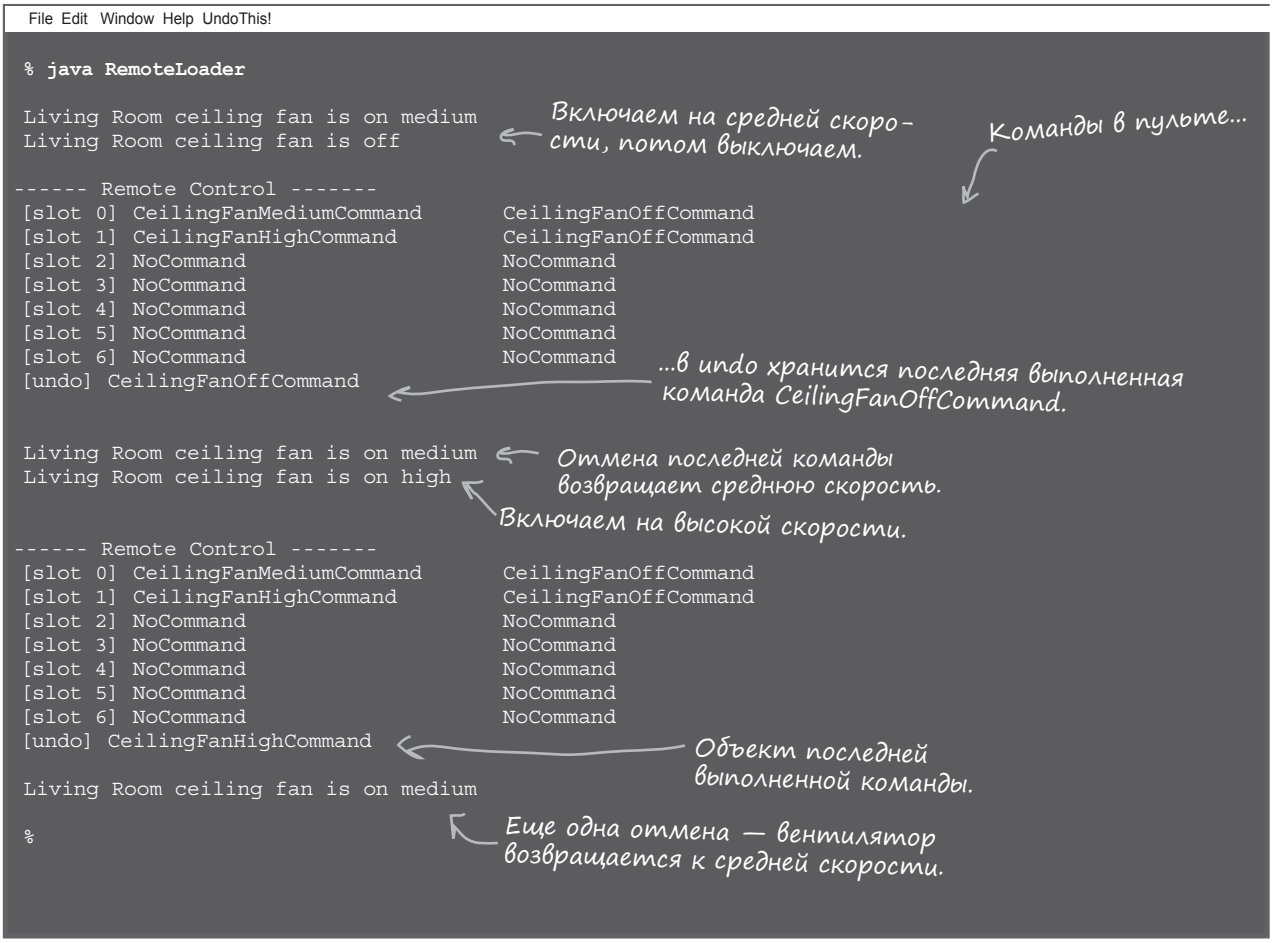
Отмена! Снова должна включиться средняя скорость.

На этот раз выбираем высокую.

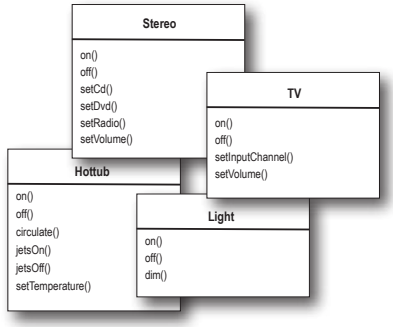
И снова отмена; должна вернуться средняя скорость.

Тестирование...

Берем пульт, загружаем команды — и нажимаем кнопки!



На каждом пульте должен быть Режим Вечеринки!



Какой прок от пульта, если он не способен нажатием одной кнопки выключить свет, включить телевизор и стереосистему, запустить воспроизведение DVD и наполнить джакузи?

Хмм, нашему пульту для каждого устройства нужна отдельная кнопка. Похоже, ничего не выйдет.

Погоди, Сью, не торопись. Я думаю, это можно сделать без изменения кода пульта!

Мэри предлагает создать новую разновидность команд, которая может выполнять другие команды... причем сразу несколько! Классная идея, верно?

```
public class MacroCommand implements Command {
    Command[] commands;

    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }
}
```

Берем массив команд и сохраняем их в объекте MacroCommand.

При выполнении макрокоманды все эти команды будут последовательно выполнены.

Использование макрокоманд

Чтобы использовать макрокоманды в своем приложении, выполняем следующие действия:

1. Сначала создается набор команд, которые войдут в макропоследовательность:

```
Light light = new Light("Living Room");
TV tv = new TV("Living Room");
Stereo stereo = new Stereo("Living Room");
Hottub hottub = new Hottub();

LightOnCommand lightOn = new LightOnCommand(light);
StereoOnCommand stereoOn = new StereoOnCommand(stereo);
TVOnCommand tvOn = new TVOnCommand(tv);
HottubOnCommand hottubOn = new HottubOnCommand(hottub);
```

Создание объектов устройств (свет, телевизор, стерео, джакузи).

Создание команд включения для управления этими устройствами.

Возьми в руку карандаш

Нам понадобятся команды для кнопок выключения. Запишите их в этом поле:

2. Затем мы создаем два массива (включение и выключение), которые заполняются соответствующими командами:

Массивы команд включения и выключения:

```
Command[] partyOn = { lightOn, stereoOn, tvOn, hottubOn};
Command[] partyOff = { lightOff, stereoOff, tvOff, hottubOff};

MacroCommand partyOnMacro = new MacroCommand(partyOn);
MacroCommand partyOffMacro = new MacroCommand(partyOff);
```

...и два объекта макрокоманд, в которых они хранятся.

3. Затем макрокоманда, как обычно, связывается с кнопкой:

```
remoteControl.setCommand(0, partyOnMacro, partyOffMacro);
```

Макрокоманда связывается с кнопкой, как и любая другая команда.

4 А дальше нажимаем кнопки и смотрим, как работает макрокоманда.

```
System.out.println(remoteControl);
System.out.println("--- Pushing Macro On---");
remoteControl.onButtonWasPushed(0);
System.out.println("--- Pushing Macro Off---");
remoteControl.offButtonWasPushed(0);
```

Результат.

```
File  Edit  Window  Help  You Can'tBeatABabka
% java RemoteLoader
------ Remote Control -------
[slot 0] MacroCommand       MacroCommand
[slot 1] NoCommand          NoCommand
[slot 2] NoCommand          NoCommand
[slot 3] NoCommand          NoCommand
[slot 4] NoCommand          NoCommand
[slot 5] NoCommand          NoCommand
[slot 6] NoCommand          NoCommand
[undo] NoCommand

--- Pushing Macro On---
Light is on
Living Room stereo is on
Living Room TV is on
Living Room TV channel is set for DVD
Hottub is heating to a steaming 104 degrees
Hottub is bubbling!

--- Pushing Macro Off---
Light is off
Living Room stereo is off
Living Room TV is off
Hottub is cooling to 98 degrees

%
```

Две макрокоманды.

Все вложенные команды выполняются при вызове макрокоманды включения...

...и выключения. Похоже, все работает.

Упражнение

Нашей макрокоманде не хватает только функциональности отмены. При нажатии кнопки отмены после вызова макрокоманды необходимо отменить действие всех вложенных команд. Перед вами код макрокоманды; напишите реализацию метода undo():

```
public class MacroCommand implements Command {
    Command[] commands;

    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }

    public void undo() {

        // ваш код здесь

    }
}
```

Часто Задаваемые Вопросы

В: Так ли необходим получатель? Почему объект команды не может реализовать всю логику execute()?

О: Как правило, мы стремимся к созданию «простых» объектов команд, которые просто инициируют операцию с получателем. Однако встречаются и «умные» объекты команд, которые реализуют бóльшую часть логики, необходимой для выполнения запроса. Конечно, такой способ тоже возможен, однако следует помнить об ухудшении логической изоляции между инициатором и получателем, а также о потере возможности параметризации команд с разными получателями.

В: Как реализовать историю отмены? Иначе говоря, я хочу, чтобы кнопку отмены можно было нажимать многократно.

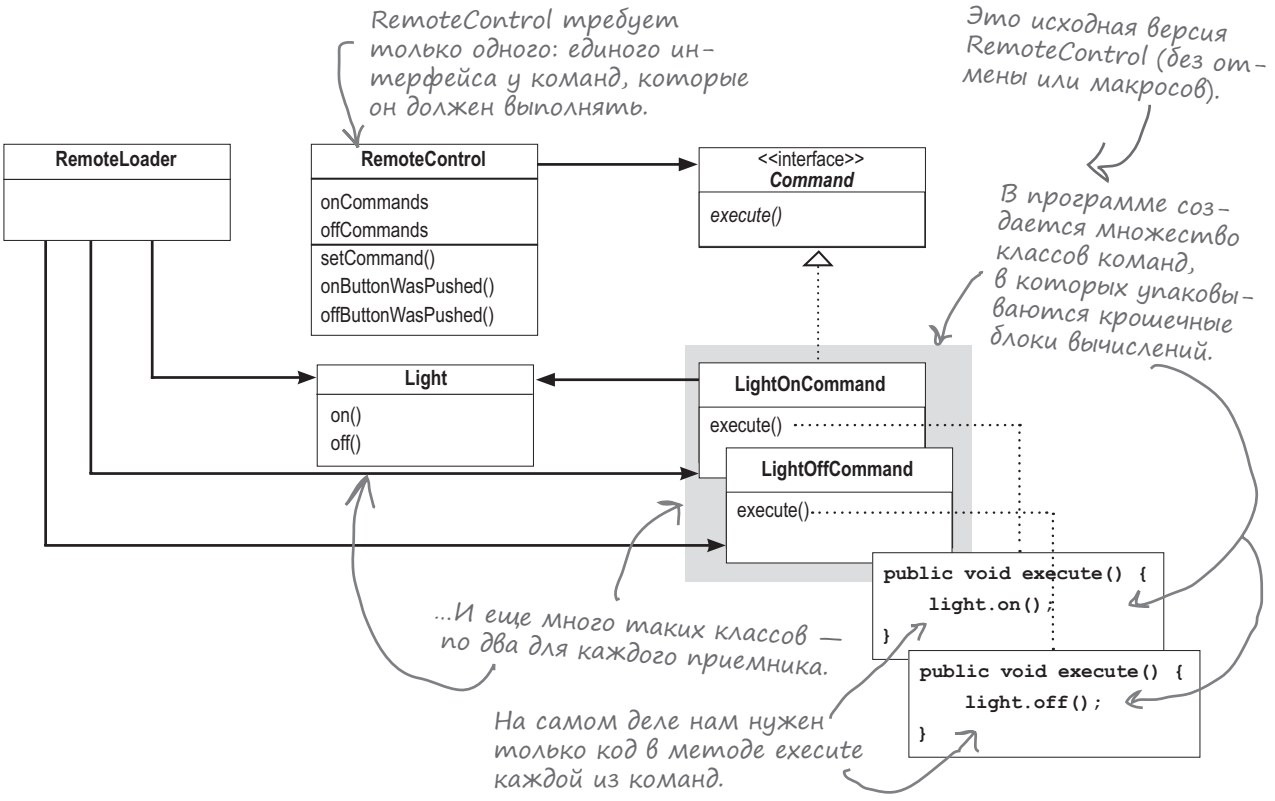
О: Отличный вопрос! На самом деле это несложно; вместо ссылки на последнюю выполненную команду необходимо хранить стек предыдущих команд. При нажатии кнопки отмены инициатор извлекает верхнюю команду из стека и вызывает метод undo().

В: Нельзя ли реализовать макрокоманду как обычную команду — создать объект PartyCommand и разместить вызовы других команд в методе execute() объекта PartyCommand?

О: Можно; но фактически это означает жестко фиксированную реализацию PartyCommand. Зачем идти на это? Макрокоманды позволяют динамически выбирать наборы команд, включаемые в PartyCommand, и обладают большей гибкостью. В общем случае решения на основе макрокоманд более элегантны и требуют меньшего объема кода.

Паттерн Команда означает множество классов команд

При использовании паттерна Команда в программе появляется множество мелких классов — конкретных реализаций Command; каждый класс инкапсулирует запрос к соответствующему приемнику. В нашей реализации пульта для каждого класса приемника создаются два класса команд. Например, для приемника Light создаются классы LightOnCommand и LightOffCommand; для приемника GarageDoor — классы GarageDoorUpCommand и GarageDoorDownCommand, и так далее. Получается слишком много лишнего для создания небольших блоков вычислений, которые имеют единый интерфейс к RemoteControl:



Так ли необходимы все эти классы команд?

Команда — не более чем упакованный блок вычислений. Команды позволяют определить общий интерфейс для поведения многих разных приемников (осветительных систем, джакузи, стереосистем), каждый из которых обладает собственным набором действий.

А теперь представьте, что вы можете оставить общий интерфейс для всех команд, но вынести все вычисления из конкретных реализаций Command и использовать их напрямую. При этом вы избавляетесь от всех лишних классов, а код упрощается. Что ж, с лямба-выражениями это возможно. Давайте посмотрим, как это делается...

Упрощение кода RemoteControl с лямбда-выражениями

Если вы еще не знакомы с лямбда-выражениями (они были добавлены в Java 8), то вряд ли сразу привыкнете к ним. Вероятно, несколько ближайших страниц вам будут понятны, но если потребуется — обращайтесь к справочнику Java за информацией о синтаксисе и семантике лямбда-выражений.

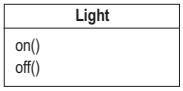
Как вы уже видели, паттерн Команда достаточно прямолинеен. Тем не менее, языка Java предоставляет удобный инструмент, который позволяет еще сильнее упростить код — а именно лямбда-выражения. Лямбда-выражение представляет собой сокращенную форму записи метода (как последовательности вычислений) точно в том месте, где она используется. Вместо того, чтобы создавать отдельный класс с методом, создавать экземпляр этого класса, а затем вызывать метод, вы просто указываете: «Вот метод, который нужно вызвать» при помощи лямбда-выражения. В нашем случае вызываться должен метод execute().

Чтобы понять, как это делается, заменим объекты LightOnCommand и LightOffCommand лямбда-выражениями. Выполните следующие действия, чтобы команды включения/выключения света реализовались лямбда-выражениями вместо объектов команд:

Шаг 1. Создание приемника

Этот шаг нисколько не изменился.

```
Light livingRoomLight = new Light("Living Room");
```



Шаг 2. Реализация команд пульта с использованием лямбда-выражений

Здесь происходит все самое интересное. Теперь вместо того, чтобы создавать объекты LightOnCommand и LightOffCommand для передачи remoteControl.setCommand(), мы просто передаем вместо каждого объекта лямбда-выражение с кодом из соответствующего метода execute():

Два лямбда-выражения.

```
remoteControl.setCommand(0, () -> { livingRoomLight.on(); }, () -> { livingRoomLight.off(); } );
```

Лямбда-выражения передаются в качестве команд методу setCommand.

```
public void setCommand(int slot, Command onCommand, Command offCommand) {
    onCommands[slot] = onCommand;
    offCommands[slot] = offCommand;
}
```

Шаг 3. Активация кнопок

Этот шаг тоже не изменился — если не считать того, что при вызове метода onButtonWasPushed(0) команда в слоте 0 представлена объектом функции (созданным при помощи лямбда-выражения).

Когда мы вызываем execute() для команды, этот метод сопоставляется с методом, определяемым лямбда-выражением, который затем выполняется.

```
remoteControl.onButtonWasPushed(0);
```

```
public void onButtonWasPushed(int slot) {
    onCommands[slot].execute();
}
```

В элементе 0 массива onCommands хранится лямбда-выражение, переданное setCommand на шаге 2. Метод execute() сопоставляется с методом в лямбда-выражении, который затем выполняется.

```
() -> { livingRoomLight.on(); }
```

Кого вы пытаетесь надуть? У лямбда-выражения, которое передается методу setCommand, даже нет метода execute. И как будет вызываться метод в лямбда-выражении?

По волшебству, как же еще?

Ладно, мы пошутили... никакого волшебства. Мы используем лямбда-выражения для замены объектов Command, а интерфейс Command содержит всего один метод: execute(). Лямбда-выражение, которое мы используем, должно иметь совместимую сигнатуру — так оно и есть: метод execute() не получает аргументов (как и наше лямбда-выражение) и не возвращает значения (как и наше лямбда-выражение). Компилятор всем доволен.

Лямбда-выражение передается в параметре Command метода setCommand():

```
public interface Command {
    public void execute();
}
```

Нет аргументов... Не возвращает значение.

```
public void setCommand(int slot,
                       Command onCommand,
                       Command offCommand) {
    ...
}
```

```
() -> { livingRoomLight.on(); }
```

Нет аргументов...

Не возвращает значение.

Да, сигнатура лямбда-выражения соответствует методу Command. Подходит!

Компилятор проверяет, что интерфейс Command содержит ровно один метод, соответствующий лямбда-выражению. Так оно и есть: это метод execute().
Затем, когда для этой команды вызывается execute(), будет вызван метод из лямбда-выражения:

```
onCommands[slot].execute();
```

Интерфейс Command содержит только один метод; именно этот метод будет заменен лямбда-выражением.

```
public interface Command {
    public void execute();
}
```

```
() -> { livingRoomLight.on(); }
```

Запомните: если интерфейс параметра, в котором передается лямбда-выражение, содержит один (и только один!) метод, и сигнатура этого метода совместима с сигнатурой лямбда-выражения, все сработает как надо.

Ссылки на методы

Чтобы еще сильнее упростить код, можно воспользоваться ссылками на методы. Если передаваемое лямбда-выражение вызывает всего один метод, вместо лямбда-выражения можно передать ссылку на метод. Это делается так:

```
remoteControl.setCommand(0, livingRoomLight::on, livingRoomLight::off);
```

Ссылка на метод on() объекта livingRoomLight.

Ссылка на метод off() объекта livingRoomLight.

Вместо того чтобы передавать лямбда-выражение, которое вызывает метод on() объекта livingRoomLight, мы *передаем ссылку на сам метод.*

А что, если лямбда-выражение должно выполнять сразу несколько операций?

Иногда лямбда-выражения, используемые для замены объектов Command, не ограничиваются одной операцией. Давайте посмотрим, как заменить объекты stereoOnWithCDCommand и stereoOffCommand лямбда-выражениями. Далее будет приведен полный код RemoteLoader, чтобы вы поняли, как все эти идеи сочетаются друг с другом.

Объект stereoOffCommand выполняет всего одну простую команду:

```
stereo.off();
```

Для этой команды вместо лямбда-выражения можно использовать ссылку на метод `stereo::off`. Но stereoOnWithCDCommand выполняет не одну, а *три* операции:

```
stereo.on();
stereo.setCD();
stereo.setVolume(11);
```

В таком случае использовать ссылку на метод не удастся. Вместо этого придется либо записывать лямбда-выражение во встроенном виде, либо создать его отдельно, присвоить имя, а потом передать методу setCommand() объекта remoteControl по имени. Вариант с созданием отдельного лямбда-выражения и присвоением ему имени выглядит так:

```
Command stereoOnWithCD = () -> {
    stereo.on(); stereo.setCD(); stereo.setVolume(11);
};
remoteControl.setCommand(3, stereoOnWithCD, stereo::off);
```

Это лямбда-выражение выполняет три операции (как и метод execute у stereoOnWithCDCommand).

Лямбда-выражение можно передать по имени.

Обратите внимание: Command используется как тип лямбда-выражения. Лямбда-выражение сопоставляется с методом execute() интерфейса Command и с параметром Command, в котором оно передается методу setCommand().

Тест-драйв команд с использованием лямбда-выражений

Чтобы использовать лямбда-выражения для упрощения кода исходной реализации RemoteControl (без отмены), мы изменим код RemoteLoader и заменим конкретные объекты Command лямбда-выражениями. Также необходимо изменить конструктор RemoteControl для использования лямбда-выражений вместо объекта NoCommand. После этого можно удалить все конкретные классы Command (LightOnCommand, LightOffCommand, HottubOnCommand, HottubOffCommand и т. д.). Вот и все! Все остальное остается прежним. Не удалите случайно интерфейс Command; он необходим для сопоставления типа объектов функций, создаваемых лямбда-выражениями, которые сохраняются в объекте пульта и передаются различным методам.

Новый код класса RemoteLoader выглядит так:

```
public class RemoteLoader {
    public static void main(String[] args) {
        RemoteControl remoteControl = new RemoteControl();

        Light livingRoomLight = new Light("Living Room");
        Light kitchenLight = new Light("Kitchen");
        CeilingFan ceilingFan = new CeilingFan("Living Room");
        GarageDoor garageDoor = new GarageDoor("Main house");
        Stereo stereo = new Stereo("Living Room");

        remoteControl.setCommand(0, livingRoomLight::on, livingRoomLight::off);
        remoteControl.setCommand(1, kitchenLight::on, kitchenLight::off);
        remoteControl.setCommand(2, ceilingFan::high, ceilingFan::off);

        Command stereoOnWithCD = () -> {
            stereo.on(); stereo.setCD(); stereo.setVolume(11);
        };
        remoteControl.setCommand(3, stereoOnWithCD, stereo::off);
        remoteControl.setCommand(4, garageDoor::up, garageDoor::down);

        System.out.println(remoteControl);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(1);
        remoteControl.offButtonWasPushed(1);
        remoteControl.onButtonWasPushed(2);
        remoteControl.offButtonWasPushed(2);
        remoteControl.onButtonWasPushed(3);
        remoteControl.offButtonWasPushed(3);
    }
}
```

Удаляем весь код создания конкретных объектов Command (вместе с самими классами). Код становится куда более компактным (а от 22 классов остается только 9).

Мы используем ссылки на методы повсюду, где используются простые команды из одного метода, и полные лямбда-выражения там, где одного вызова метода недостаточно.

(Ссылку на метод можно рассматривать как компактное лямбда-выражение. По сути это одно и то же; ссылка на метод — просто сокращенная запись для лямбда-выражения, которое вызывает всего один метод.)

И не забудьте: мы должны изменить конструктор RemoteControl, удалить из него код конструирования объектов NoCommand и заменить его лямбда-выражениями.

Ого, в этой реализации от 22 классов осталось всего 9. С ними управляться будет намного проще.

```
public class RemoteControl {
    Command[] onCommands;
    Command[] offCommands;

    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        for (int i = 0; i < 7; i++) {
            onCommands[i] = () -> { };
            offCommands[i] = () -> { };
        }
    }
    // Остальной код
}
```

Удаляем код создания объекта NoCommand

Вместо объекта NoCommand используем лямбда-выражение, которое не делает ничего! (По аналогии с методом execute() в объекте NoCommand, который тоже ничего не делал.)

А теперь проверим результаты всех этих команд с лямбда-выражениями...

File Edit Window Help CommandsGetThingsDone

```
% java RemoteLoader
------ Remote Control -------
[slot 0] RemoteLoader$$Lambda$1/168423058     RemoteLoader$$Lambda$2/1247233941
[slot 1] RemoteLoader$$Lambda$3/258952499     RemoteLoader$$Lambda$4/603742814
[slot 2] RemoteLoader$$Lambda$5/1325547227    RemoteLoader$$Lambda$6/980546781
[slot 3] RemoteLoader$$Lambda$9/1706377736    RemoteLoader$$Lambda$10/1804094807
[slot 4] RemoteControl$$Lambda$1/713338599    RemoteControl$$Lambda$2/1247233941
[slot 5] RemoteControl$$Lambda$1/713338599    RemoteControl$$Lambda$2/1247233941
[slot 6] RemoteControl$$Lambda$1/713338599    RemoteControl$$Lambda$2/1247233941

Living Room light is on
Living Room light is off
Kitchen light is on
Kitchen light is off
Living Room ceiling fan is on high
Living Room ceiling fan is off
Living Room stereo is on
Living Room stereo is set for CD input
Living Room Stereo volume set to 11
Living Room stereo is off

%
```

Вкл

Выкл

Теперь вместо имен классов Command выводится что-то странное. Пользы от такого вывода немного.

И снова команды успешно работают. Только на этот раз они определяются лямбда-выражениями вместо объектов Command.

Часто Задаваемые Вопросы

В: Может ли лямбда-выражение получать параметры или возвращать значение? Или оно всегда должно представлять метод без аргументов без возвращаемого значения?

О: Да, лямбда-выражение может иметь параметры и возвращать значение (вернитесь к главе 2 и посмотрите, как мы используем лямбда-выражение с одним аргументом вместо объекта ActionListener в примере с наблюдателем Swing). Однако правила остаются неизменными: сигнатура лямбда-выражения должна соответствовать сигнатуре одного метода в типе объекта, заменяемого лямбда-выражением. Если вам захочется больше узнать о том, как пишутся лямбда-выражения с параметрами и возвращаемыми значениями (и что делать с типами), обращайтесь к документации Java.

В: Вы все время говорите, что лямбда-выражение должно сопоставляться с методом в интерфейсе с одним — и только одним — методом. Выходит, если интерфейс содержит два метода, использовать лямбда-выражение не удастся?

О: Верно. Интерфейс, содержащий всего один метод — как исходная версия Command (или ActionListener в другом примере) — называется *функциональным интерфейсом*. Лямбда-выражения разрабатывались конкретно для замены методов функциональных интерфейсов, не в последнюю очередь для сокращения объема рутинного кода при большом количестве мелких классов с функциональными интерфейсами. Если интерфейс содержат два метода, он не является функциональным, и заменить его лямбда-выражением не удастся. Подумайте: лямбда-выражение заменяет метод, а не весь объект. Заменить два метода одним лямбда-выражением невозможно.

В: Означает ли это, что мы не сможем использовать лямбда-выражения в реализации RemoteControl с отменой? В этом случае интерфейс Command содержит два метода: execute() и undo().

О: Верно. Не исключено, что вам удастся придумать способ использования лямбда-выражений с отменой (создав два разных типа команд). Но скорее всего, в итоге ваш код будет более сложным, чем если бы вы просто использовали объекты Command, как это было сделано при реализации RemoteControl с отменой ранее в этой главы.

Лямбда-выражения предназначены для использования с функциональными интерфейсами (только с одним методом) для упрощения кода. Если вам приходится искать обходные пути для поддержки таких случаев, как Command с отменой, вероятно, вместо лямбда-выражений стоит поискать другое решение.

В: Почему имена слотов так странно выглядят при выводе RemoteControl?

О: Если вы еще раз взглянете на реализацию метода toString() класса RemoteControl, то увидите, что мы используем getClass() для получения класса объекта Command, а затем метод getName() для получения имени класса, после чего выводим информацию на консоль в строковом виде. Это способ вывода имен классов в разных слотах был удобен, но он не универсален.

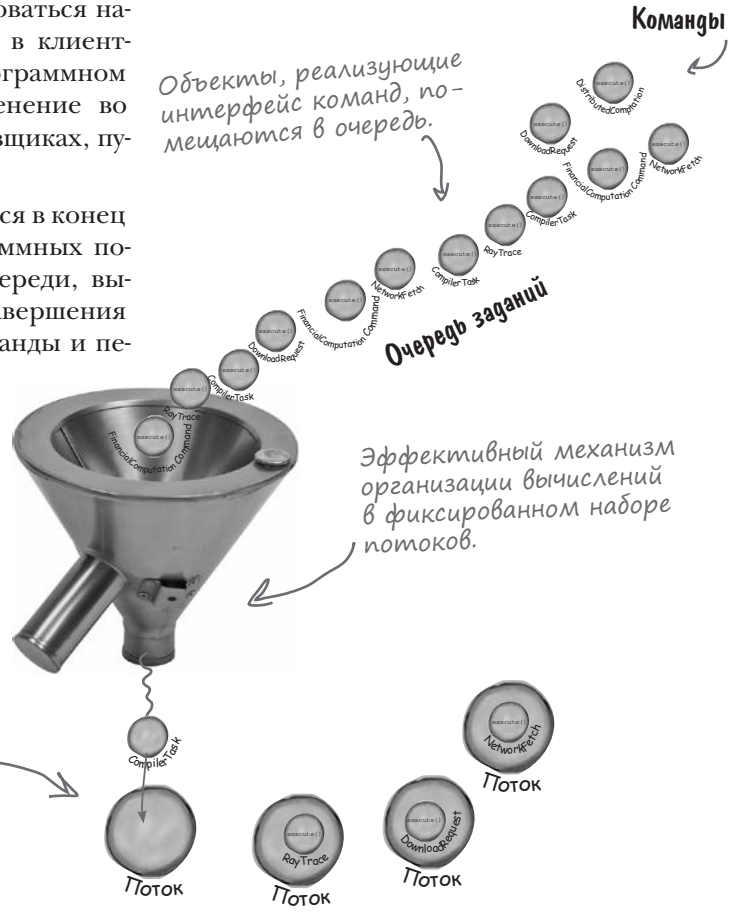
Как видно из вывода, у лямбда-выражений нет удобных имен классов. Дело в том, что исполнительная среда Java назначает им внутренние имена, и Java понятия не имеет, что они собой представляют; для Java это обычные объекты функций, которые просто соответствуют методу в интерфейсе.

Чтобы исправить выходную форму RemoteControl, необходимо изменить код setCommand() в RemoteControl — например, передавать дополнительный параметр с именем, и использовать метод toString() для использования этого имени. Затем в RemoteLoader методу setCommand() наряду с командами будет передаваться симпатичное, понятное для человека имя. Вероятно, такое решение будет лучше отражать ситуацию в реальной жизни (если вы программируете собственный пульт управления, скорее всего, вы захотите присвоить каналам имена по своему усмотрению.)

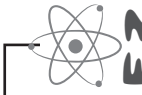
Расширенные возможности паттерна Команда: очереди запросов

Команды обеспечивают механизм инкапсуляции «вычислительных блоков» (получатель + набор операций) и передачи их в виде полноценных объектов. При этом сами операции могут инициироваться намного позже создания объекта команды в клиентском приложении (и даже в другом программном потоке). Этот сценарий находит применение во многих полезных приложениях: планировщиках, пулах потоков, очередях заданий и т. д.

Возьмем очередь заданий: команды ставятся в конец очереди, обслуживаемой группой программных потоков. Потоки извлекают команду из очереди, вызывают ее метод execute(), ожидают завершения вызова, уничтожают текущий объект команды и переходят к следующей команде.



Классы очередей заданий полностью отделены от объектов, выполняющих обработку. В один момент времени поток может выполнять финансовые расчеты, а в другой — загружать данные по сети. Для объекта очереди это совершенно неважно; он просто загружает объекты команд и вызывает их методы execute(). Если очередь реализована на базе паттерна Команда, метод execute() помещенного в нее объекта будет вызван при наличии свободного потока.

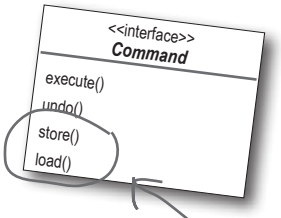


МОЗГОВОЙ ШТУРМ

Как использовать такую очередь в веб-сервере? Какие еще практические применения вы могли бы предложить?

Расширенные возможности паттерна Команда: регистрация запросов

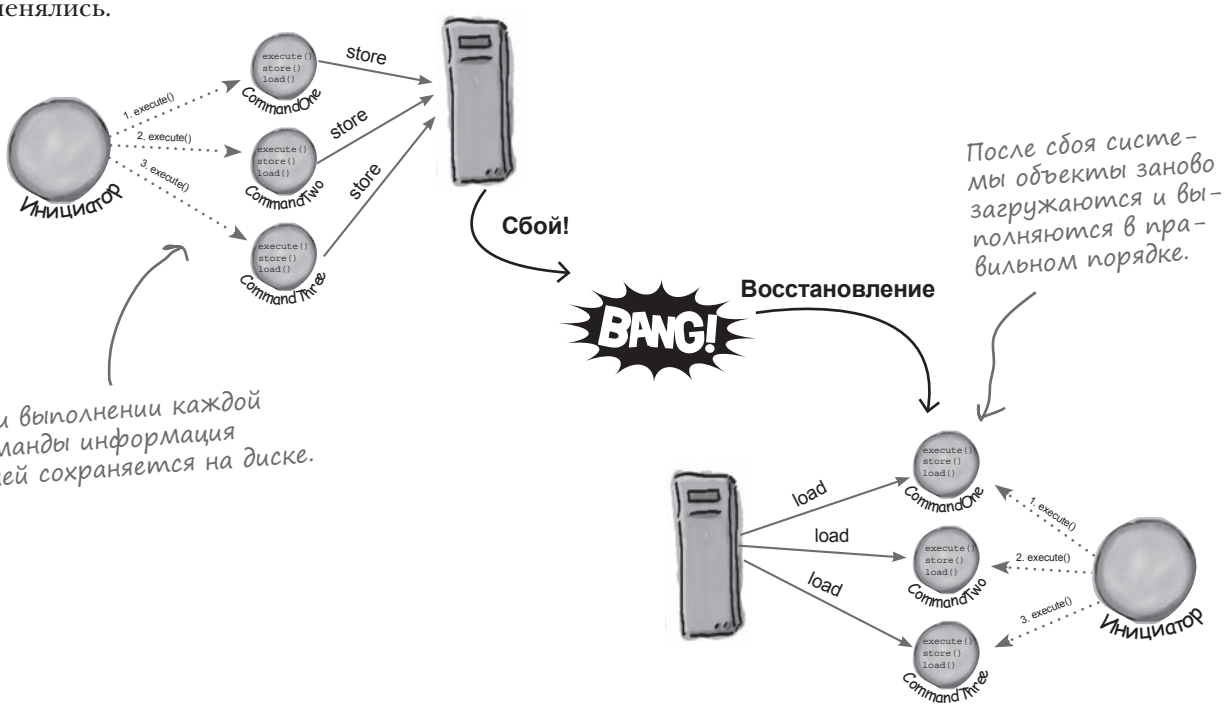
Семантика некоторых приложений требует регистрации всех выполняемых операций и возможности восстановления после сбоя. В паттерне Команда для поддержки этой семантики используются два метода: store() и load(). В языке Java в реализации этих методов можно воспользоваться средствами сериализации объектов, но тогда приходится учитывать стандартные ограничения сериализации.



Два метода для поддержки регистрации.

Как работает регистрация? Информация о выполняемых командах сохраняется в журнальном файле на диске. Если в работе приложения происходит сбой, мы снова загружаем объекты команд и последовательно выполняем их методы execute().

В нашем примере с пультом такая регистрация не имеет смысла; однако многие приложения работают с большими структурами данных, которые невозможно быстро сохранить при каждом изменении. Регистрация позволяет сохранить все операции от последней контрольной точки и в случае возникновения сбоя — применить их к контрольной точке. Скажем, в электронной таблице разумно реализовать восстановление посредством регистрации операций (вместо того, чтобы записывать копию таблицы на диск при каждом изменении). В более сложном приложении возможно транзакционное расширение механизма регистрации, чтобы все операции либо закреплялись как единое целое, либо одновременно отменялись.

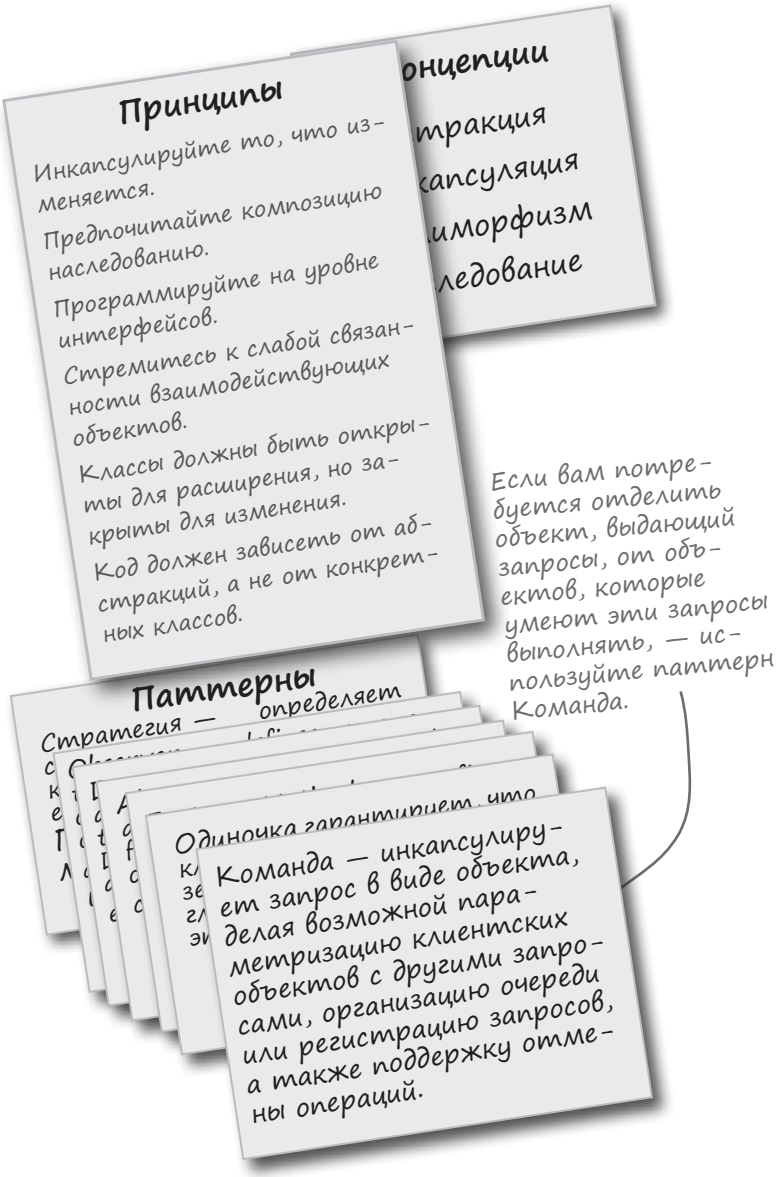


При выполнении каждой команды информация о ней сохраняется на диске.

После сбоя системы объекты заново загружаются и выполняются в правильном порядке.

Новые инструменты

Ваш инструментарий постепенно расширяется! В этой главе он пополнился паттерном, позволяющим инкапсулировать методы в объектах Команды: сохранять их, передавать и активизировать по мере необходимости.

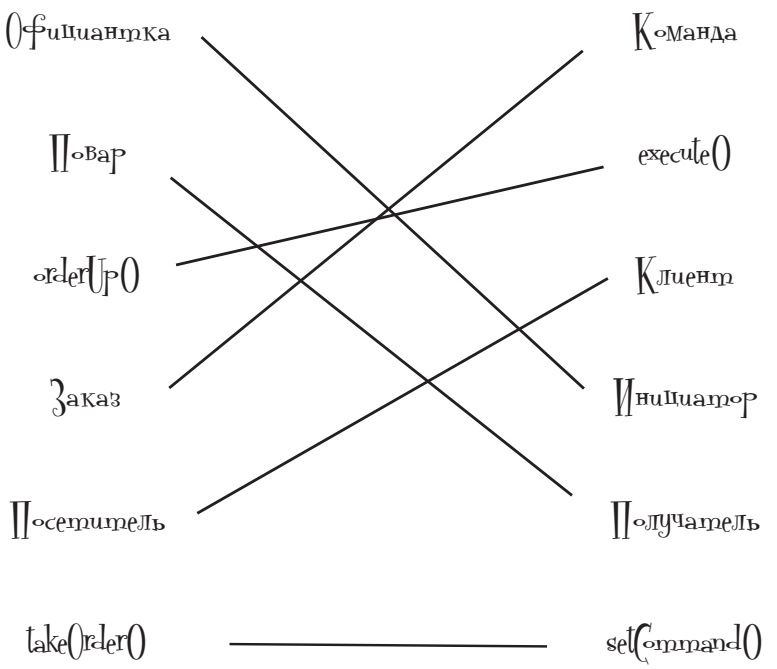


КЛЮЧЕВЫЕ МОМЕНТЫ

- Паттерн Команда отделяет объект, выдающий запросы, от объекта, который умеет эти запросы выполнять.
- Объект команды инкапсулирует получателя с операцией (или набором операций).
- Инициатор вызывает метод execute() объекта команды, что приводит к выполнению соответствующих операций с получателем.
- Возможна параметризация инициаторов командами (даже динамическая во время выполнения).
- Команды могут поддерживать механизм отмены, восстанавливающий объект в состоянии до последнего вызова метода execute().
- Макрокоманды — простое расширение паттерна Команда, позволяющее выполнять цепочки из нескольких команд. В них также легко реализуется механизм отмены.
- На практике нередко встречаются «умные» объекты команд, которые реализуют запрос самостоятельно вместо его делегирования получателю.
- Команды также могут использоваться для реализации систем регистрации команд и поддержки транзакций.

Кто и что делает? Решение

Сопоставьте роли и методы из примера с кафе с соответствующими ролями и методами паттерна Команда.



Возьми в руку карандаш
Решение

```
public class GarageDoorOpenCommand implements Command {
    GarageDoor garageDoor;
    public GarageDoorOpenCommand(GarageDoor garageDoor) {
        this.garageDoor = garageDoor;
    }

    public void execute() {
        garageDoor.up();
    }
}
```

File Edit Window Help GreenEggs&Ham

```
%java RemoteControlTest
Light is on
Garage Door is Open

%
```

Упражнение
Решение

Напишите реализацию метода undo() для макрокоманды.

```
public class MacroCommand implements Command {
    Command[] commands;
    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }

    public void undo() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].undo();
        }
    }
}
```

Возьми в руку карандаш
Решение

Нам также понадобятся команды для кнопок выключения. Запишите их в этом поле:

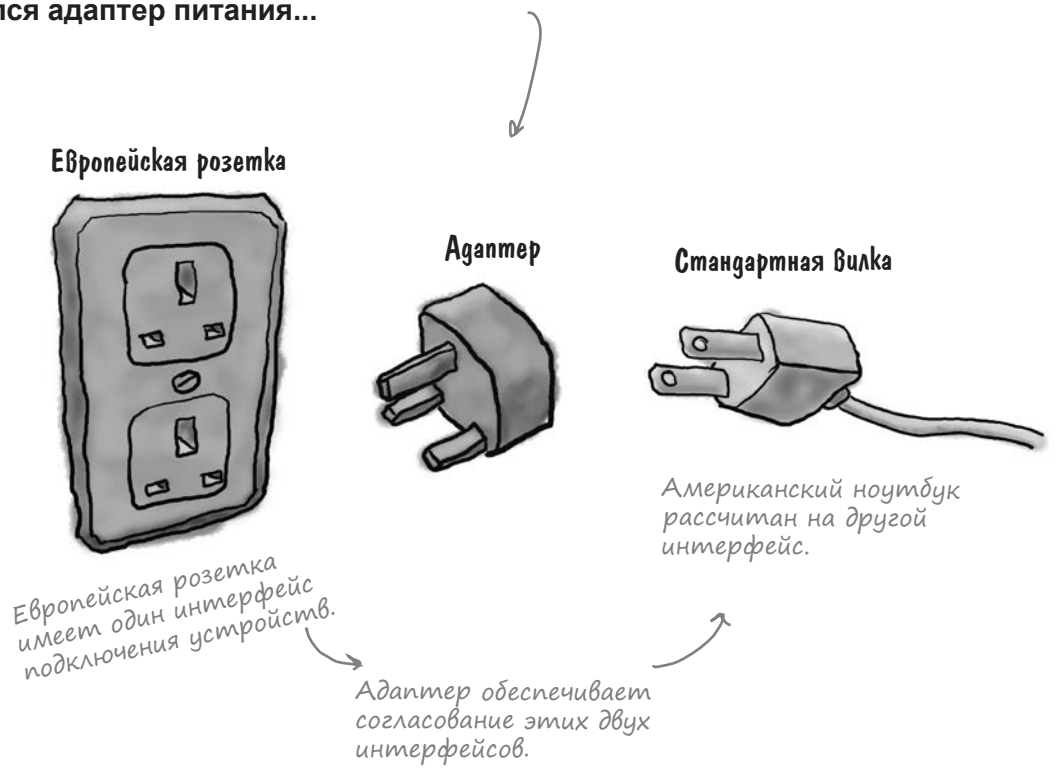
```
LightOffCommand lightOff = new LightOffCommand(light);
StereoOffCommand stereoOff = new StereoOffCommand(stereo);
TVOffCommand tvOff = new TVOffCommand(tv);
HottubOffCommand hottubOff = new HottubOffCommand(hottub);
```

Умение приспосабливаться

В этой главе мы займемся всякими невозможными трюками — будем затыкать круглые дырки квадратными пробками. Невозможно, скажете вы? Только не с паттернами проектирования. Помните паттерн Декоратор? Мы «упаковывали» объекты, чтобы расширить их возможности. А в этой главе мы займемся упаковкой объектов с другой целью: чтобы имитировать интерфейс, которым они в действительности не обладают. Для чего? Чтобы адаптировать архитектуру, рассчитанную на один интерфейс, для класса, реализующего другой интерфейс. Но и это еще не все; попутно будет описан другой паттерн, в котором объекты упаковываются для упрощения их интерфейса.

Адаптеры вокруг нас

Вы без труда поймете концепцию ОО-адаптера, потому что в реальном мире вокруг нас полно адаптеров. Простейший пример: вам доводилось подключать компьютер, выпущенный в США, к европейскому источнику питания? Скорее всего, вам понадобился адаптер питания...



Адаптер включается между вилкой ноутбука и розеткой европейского стандарта; он адаптирует европейскую розетку, чтобы вы могли подключить к ней свое устройство и пользоваться ей. Или можно сказать иначе: адаптер приводит интерфейс розетки к интерфейсу, на который рассчитан ваш ноутбук.

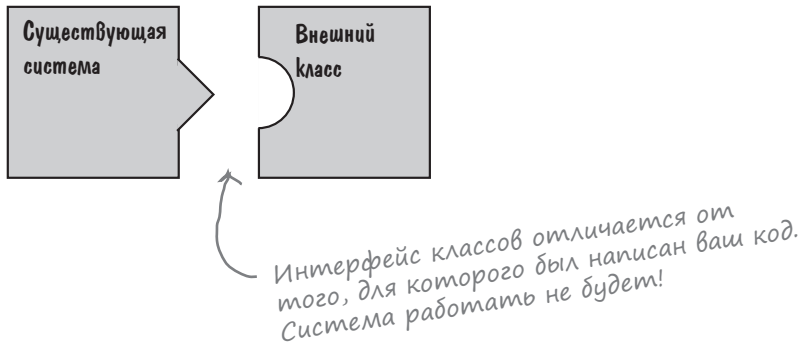
Какие еще реально существующие адаптеры вам известны?

Некоторые адаптеры предельно просты — они изменяют только форму разъема, чтобы она соответствовала форме вилки, а напряжение остается неизменным. Другие адаптеры устроены сложнее, им приходится повышать или понижать напряжение в соответствии с потребностями устройства.

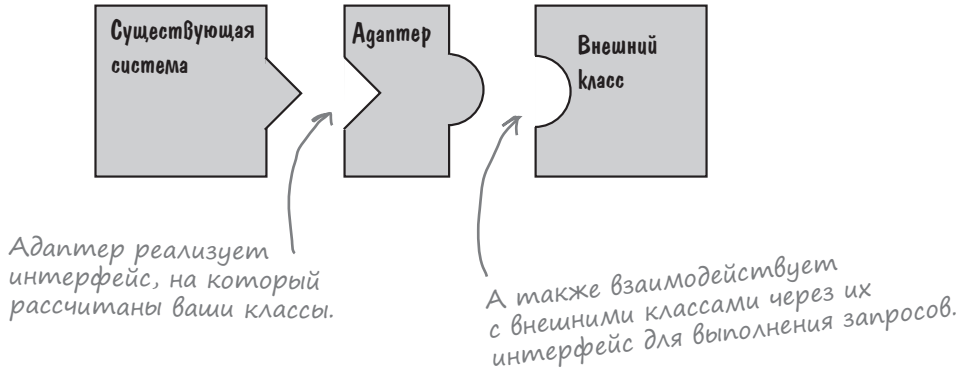
С реальным миром понятно, а как насчет объектно-ориентированных адаптеров? Они играют ту же роль, что и их прототипы в реальном мире: адаптеры преобразуют интерфейс к тому виду, на который рассчитан клиент.

Объектно-ориентированные адаптеры

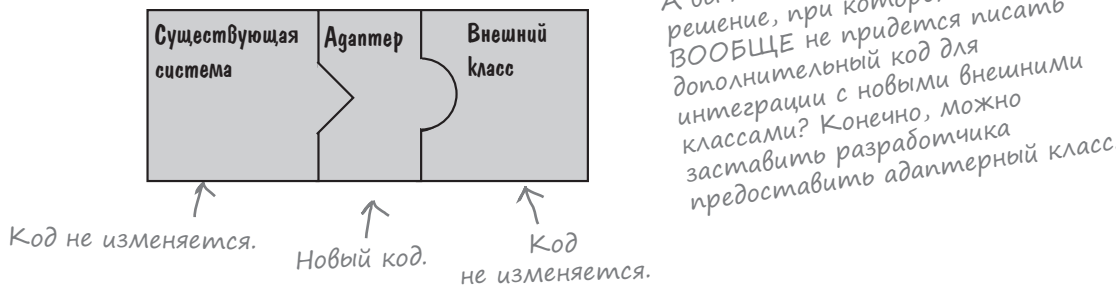
Допустим, у вас имеется готовая программная система, которая должна работать с новой библиотекой внешних классов, однако поставщик библиотеки слегка изменил их интерфейс:



Решать проблему за счет изменения существующего кода не хочется (а код внешних классов недоступен). Что же делать? Напишите класс, который адаптирует интерфейс новых классов к нужному вам.



Адаптер играет роль посредника: он получает запросы от клиента и преобразует их в запросы, понятные внешним классам.



Если кто-то ходит, как утка, и крякает, как утка, то это ~~и есть~~ может быть ~~утка~~ индюшка с утиным адаптером...

Пора взглянуть на адаптеры в действии. Еще не забыли наших уток из главы 1? Давайте рассмотрим слегка упрощенную версию интерфейсов и классов иерархии Duck:

```
public interface Duck {
    public void quack();
    public void fly();
}
```

На этот раз утки реализуют интерфейс Duck.

Конкретный подкласс Duck:

```
public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Простейшие реализации выводят сообщения о выполняемой операции.

А теперь познакомьтесь с новым обитателем птичника:

```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

Индюшки не крякают (у них нет метода quack())...

...но могут летать, хотя и недалеко.

```
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short
distance");
    }
}
```

Конкретная реализация обобщенного класса Turkey: как и класс Duck, она просто выводит описания своих действий.

Допустим, нам не хватает объектов Duck и мы хотим использовать вместо них объекты Turkey. Разумеется, простая замена невозможна, потому что эти объекты обладают разными интерфейсами.

Создаем адаптер:

Код под увеличительным стеклом

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

Прежде всего необходимо реализовать интерфейс того типа, на который рассчитан ваш клиент.

Затем следует получить ссылку на адаптируемый объект; обычно это делается в конструкторе.

Адаптер должен реализовать все методы интерфейса. Преобразование quack() между классами выполняется просто — реализация вызывает gobble().

Хотя метод fly() входит в оба интерфейса, индюшка не умеет летать на дальние расстояния. Чтобы установить соответствие между этими методами, мы вызываем метод fly() класса Turkey пять раз.

Тестирование адаптера

Нам понадобится тестовый код для проверки адаптера:

```
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

Создаем объект Duck...

и объект Turkey.

Затем Turkey упаковывается в TurkeyAdapter и начинает выглядеть, как Duck.

Тестируем методы Turkey.

Теперь вызываем метод testDuck(), который получает объект Duck.

Важный тест: выдаем Turkey за Duck...

Метод testDuck() получает объект Duck и вызывает его методы quack() и fly().

Тест

```
File Edit Window Help Don'tForgetToDuck
%java DuckTestDrive

The Turkey says...

Gobble gobble

I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

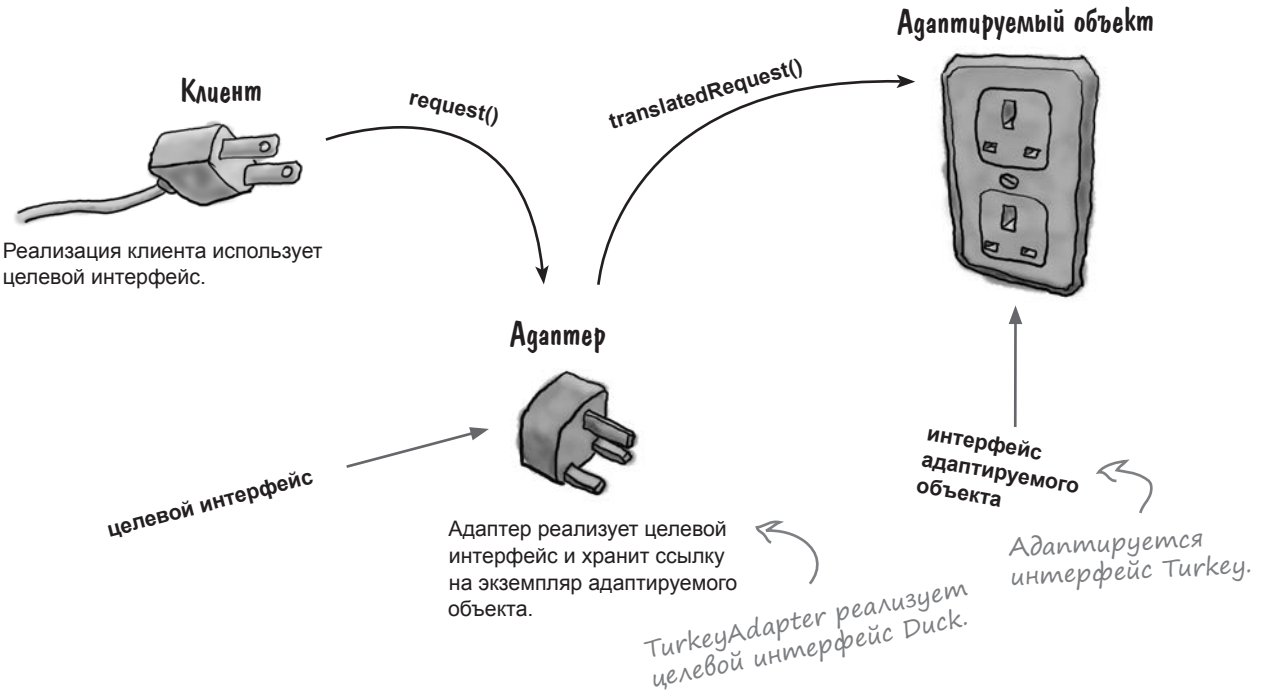
Методы Turkey.

Методы Duck.

Адаптер вызывает gobble() при вызове quack() и выводит повторные сообщения при вызове fly(). Метод testDuck() и не подозревает, что он имеет дело с объектом Turkey, замаскированным под Duck!

Как работает паттерн Адаптер

Теперь, когда вы примерно представляете себе, как работает Адаптер, мы вернемся на пару шагов назад и снова рассмотрим все компоненты.



Как клиент работает с адаптером:

1. **Клиент обращается с запросом к адаптеру, вызывая его метод через целевой интерфейс.**

2. **Адаптер преобразует запрос в один или несколько вызовов к адаптируемому объекту (в интерфейсе последнего).**

3. **Клиент получает результаты вызова, даже не подозревая о преобразованиях, выполненных адаптером.**

Обратите внимание: Клиент полностью изолирован от Адаптера; они ничего не знают друг о друге.

Возьми в руку карандаш

Предположим, вам также понадобился адаптер для преобразования Duck в Turkey — назовем его DuckAdapter. Напишите код этого класса:

Как вы решили проблему с методом fly() (ведь, как известно, утки летают дальше, чем индюшки)? Наше решение приведено в конце главы. Сможете ли вы предложить что-нибудь лучше?

Часто Задаваемые Вопросы

В: Какой объем работы должен выполняться адаптером? Ведь чтобы реализовать большой целевой интерфейс, придется ОСНОВАТЕЛЬНО потрудиться.

О: Сложность реализации адаптера пропорциональна размеру целевого интерфейса. Но подумайте, какой у вас выбор. Конечно, можно переработать все клиентские вызовы к интерфейсу, но это потребует огромной работы по анализу и изменению кода. А можно предоставить всего один класс, который инкапсулирует все изменения.

В: Всегда ли адаптер преобразует один и только один класс?

О: Задача паттерна Адаптер — преобразовать один интерфейс к другому интерфейсу. Хотя в большинстве наших примеров используется один адаптируемый объект, на практике адаптеру иногда приходится хранить два и более объекта, необходимых для реализации целевого интерфейса.

Проблема связана с другим паттерном — Фасадом; эти два паттерна часто путают. Мы еще вернемся к этой теме позднее в этой главе.

В: А если система состоит из новых и старых компонентов? Старые компоненты рассчитаны на старый интерфейс, но мы уже создали новые компоненты для нового интерфейса? Попеременное использование адаптеров и неадаптированных интерфейсов создаст путаницу. Разве не лучше переписать старый код и забыть об адаптере?

О: Не всегда. Вы всегда можете создать Двойной адаптер с поддержкой обоих интерфейсов. Реализуйте оба интерфейса, чтобы адаптер мог использоваться в обеих ролях.

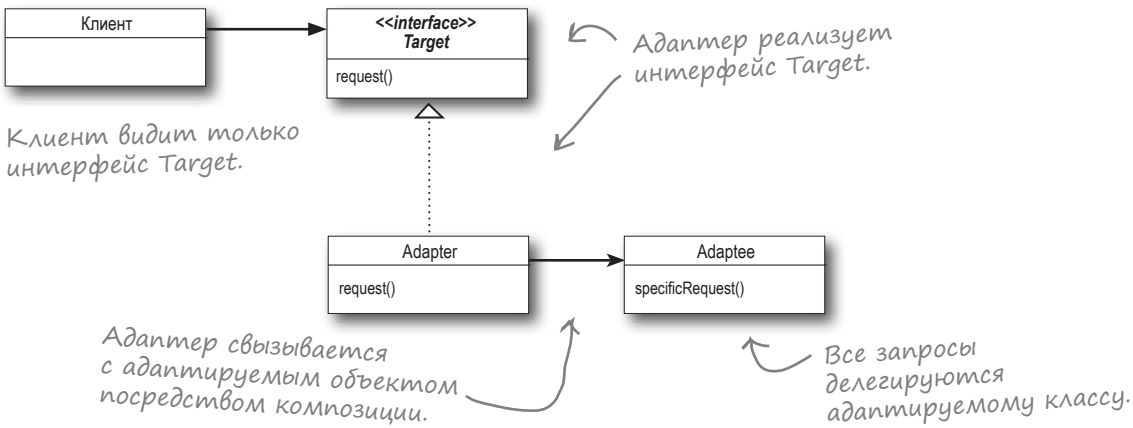
Определение паттерна Адаптер

Довольно уток и индюшек; ниже приведено официальное определение паттерна Адаптер.

> **Паттерн Адаптер** преобразует интерфейс класса к другому интерфейсу, на который рассчитан клиент. Адаптер обеспечивает совместную работу классов, невозможную в обычных условиях из-за несовместимости интерфейсов.

Итак, чтобы использовать клиента с несовместимым интерфейсом, мы создаем адаптер, который выполняет преобразование. Таким образом клиент отделяется от реализованного интерфейса; и если мы ожидаем, что интерфейс будет изменяться со временем, адаптер инкапсулирует эти изменения, чтобы клиента не приходилось изменять каждый раз, когда ему потребуется работать с новым интерфейсом.

Мы проанализировали поведение паттерна в ходе выполнения; давайте также рассмотрим его диаграмму классов:



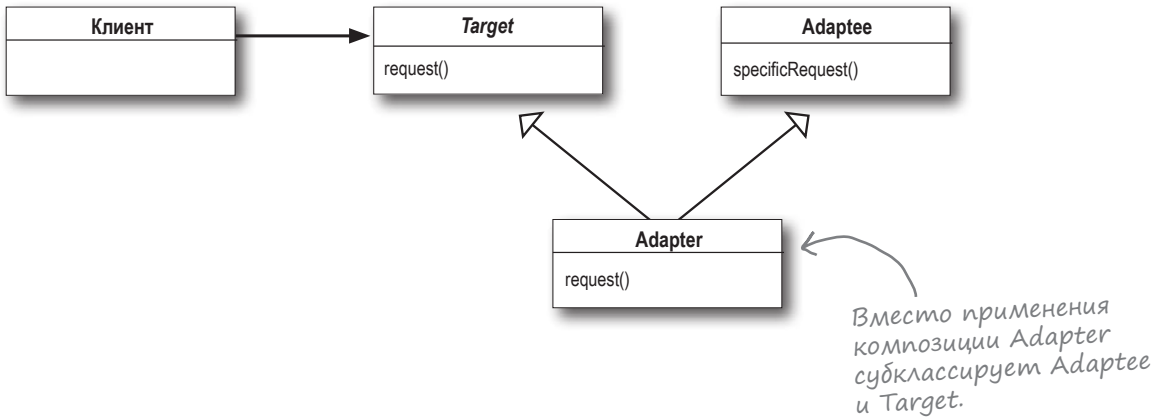
В паттерне Адаптер проявляются многие признаки качественного ОО-проектирования: обратите внимание на использование композиции для «упаковки» адаптируемого объекта в измененный интерфейс. Дополнительное преимущество такого решения заключается в том, что адаптер будет работать с любым субклассом адаптируемого объекта.

Также обратите внимание на то, что паттерн связывает клиента с интерфейсом, а не с реализацией; мы можем использовать несколько адаптеров, каждый из которых выполняет преобразование для своего набора классов. Кроме того, новые реализации могут добавляться позднее. Единственным ограничением является лишь их соответствие интерфейсу Target.

Адаптеры объектов и классов

Мы привели формальное определение паттерна, но еще не рассказали всей истории. На самом деле существует *две* разновидности адаптеров: адаптеры *объектов* и адаптеры *классов*. В этой главе рассматриваются адаптеры объектов, а на диаграмме классов на предыдущей странице также изображен адаптер объектов.

Что же такое адаптер *классов*, и почему мы о них ничего не рассказывали? Потому что для их реализации необходимо множественное наследование, запрещенное в языке Java. Но это не означает, что необходимость в адаптерах классов не возникнет, когда вы будете работать на языке с множественным наследованием! Рассмотрим диаграмму классов с множественным наследованием.



Знакомая картина? Все верно — единственное различие заключается в том, что с адаптером классов мы субклассируем Target и Adaptee, а с адаптером объектов для передачи запросов Adaptee используется механизм композиции.

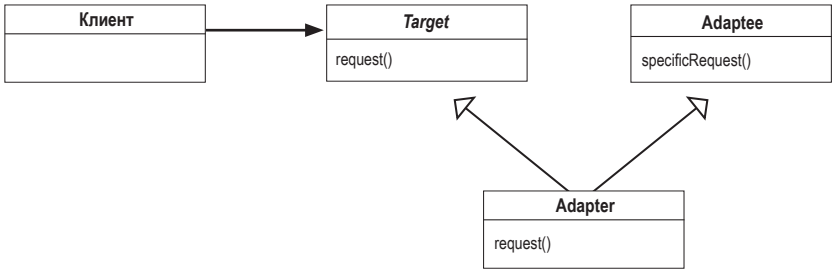
МОЗГОВОЙ ШТУРМ

Адаптеры объектов и адаптеры классов используют два различных способа адаптации (композиция и наследование). Как эти различия в реализации влияют на гибкость адаптера?

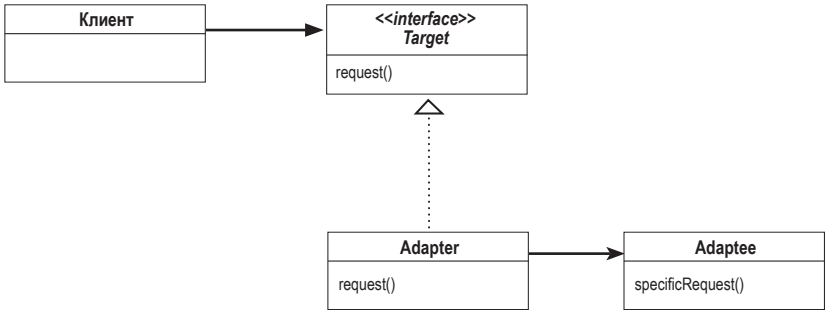
Магниты с утками

Ваша задача — разместить магниты с утками и индюшками на частях диаграммы, описывающих роль этих птиц в приведенном ранее примере. (Постарайтесь не возвращаться на несколько страниц назад.) Добавьте свои примечания по поводу того, как работает эта схема.

Адаптер классов



Адаптер объектов

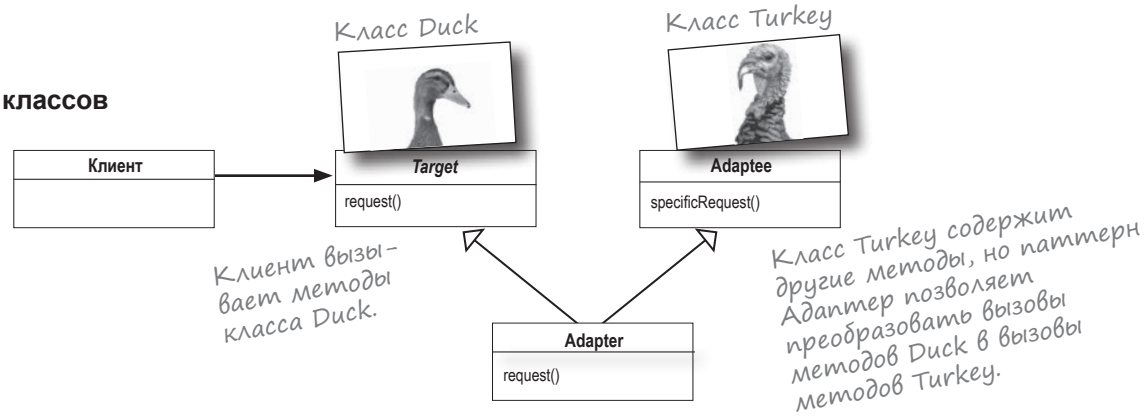


Разместите на диаграмме классов; укажите, какие части диаграммы представляют уток (Duck), а какие — индюшек (Turkey).

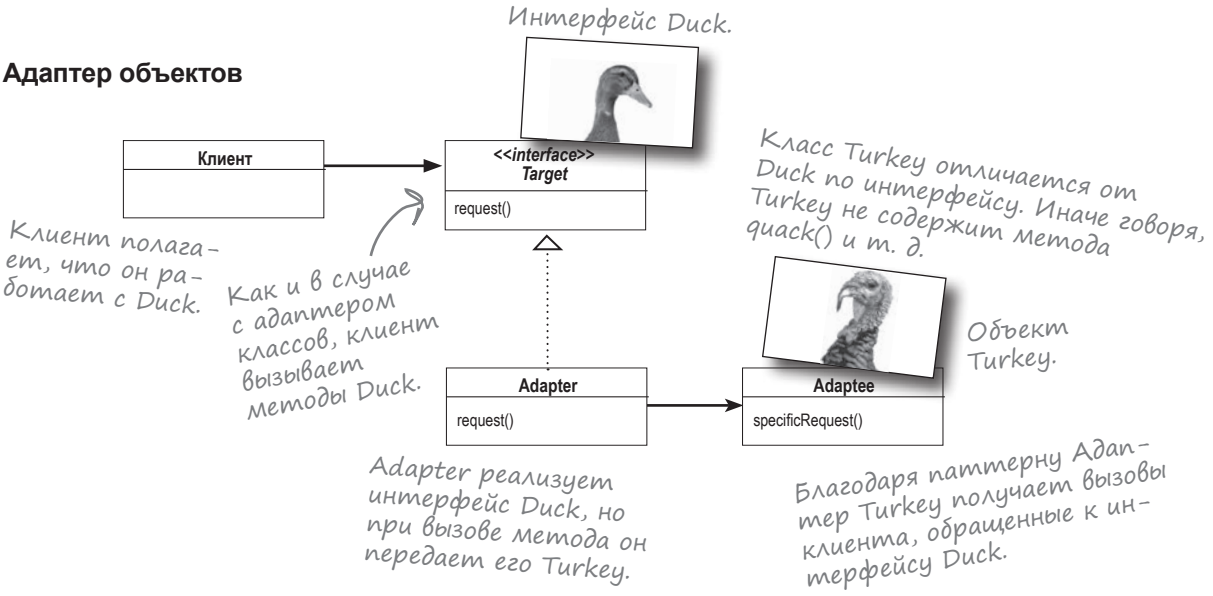
Магниты с утками
Решение

Адаптер классов использует множественное наследование, поэтому в Java такое решение невозможно...

Адаптер классов



Адаптер объектов



Беседа у камина

Адаптер объектов и Адаптер классов встречаются лицом к лицу.

Адаптер объектов

Использование композиции дает мне немалые преимущества. Я могу адаптировать не только отдельный класс, но и все его субклассы.

В моих краях рекомендуется отдавать предпочтение композиции перед наследованием; возможно, получается на несколько строк больше, но мой код просто делегирует вызовы адаптируемому объекту. Мы выбираем гибкость.

Кого волнует один крошечный объект? Ты позволяешь быстро переопределить метод, но поведение, которое я добавляю в код адаптера, работает с моим адаптируемым классом *и* всеми его субклассами.

Подумаешь, нужно включить объект субкласса посредством композиции, и все заработает.

Хочешь увидеть лишние хлопоты? Взгляни в зеркало!

Адаптер классов

Верно, у меня с этим проблемы — я рассчитан на один адаптируемый класс, но зато мне не приходится реализовать его заново. А если потребуется, я могу переопределить поведение адаптируемого класса, ведь речь идет о простом субклассировании.

Гибкость — возможно. Эффективность? Нет. Для моей работы необходим лишь один экземпляр меня самого, а лишние экземпляры адаптера и адаптируемого класса не нужны.

Да, а если в субклассе добавится новое поведение? Что тогда?

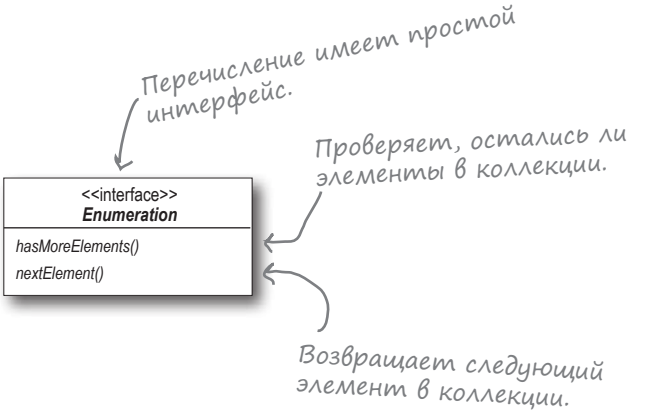
Так ведь лишние хлопоты...

Практическое применение адаптеров

Рассмотрим несколько реальных примеров применения простых адаптеров (по крайней мере более серьезных, чем превращение утки в индюшку)...

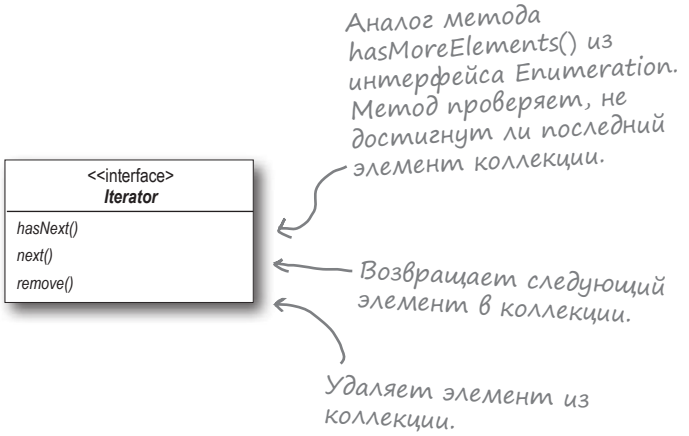
Классические перечисления

Опытные Java-программисты помнят, что ранние реализации коллекций (Vector, Stack, Hashtable и некоторые другие) реализовали метод elements(), который возвращал перечисление (Enumeration). Интерфейс Enumeration позволяет перебирать элементы коллекции, не зная конкретного механизма управления ими в коллекции.



Современные итераторы

В обновленных классах коллекций используется более современный интерфейс итераторов. Как и перечисления, итераторы, позволяют перебирать элементы коллекций, но также обладают возможностью удаления элементов.

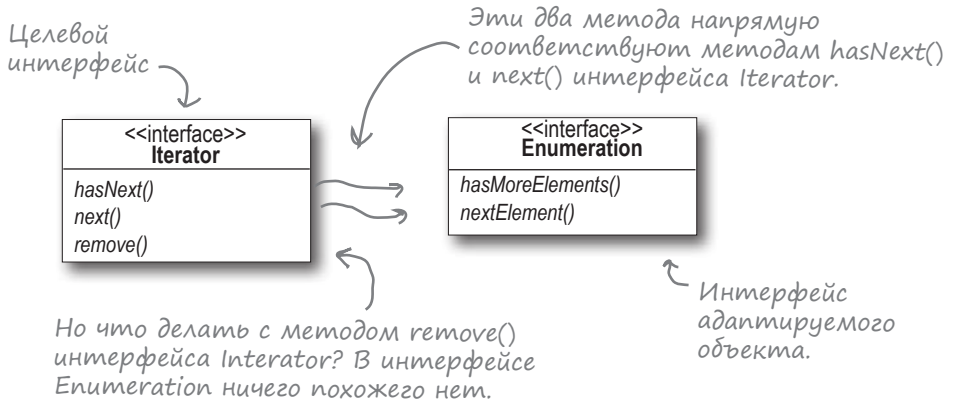


И наконец...

Нам часто приходится иметь дело со старым кодом, поддерживающим интерфейс Enumerator, хотя хотелось бы, чтобы новый код работал только с итераторами. Похоже, придется создать для него адаптер.

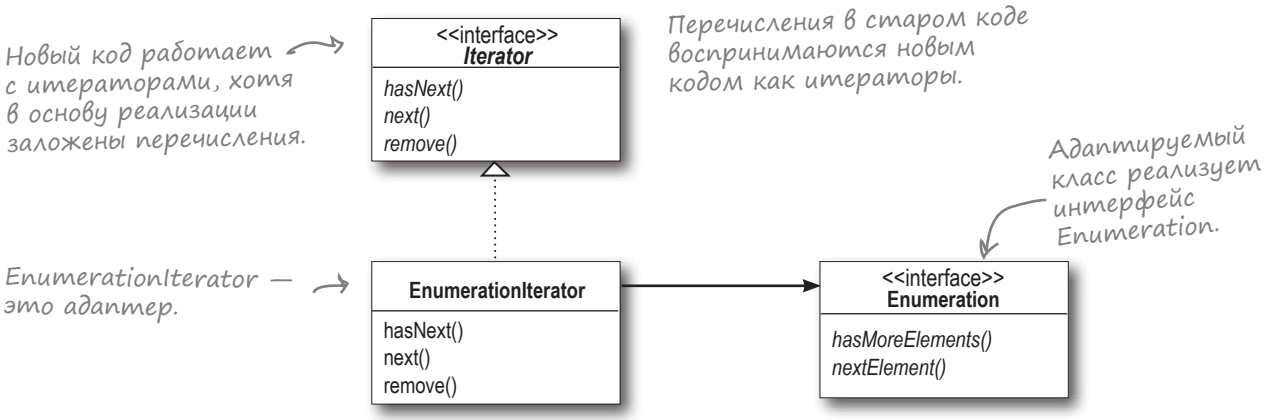
Адаптация перечисления к итератору

Сначала мы сопоставляем два интерфейса, чтобы понять, как их методы соответствуют друг другу. Иначе говоря, нужно понять, какой метод адаптируемого объекта следует использовать при вызове того или иного метода целевого интерфейса.



Проектирование адаптера

Итак, наш адаптер должен реализовать целевой интерфейс, и адаптируемый объект должен включаться в него посредством композиции. Методы hasNext() и next() имеют четкие аналоги в адаптируемом и целевом интерфейсе: их можно просто передавать напрямую. Но что делать с методом remove()? Задумайтесь на минуту (ответ приведен на следующей странице). А пока посмотрим диаграмму классов:



Проблема с методом remove()

Мы знаем, что Enumeration не поддерживает метод remove() — этот интерфейс обеспечивает доступ «только для чтения». В адаптере невозможно реализовать полнофункциональный метод remove(); фактически лучшее, что мы можем сделать, — это выдать исключение времени выполнения. К счастью, проектировщики интерфейса Iterator предвидели такую необходимость и определили метод remove() так, чтобы он поддерживал UnsupportedOperationException.

В данной ситуации адаптер не идеален; клиенты должны следить за потенциальными исключениями. Но если клиент будет достаточно осторожен, а адаптер — хорошо документирован, такое решение вполне приемлемо.

Программирование адаптера EnumerationIterator

Простой, но эффективный код адаптера для старых классов, которые продолжают работать с перечислениями:

```
public class EnumerationIterator implements Iterator<Object> {
    Enumeration<?> enumeration;

    public EnumerationIterator(Enumeration<?> enumeration) {
        this.enumeration = enumeration;
    }


    public boolean hasNext() {
        return enumeration.hasMoreElements();
    }

    public Object next() {
        return enumeration.nextElement();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Чтобы адаптер воспринимался клиентским кодом как итератор, он реализует интерфейс Iterator.

Адаптируемый объект Enumeration сохраняется в переменной (композиция).

Метод hasNext() интерфейса Iterator передает управление методу hasMoreElements() интерфейса Enumeration...

... а метод next() интерфейса Iterator передает управление методу nextElement().

К сожалению, поддержать метод remove() интерфейса Iterator не удастся, поэтому мы просто инициируем исключение.

Упражнение

Хотя язык Java развивается в направлении итераторов, осталось немало **старого кода**, который зависит от интерфейса Enumeration, поэтому адаптер, преобразующий Iterator в Enumeration, будет весьма полезным.

Напишите обратный адаптер, который преобразует Iterator в Enumeration. Для тестирования своего кода используйте класс ArrayList. Он поддерживает интерфейс Iterator, но не поддерживает Enumeration (по крайней мере *пока* не поддерживает).

МОЗГОВОЙ ШТУРМ

Некоторые адаптеры питания не ограничиваются простым изменением интерфейса — они добавляют такие функции, как защита от скачков напряжения, индикаторы и т. д.

Если вам потребуется реализовать такие функции, какой паттерн вы выберете?

Беседа у камина

Паттерн Декоратор и паттерн Адаптер обсуждают свои различия.

Декоратор

Я — важная персона. Если при проектировании используется паттерн Декоратор, значит, в архитектуру добавляются новые обязанности или аспекты поведения.

Тоже верно, но не думайте, что нам не приходится трудиться. Декорирование большого интерфейса требует весьма значительного объема кода.

Не стоит думать, что нам достается вся слава. Иногда я оказываюсь всего лишь очередным декоратором, который «упаковывается» в другие декораторы. Получая делегированный вызов метода, я не знаю, сколько декораторов уже приложило к нему руки и заметит ли кто-нибудь мою роль в обработке запроса.

Адаптер

Значит, вся слава достается вам, а мы, адаптеры, сидим в канаве и выполняем всю грязную работу по преобразованию интерфейсов? Возможно, наша работа не так эффектна, но клиенты ценят нас за то, что мы упрощаем их жизнь.

Побывали бы вы адаптером, когда он сводит воедино несколько классов для получения нужного интерфейса... Это я понимаю.

А если адаптер хорошо справляется со своей работой, то клиент вообще не замечает его. Совершенно неблагодарная работа.

Декоратор

Что ж, мы, декораторы, делаем примерно то же, но позволяем включать в классы *новое поведение* без изменения существующего кода. На мой взгляд, адаптер — всего лишь разновидность декоратора; в конце концов вы, как и мы, просто инкапсулируете объекты.

Нет, мы расширяем поведение или обязанности объектов. Неправильно считать нас *простой прокладкой*.

Что ж, надо признать, на бумаге мы действительно немного похожи, но в своем *предназначении* мы очень далеки друг от друга.

Адаптер

Но самое замечательное в нас, адаптерах, то, что мы позволяем клиентам использовать новые библиотеки *без изменения кода*; они просто полагаются на то, что мы выполним преобразования за них. Может, это и узкая специализация, но мы с ней хорошо справляемся.

Нет-нет, ничего подобного. Мы *всегда* преобразуем интерфейс внутреннего объекта, вы этого не делаете *никогда*. Я бы сказал, что декоратор является разновидностью адаптера, которая никогда не изменяет интерфейс!

Эй, ты кого назвал «простой прокладкой»? Долго ли ты протянешь, если тебе придется преобразовывать *несколько* интерфейсов?

Пожалуй, я соглашусь.

А теперь сменим тему...

В этой главе описан еще один паттерн.

Вы уже видели, как паттерн Адаптер преобразует интерфейс класса к другому интерфейсу, на который рассчитан ваш клиент. Также мы выяснили, что в Java эта задача решается «упаковкой» объекта, обладающего несовместимым интерфейсом, в объект, реализующий правильный интерфейс.

Наш следующий паттерн тоже изменяет интерфейс, но по другой причине: для его упрощения. Его название — Фасад — подобрано весьма удачно: паттерн скрывает все сложности внутреннего строения одного или нескольких классов за лаконичным, четко определенным фасадом.

Кто и что делает?

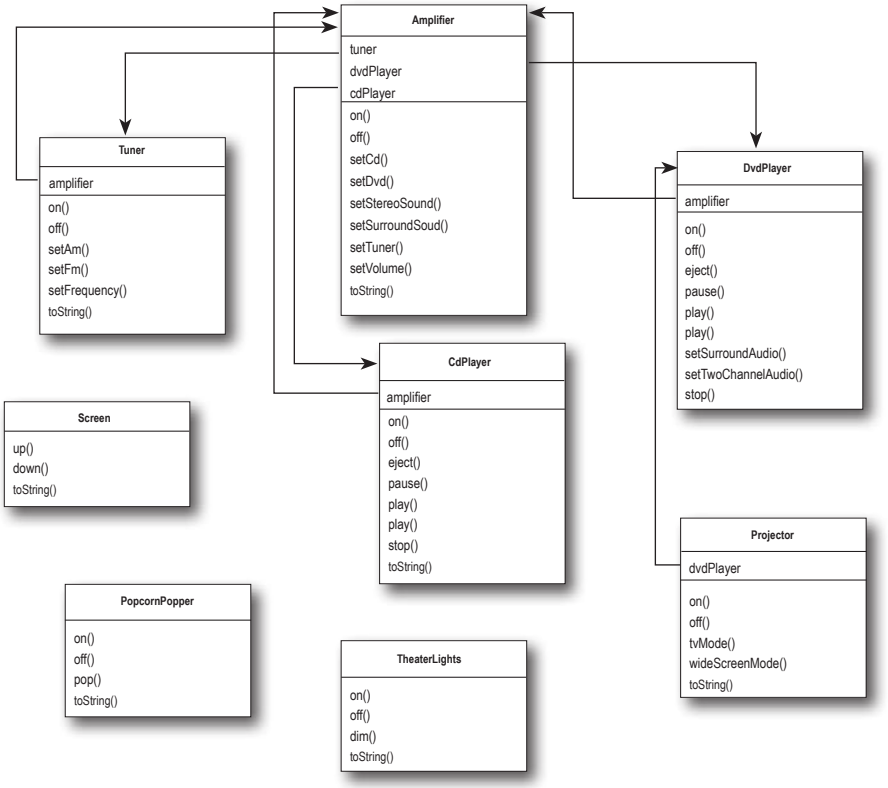
Проведите стрелку от каждого паттерна к его предназначению:

Паттерн	Предназначение
Декоратор	Преобразует один интерфейс к другому
Адаптер	Не изменяет интерфейс, но добавляет новые обязанности
Фасад	Упрощает интерфейс

Домашний кинотеатр

Прежде чем погружаться в подробности строения паттерна Фасад, давайте обратимся к модной нынче теме: построению домашнего кинотеатра.

Вы собрали данные и собрали систему своей мечты с DVD-проигрывателем, проектором, автоматизированным экраном, системой окружающего звука и даже аппаратом для приготовления попкорна.



Много классов, много взаимодействий, много интерфейсов, которые нужно изучать и использовать.

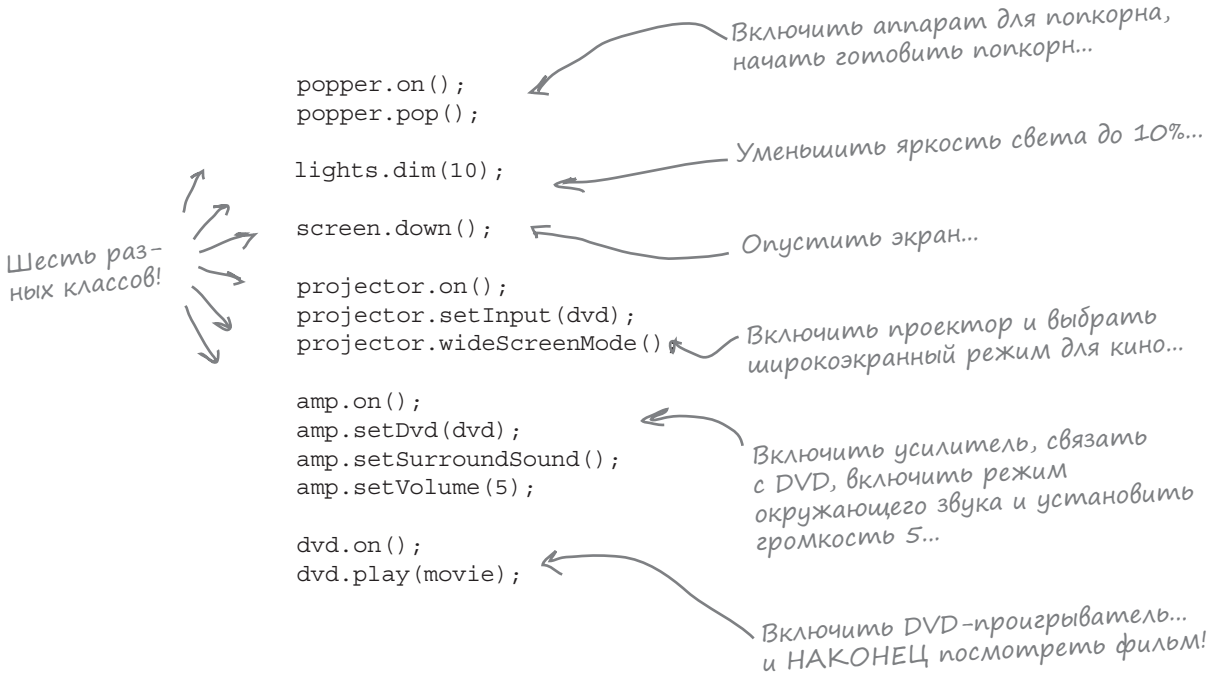
Вы провели целую неделю за прокладкой проводов, установкой проектора, подключением и настройкой аппаратуры. Теперь пора запустить систему и насладиться фильмом...

Просмотр фильма (сложный способ)

Выберите диск, расслабьтесь и приготовьтесь встретиться с киноволшебством. Да, и еще одно... Чтобы посмотреть кино, нужно выполнить несколько операций:

1. Включить аппарат для попкорна.
2. Запустить приготовление попкорна.
3. Выключить свет.
4. Опустить экран.
5. Включить проектор.
6. Связать вход проектора с выходом DVD.
7. Включить полноэкранный режим на проекторе.
8. Включить усилитель.
9. Связать вход усилителя с выходом DVD.
10. Включить на усилителе режим окружающего звука
11. Установить на усилителе среднюю громкость (5).
12. Включить DVD-проигрыватель.
13. Включить воспроизведение на DVD-проигрывателе.

Рассмотрим те же операции в контексте классов и вызовов методов, необходимых для их выполнения:



Но это еще не все...

- Когда фильм закончится, как отключить всю аппаратуру?

  Не придется ли вам проделывать все заново в обратном порядке?

- Не возникнут ли такие же сложности при прослушивании компакт-дисков или радио?

- Если вы решите обновить свою систему, вероятно, вам придется изучать новую последовательность действий.

Что же делать? Очевидно, работать с домашним кинотеатром слишком сложно!

К счастью, паттерн Фасад способен избавить от лишних хлопот, чтобы мы просто получили удовольствие от просмотра...

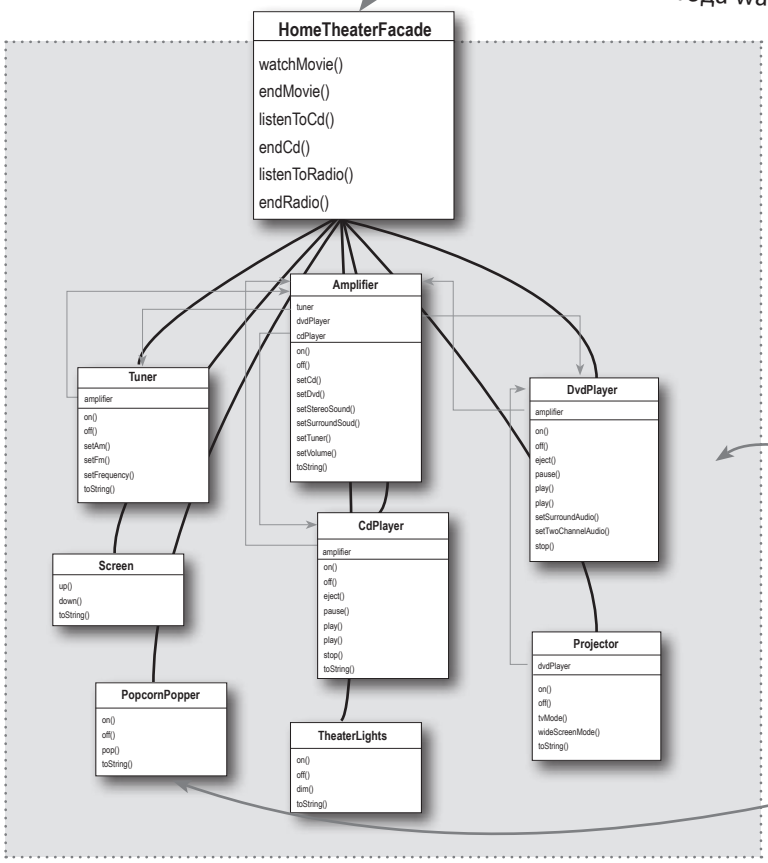
Свет, камера, фасад!

Фасад — именно то, что нам нужно: мы берем сложную подсистему и для упрощения работы с ней реализуем фасадный класс, который предоставляет общий, более удобный интерфейс. Не беспокойтесь; вся мощь сложной подсистемы остается в вашем распоряжении, но если вам нужен только упрощенный интерфейс — пользуйтесь Фасадом.

Давайте посмотрим, как работает паттерн Фасад:

1 Мы создадим фасадный интерфейс для домашнего кинотеатра. Класс HomeTheaterFacade предоставляет простые методы — такие, как watchMovie().

2 Класс фасада рассматривает компоненты домашнего кинотеатра как подсистему и обращается с вызовами к этой подсистеме для реализации своего метода watchMovie().



3 Клиентский код теперь обращается с вызовами к фасадному классу домашнего кинотеатра, а не к подсистеме. Таким образом, чтобы просмотреть фильм, мы вызываем всего один метод — watchMovie(), — а тот берет на себя все взаимодействие с освещением, DVD-проигрывателем, проектором, усилителем, экраном и т. д.

4 В паттерне Фасад подсистема остается открытой для непосредственного использования. Если вам понадобится расширенная функциональность классов подсистемы — обращайтесь к ним напрямую.

Часто Задаваемые Вопросы

В: Если Фасад инкапсулирует классы подсистемы, как клиент сможет получить доступ к ним?

О: Фасады не «инкапсулируют» классы подсистемы, они всего лишь предоставляют упрощенный интерфейс к их функциям. Классы подсистемы остаются доступными для прямого использования со стороны клиентов, нуждающихся в более конкретных интерфейсах. Это одно из преимуществ паттерна Фасад: он предоставляет упрощенный интерфейс, оставляя доступ к полной функциональности подсистемы.

В: Добавляет ли фасад какую-либо функциональность или просто передает запросы подсистеме?

О: Фасад может «проявить сообразительность» в работе с подсистемой. Например, в случае с домашним кинотеатром он знает, что аппарат для попкорна нужно включить заранее.

В: Подсистема может иметь только один фасад?

О: Не обязательно. Паттерн позволяет создать для подсистемы любое количество фасадов.

В: Какими преимуществами обладает Фасад, кроме упрощения интерфейса?

О: Паттерн Фасад также позволяет отделить клиентскую реализацию от конкретной подсистемы. Допустим, вы решили обновить свой домашний кинотеатр новыми компонентами с другим интерфейсом. Если клиентский код написан для работы с фасадом, а не с подсистемой, изменять его не придется — достаточно изменить фасад.

В: Выходит, различие между паттернами Адаптер и Фасад заключается в том, что Адаптер инкапсулирует один класс, а Фасад может представлять много классов?

О: Нет! Хотя в большинстве примеров адаптер адаптирует один класс, иногда приходится адаптировать несколько классов для формирования интерфейса, на который запрограммирован клиент. Различие между ними определяется не количеством «инкапсулируемых» классов, а целью. Целью паттерна Адаптер является изменение интерфейса и приведение его к тому виду, на который рассчитан клиент. Целью паттерна Фасад является упрощение интерфейса подсистемы.

Фасад не только упрощает интерфейс, но и обеспечивает логическую изоляцию клиента от подсистемы, состоящей из многих компонентов.

Фасад применяется для упрощения, а адаптер — для преобразования интерфейса к другой форме.

Построение фасада для домашнего кинотеатра

Давайте по шагам разберем процесс построения фасада для домашнего кинотеатра. Прежде всего мы воспользуемся композицией, чтобы фасад имел доступ ко всем компонентам подсистемы:

```
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
                Tuner tuner,
                DvdPlayer dvd,
                CdPlayer cd,
                Projector projector,
                Screen screen,
                TheaterLights lights,
                PopcornPopper popper) {

        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

        // Другие методы

}
```

Композиция: компоненты подсистемы, которые мы собираемся использовать.

В конструкторе фасада передаются ссылки на все компоненты. Фасад присваивает их соответствующим переменным экземпляра.

Сейчас мы займемся ими...

Реализация упрощенного интерфейса

Настало время свести компоненты подсистемы в единый интерфейс. Начнем с методов watchMovie() и endMovie():

```
public void watchMovie(String movie) {
    System.out.println("Get ready to watch a movie...");
    popper.on();
    popper.pop();
    lights.dim(10);
    screen.down();
    projector.on();
    projector.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
}

public void endMovie() {
    System.out.println("Shutting movie theater down...");
    popper.off();
    lights.on();
    screen.up();
    projector.off();
    amp.off();
    dvd.stop();
    dvd.eject();
    dvd.off();
}
```

Метод watchMovie() выполняет те же операции, которые ранее выполнялись нами вручную. Обратите внимание: выполнение каждой операции делегируется соответствующему компоненту подсистемы.

Метод endMovie() выключает всю аппаратуру за нас. И снова каждая операция делегируется соответствующему компоненту подсистемы.

Вспомните фасады, которые встречались вам в Java API. В каких областях вы предложили бы создать новые фасады?

Просмотр фильма (простой способ)

Сеанс начинается!

```
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        // Создание экземпляров компонентов

        HomeTheaterFacade homeTheater =
                new HomeTheaterFacade(amp, tuner, dvd, cd,
                        projector, screen, lights, popper);

        homeTheater.watchMovie("Raiders of the Lost Ark");
        homeTheater.endMovie();
    }
}
```

Компоненты создаются прямо в ходе тестирования. Обычно клиент получает фасад, а не создает его.

Сначала мы создаем фасад со всеми компонентами подсистемы.

Упрощенный интерфейс используется для запуска и для прекращения просмотра.

Результат.

Вызов метода watchMovie() фасада выполняет всю работу за нас...

...просмотр закончен, вызов endMovie() выключает аппаратуру.

```
File  Edit  Window  Help  SnakesWhy'dItHaveToBeSnakes?
%java HomeTheaterTestDrive

Get ready to watch a movie...
Popcorn Popper on
Popcorn Popper popping popcorn!
Theater Ceiling Lights dimming to 10%
Theater Screen going down
Top-O-Line Projector on
Top-O-Line Projector in widescreen mode (16x9 aspect ratio)
Top-O-Line Amplifier on
Top-O-Line Amplifier setting DVD player to Top-O-Line DVD Player
Top-O-Line Amplifier surround sound on (5 speakers, 1 subwoofer)
Top-O-Line Amplifier setting volume to 5
Top-O-Line DVD Player on
Top-O-Line DVD Player playing "Raiders of the Lost Ark"
Shutting movie theater down...
Popcorn Popper off
Theater Ceiling Lights on
Theater Screen going up
Top-O-Line Projector off
Top-O-Line Amplifier off
Top-O-Line DVD Player stopped "Raiders of the Lost Ark"
Top-O-Line DVD Player eject
Top-O-Line DVD Player off
%
```

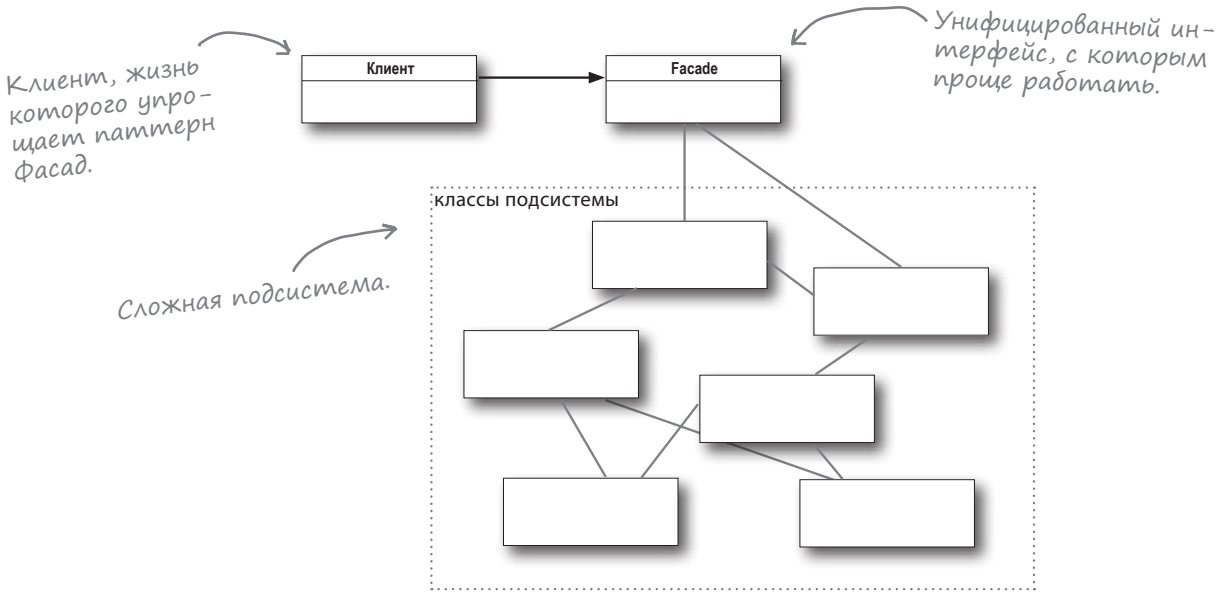
Определение паттерна Фасад

При использовании паттерна Фасад мы создаем класс, который упрощает и унифицирует набор более сложных классов, образующих некую подсистему. В отличие от многих других паттернов, Фасад относительно прост; в нем нет умопомрачительных абстракций, в которых приходится подолгу разбираться. Но от этого он не становится менее полезным; паттерн Фасад предотвращает появление сильных связей между клиентом и подсистемой и, как вы вскоре увидите, способствует соблюдению нового принципа объектно-ориентированного проектирования.

Но прежде чем переходить к новому принципу, мы рассмотрим официальное определение паттерна:

> **Паттерн Фасад** предоставляет унифицированный интерфейс к группе интерфейсов подсистемы. Фасад определяет высокоуровневый интерфейс, упрощающий работу с подсистемой.

В этом определении нет ничего такого, чего бы вы не знали; самое важное, что нужно запомнить об этом паттерне, — это его предназначение. Определение четко и недвусмысленно говорит, что целью фасада является упрощение работы с подсистемой за счет использования упрощенного интерфейса. Это хорошо видно из диаграммы классов паттерна:



Собственно, это всё; в вашем арсенале появился новый паттерн! А теперь можно и перейти к новому принципу ОО-проектирования. Будьте внимательны — он противоречит некоторым распространенным представлениям!

Принцип минимальной информированности

Принцип минимальной информированности требует сократить взаимодействия между объектами до нескольких близких «друзей». Обычно он формулируется в следующем виде:

> **Принцип проектирования**
>
> 
>
> *Принцип минимальной информированности: общайтесь только с близкими друзьями.*

Что это означает на практике? Что при проектировании системы для любого объекта следует обратить особое внимание на количество классов, с которыми он взаимодействует, и каким образом организовано это взаимодействие.

Этот принцип препятствует созданию архитектур с большим количеством тесно связанных классов, в которых изменение в одной части системы каскадно распространяется в другие части. При формировании многочисленных зависимостей между классами система теряет гибкость и становится сложной для понимания, а затраты на ее сопровождение возрастают.

МОЗГОВОЙ ШТУРМ

Со сколькими классами связан этот фрагмент кода?

```
public float getTemp() {
     return station.getThermometer().getTemperature();
}
```

Как НЕ приобретать друзей и оказывать влияние на объекты

Но как добиться этой цели? Принцип дает некоторые рекомендации. Возьмем произвольный объект; согласно принципу, из любого метода этого объекта должны вызываться только методы, принадлежащие:

- самому объекту;
- объектам, переданным в параметрах метода;
- любому объекту, созданному внутри метода;
- любым компонентам объекта.

Эти правила запрещают вызывать методы для объектов, полученных в результате вызова других методов!

Под «компонентом» следует понимать любой объект, на который ссылается переменная экземпляра. Иначе говоря, речь идет об объекте, связанном отношением типа СОДЕРЖИТ.

Ограничения кажутся вам слишком жесткими? Какой вред принесет вызов метода объекта, полученного в результате другого вызова? Дело в том, что он обращен к одной из подчастей другого объекта, а следовательно, увеличивает число объектов, о которых непосредственно «знает» текущий объект. В таких случаях принцип требует, чтобы объект выдавал этот запрос за нас (чтобы круг «друзей» оставался по возможности узким). Пример:

Без принципа

```
public float getTemp() {
    Thermometer thermometer = station.getThermometer();
    return thermometer.getTemperature();
}
```

Здесь мы получаем объект Thermometer от station, а затем вызываем метод getTemperature() самостоятельно.

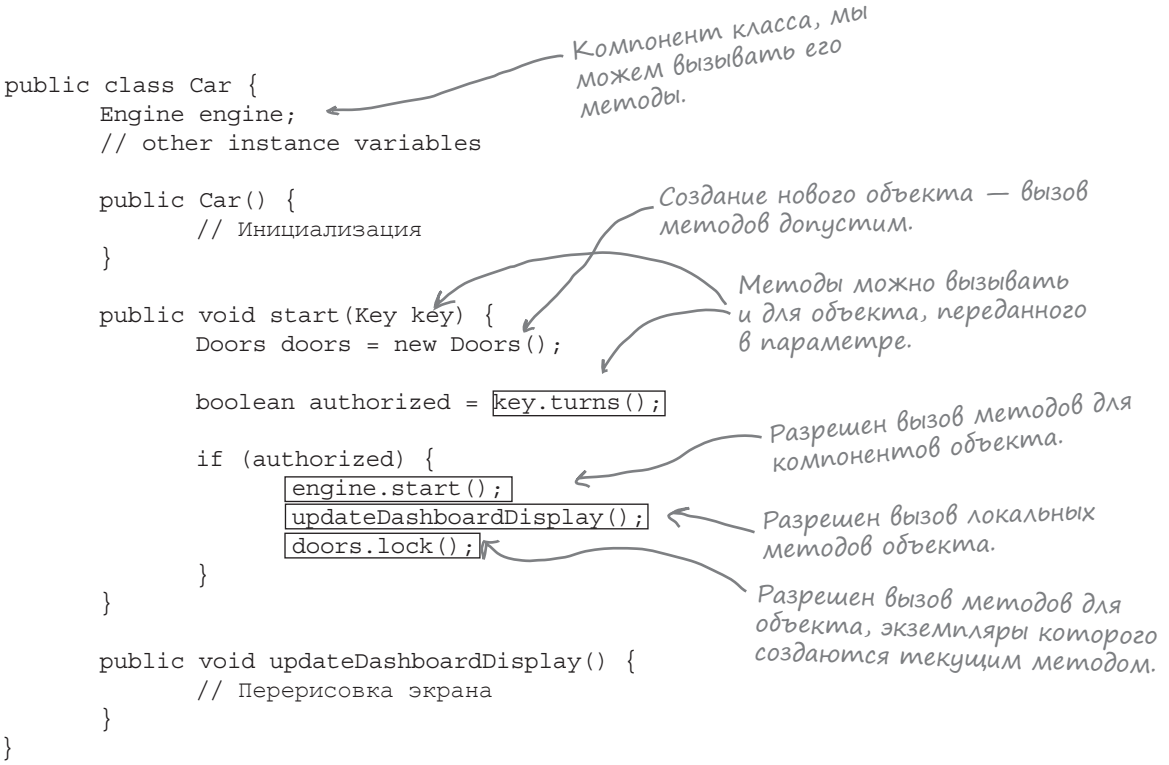
С принципом

```
public float getTemp() {
    return station.getTemperature();
}
```

В класс Station включается метод, который обращается с запросом к Thermometer за нас. Тем самым сокращается количество классов, от которых зависит наш код.

Ограничение вызовов методов

Приведенный ниже класс Car демонстрирует все возможности вызова методов, соответствующие принципу минимальной информированности:




```
public class Car {
        Engine engine;
        // other instance variables

        public Car() {
                // Инициализация
        }

        public void start(Key key) {
                Doors doors = new Doors();

                boolean authorized = key.turns();

                if (authorized) {
                        engine.start();
                        updateDashboardDisplay();
                        doors.lock();
                }
        }

        public void updateDashboardDisplay() {
                // Перерисовка экрана
        }
}
```

Часто Задаваемые Вопросы

В: Существует похожий принцип, называемый «законом Деметры»; как они связаны?

О: Это два разных названия одного принципа. Мы предпочитаем термин «принцип минимальной информированности» по двум причинам: (1) он более интуитивен, и (2) слово «закон» наводит на мысль о том, что этот принцип должен применяться всегда. Однако принципы следует применять только тогда, когда они приносят практическую пользу, а перед их применением следует учесть все факторы (абстракции/скорость, затраты памяти/времени и т. д.)

В: Есть ли недостатки у принципа минимальной информированности?

О: Да; хотя принцип сокращает зависимости между объектами, а анализ показал, что это снижает затраты на сопровождение, применение принципа приводит «оберток» для вызова методов других компонентов. Это усложняет код и увеличивает время разработки, а также к снижает быстродействие на стадии выполнения.

Возьми в руку карандаш Нарушают ли какие-либо из этих классов принцип минимальной информированности? Почему?

```
public House {
    WeatherStation station;

    // Другие методы и конструктор

    public float getTemp() {
         return station.getThermometer().getTemperature();
    }
}
```

```
public House {
     WeatherStation station;

    // Другие методы и конструктор

    public float getTemp() {
         Thermometer thermometer = station.getThermometer();
         return getTempHelper(thermometer);
    }

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

ОСТОРОЖНО!
ОПАСНАЯ ЗОНА!

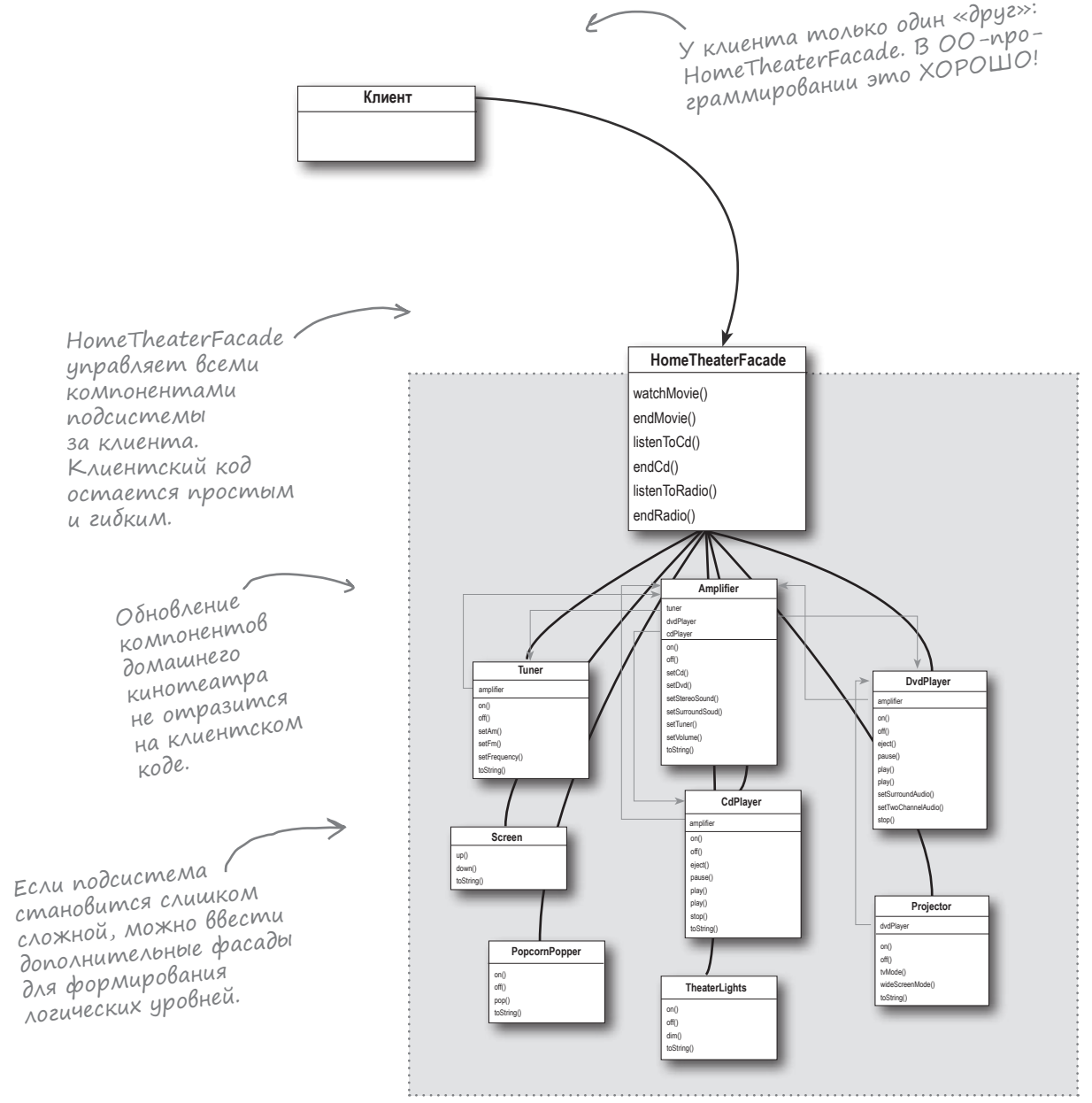
МОЗГОВОЙ ШТУРМ

Приведите пример стандартной конструкции Java, нарушающей принцип минимальной информированности.

Нужно ли переживать по этому поводу?

Ответ: Как насчет System.out.println()?

Фасад и принцип минимальной информированности



Новые инструменты

Наш инструментарий основательно разросся! В этой главе были описаны два паттерна, изменяющих интерфейсы и сокращающих привязку клиента к используемой системе.

Концепции

- ...страция
- ...капсуляция
- ...лиморфизм
- ...следование

Принципы

- Инкапсулируйте то, что изменяется.
- Предпочитайте композицию наследованию.
- Программируйте на уровне интерфейсов.
- Стремитесь к слабой связанности взаимодействующих объектов.
- Классы должны быть открыты для расширения, но закрыты для изменения.
- Код должен зависеть от абстракций, а не от конкретных классов.
- Взаимодействуйте только с «друзьями».

У нас появился новый принцип проектирования...

...и ДВА новых паттерна. Оба изменяют интерфейсы: Адаптер — для преобразования, Фасад — для унификации и упрощения.

Паттерны

Адаптер преобразует интерфейс класса к другому интерфейсу, на который рассчитан клиент. Адаптер обеспечивает совместную работу классов, невозможную в обычных условиях из-за несовместимости интерфейсов.

...нсулиру-
...объекта,
...пара-
...ентских
...ми запро-

Фасад предоставляет унифицированный интерфейс к группе интерфейсов подсистемы. Фасад определяет высокоуровневый интерфейс, упрощающий работу с подсистемой.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Если вам понадобится использовать существующий класс с неподходящим интерфейсом — используйте адаптер.
- Если вам понадобится упростить большой интерфейс или семейство сложных интерфейсов — используйте фасад.
- Адаптер приводит интерфейс к тому виду, на который рассчитан клиент.
- Фасад изолирует клиента от сложной подсистемы.
- Объем работы по реализации адаптера зависит от размера и сложности целевого интерфейса.
- Реализация фасада основана на композиции и делегировании.
- Существуют две разновидности адаптеров: адаптеры объектов и адаптеры классов. Для адаптеров классов необходимо множественное наследование.
- Для подсистемы можно реализовать несколько фасадов.

Возьми в руку карандаш

Решение

Предположим, вам также понадобился адаптер для преобразования Duck в Turkey — назовем его DuckAdapter. Напишите код этого класса:

Теперь Duck адаптируется в Turkey, поэтому реализуем интерфейс Turkey.

```
public class DuckAdapter implements Turkey {
    Duck duck;
    Random rand;

    public DuckAdapter(Duck duck) {
        this.duck = duck;
        rand = new Random();
    }

    public void gobble() {
        duck.quack();
    }

    public void fly() {
        if (rand.nextInt(5)  == 0) {
            duck.fly();
        }
    }
}
```

Сохраняем ссылку на адаптируемый объект Duck.

Случайный объект используется в методе fly().

Вызов gobble() превращается в quack().

Так как утки летают намного дальше индюшек, мы решили, что утка будет летать в среднем один раз из пяти.

Возьми в руку карандаш

Решение

Нарушают ли какие-либо из этих классов принцип минимальной информированности? Почему?

```
public House {
    WeatherStation station;

    // Другие методы и конструктор

    public float getTemp() {
        return station.getThermometer().getTemperature();
    }
}
```

Принцип нарушается! Мы вызываем метод объекта, полученного в результате вызова другого метода.

```
public House {
     WeatherStation station;

    // Другие методы и конструктор

    public float getTemp() {
        Thermometer thermometer = station.getThermometer();
        return getTempHelper(thermometer);
    }

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

Принцип не нарушается! Но что реально изменилось с перемещением вызова в другой метод?

Возьми в руку карандаш
Решение

Вы уже видели, как реализовать адаптер, преобразующий Enumeration в Iterator; напишите обратный адаптер, преобразующий Iterator в Enumeration.

Обратите внимание: тип сделан обобщенным параметром, чтобы решение работало для объектов любого типа.

```
public class IteratorEnumeration implements Enumeration<Object> {
    Iterator<?> iterator;

    public IteratorEnumeration(Iterator<?> iterator) {
        this.iterator = iterator;
    }

    public boolean hasMoreElements() {
        return iterator.hasNext();
    }

    public Object nextElement() {
        return iterator.next();
    }
}
```

Кто и что делает?
Решение

Проведите стрелку от каждого паттерна к его предназначению:

Паттерн	Предназначение
Декоратор	Преобразует один интерфейс к другому
Адаптер	Не изменяет интерфейс, но добавляет новые обязанности
Фасад	Упрощает интерфейс

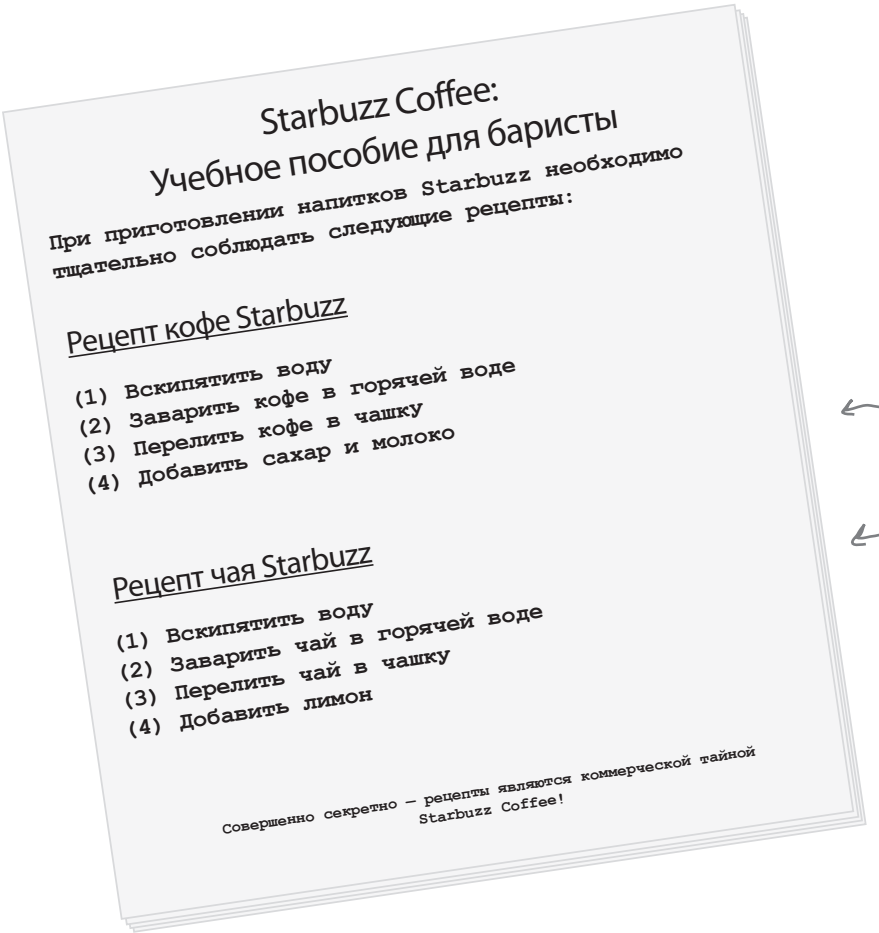
Инкапсуляция алгоритмов

Мы уже «набили руку» на инкапсуляции; мы инкапсулировали создание объектов, вызовы методов, сложные интерфейсы, уток, пиццу... Что дальше? Следующим шагом будет инкапсуляция алгоритмических блоков, чтобы субклассы могли в любой момент связаться с нужным алгоритмом обработки. В этой главе даже будет описан принцип проектирования, вдохновленный Голливудской практикой.

Пора принимать кофеин

Одни люди не могут жить без кофе; другие не могут жить без чая. Общий ингредиент? Кофеин, конечно!

Но это еще не все; способы приготовления чая и кофе имеют много общего. Посмотрите сами:



Starbuzz Coffee:
Учебное пособие для баристы

При приготовлении напитков Starbuzz необходимо тщательно соблюдать следующие рецепты:

Рецепт кофе Starbuzz

(1) Вскипятить воду
(2) Заварить кофе в горячей воде
(3) Перелить кофе в чашку
(4) Добавить сахар и молоко

Рецепт чая Starbuzz

(1) Вскипятить воду
(2) Заварить чай в горячей воде
(3) Перелить чай в чашку
(4) Добавить лимон

Совершенно секретно — рецепты являются коммерческой тайной Starbuzz Coffee!

Рецепт приготовления кофе очень похож на рецепт приготовления чая, не правда ли?

Кофе и чай (на языке Java)

Давайте поиграем в «баристу от программирования» и напишем классы для кофе и чая.

Сначала кофе:

Класс Coffee для приготовления кофе:

Рецепт кофе взят прямо из учебного пособия.

Каждый шаг реализован в виде отдельного метода.

```
public class Coffee {

    void prepareRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void brewCoffeeGrinds() {
        System.out.println("Dripping Coffee through
filter");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }

    public void addSugarAndMilk() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Каждый из этих методов реализует один шаг алгоритма: кипячение воды, настаивание кофе, разливание по чашкам, добавление сахара и молока.

А теперь чай...

```
public class Tea {

    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void steepTeaBag() {
        System.out.println("Steeping the tea");
    }

    public void addLemon() {
        System.out.println("Adding Lemon");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

Реализация очень похожа на реализацию Coffee; шаги 2 и 4 различаются, но рецепт почти не изменился.

Эти два метода специфичны для класса Tea.

А эти два метода в точности совпадают с методами Coffee! Имеет место явное дублирование кода.

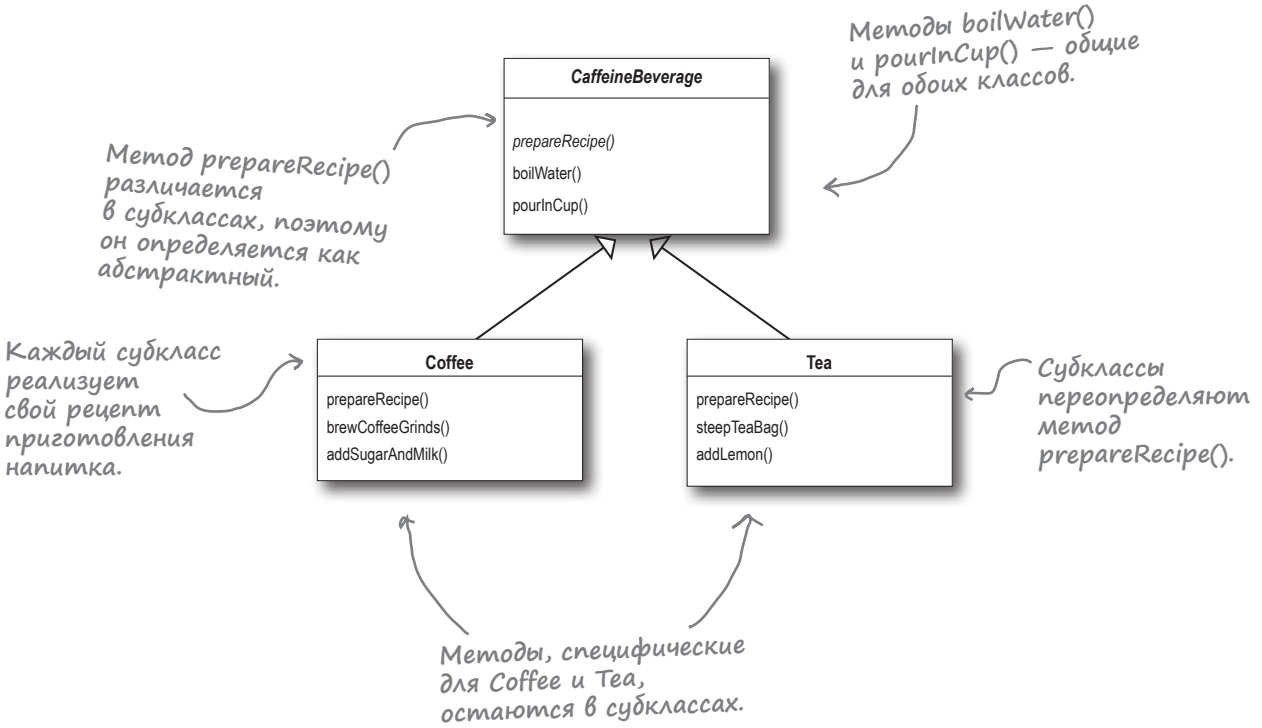
Дублирование кода — верный признак того, что в архитектуру необходимо вносить изменения. Раз чай и кофе так похожи, может, стоит выделить их сходные аспекты в базовый класс?

Головоломка

Классы Coffee и Tea содержат дублирующийся код. Взгляните еще раз на эти классы и нарисуйте обновленную диаграмму классов, в которой из этих классов устраняется избыточность:

Сэр, вам налить абстрактного кофе?

Упражнение с классами Coffee и Tea на первый взгляд кажется весьма заурядным. Первая версия может выглядеть примерно так:

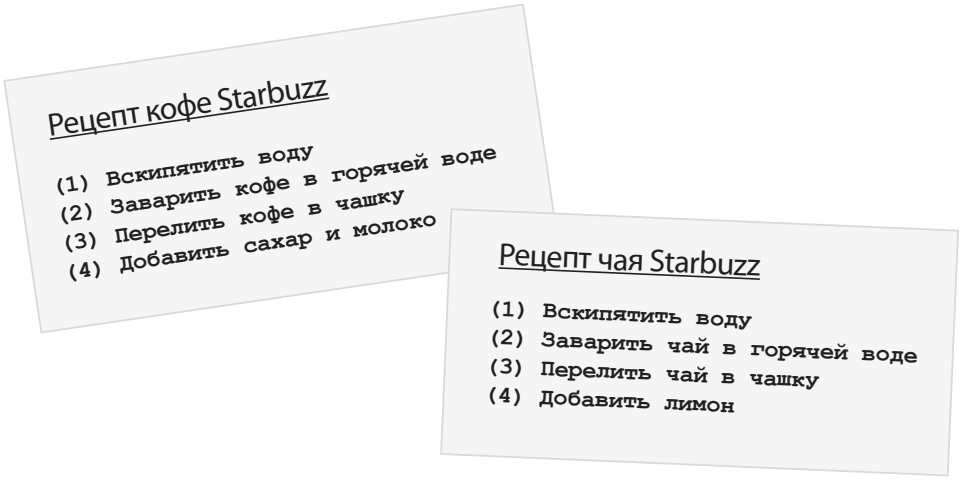


МОЗГОВОЙ ШТУРМ

Как вам результаты переработки архитектуры? Хорошо? Хмм, взгляните еще разок. Не упустили ли мы некоторые общие аспекты? В чем еще проявляется сходство классов Coffee и Tea?

Продолжаем переработку...

Что еще общего у классов Coffee и Tea? Начнем с рецептов.



Обратите внимание: оба рецепта следуют одному алгоритму:

1. **Вскипятить воду.**
2. **Использовать горячую воду для настаивания кофе или чая.**
3. **Перелить напиток в чашку.**
4. **Добавить соответствующие дополнения в напиток.**

Эти операции не абстрагированы, но по сути одинаковы — просто они выполняются с разными напитками.

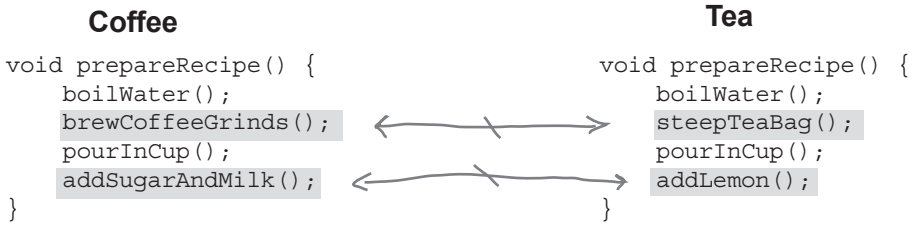
Эти две операции уже выведены в базовый класс.

Итак, нельзя ли абстрагировать и метод prepareRecipe()? Давайте посмотрим...

Абстрагирование prepareRecipe()

Давайте шаг за шагом рассмотрим процесс абстрагирования prepareRecipe() в субклассах (то есть классах Coffee и Tea)...

❶ Первая проблема: класс Coffee использует методы brewCoffeeGrinds() и addSugarAndMilk(), тогда как класс Tea использует методы steepTeaBag() и addLemon().



Однако процессы заваривания кофе и чая мало чем различаются. Давайте создадим новый метод (скажем, с именем brew()) и будем использовать его для заварки как кофе, так и чая.

Аналогичным образом добавление сахара и молока имеет много общего с добавлением лимона. Для выполнения этой операции мы создадим новый метод addCondiments(). Обновленная версия метода prepareRecipe() выглядит так:

```
void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
}
```

❷ Новый метод prepareRecipe() необходимо связать с кодом. Для этого мы начнем с суперкласса CaffeineBeverage:

CaffeineBeverage — абстрактный класс, как и в исходной архитектуре.

```
public abstract class CaffeineBeverage {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

Теперь для приготовления чая и кофе будет использоваться один метод — prepareRecipe(). Этот метод объявлен с ключевым словом final, потому что суперклассы не должны переопределять этот метод! Шаги 2 и 4 заменены обобщенными вызовами brew() и addCondiments().

Так как классы Coffee и Tea реализуют эти методы по-разному, мы объявляем их абстрактными. Субклассы должны предоставить их реализацию!

Не забывайте: эти методы перемещены в класс CaffeineBeverage (см. диаграмму классов).

❸ Теперь приготовление напитков определяется суперклассом CaffeineBeverage, так что классам Coffee и Tea остается лишь предоставить реализации нужных методов:

Как и в исходной архитектуре, Tea и Coffee расширяют класс CaffeineBeverage.

```
public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }
    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}
```

Класс Tea должен определить brew() и addCondiments() — два абстрактных метода из Beverage.

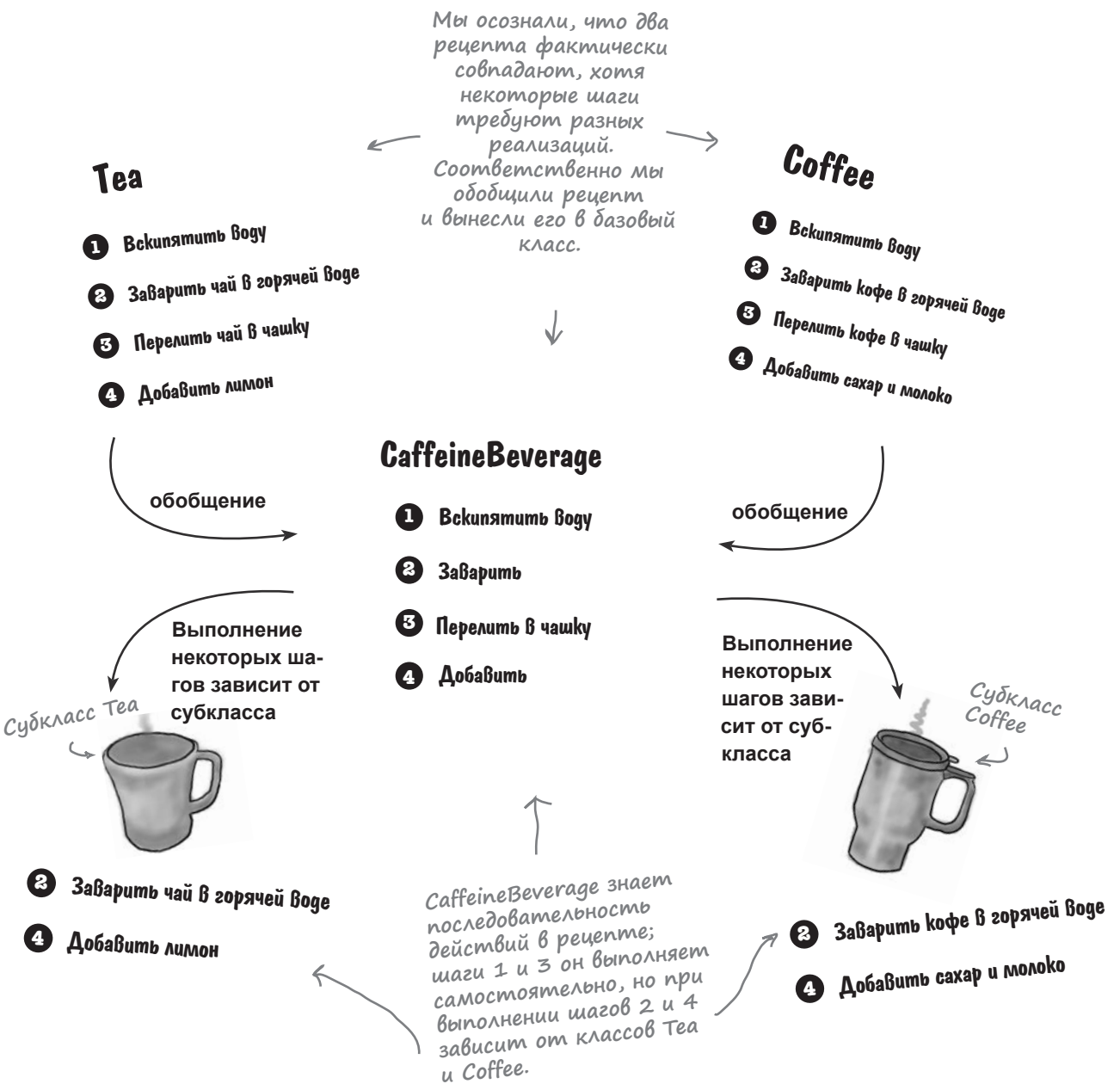
То же самое должен сделать и класс Coffee, только вместо пакетика чая и лимона он добавляет в напиток сахар и молоко.

```
public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }
    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Возьми в руку карандаш

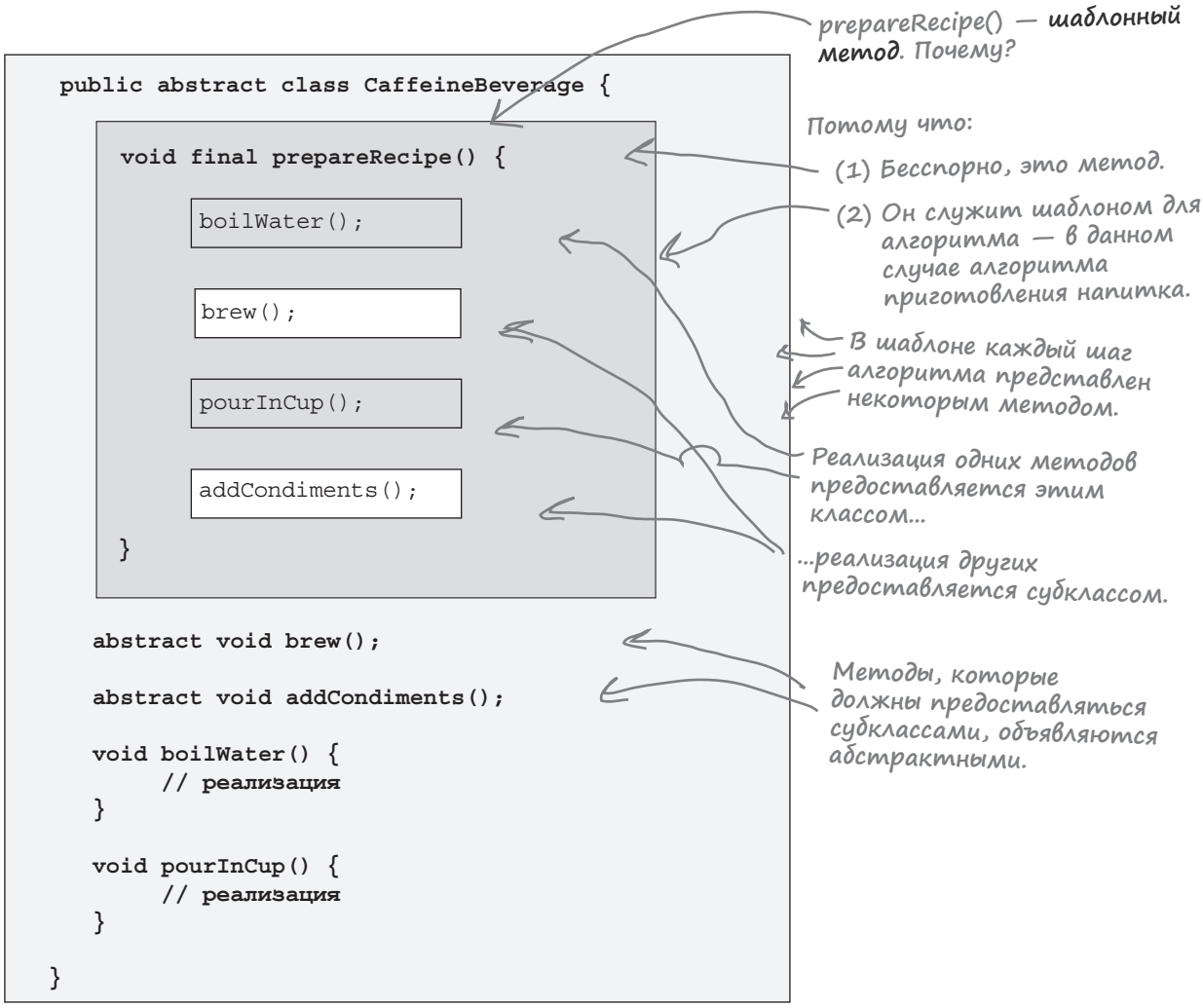
Нарисуйте новую диаграмму классов после перемещения реализации prepareRecipe() в класс CaffeineBeverage:

Что мы сделали?



Паттерн Шаблонный Метод

В сущности, мы только что реализовали паттерн Шаблонный Метод. Что это такое? Рассмотрим структуру класса CaffeineBeverage:



Шаблонный Метод определяет основные шаги алгоритма и позволяет субклассам предоставить реализацию одного или нескольких шагов.

Готовим чай...

Давайте последовательно разберем процедуру приготовления чая, обращая особое внимание на то, как работает шаблонный метод. Вы увидите, что шаблонный метод управляет алгоритмом; в некоторых точках алгоритма он дает возможность субклассу предоставить свою реализацию...

За сценой 

```
boilWater();
brew();
pourInCup();
addCondiments();
```

Метод prepareRecipe() определяет алгоритм, а реализации (полностью или частично) предоставляются субклассами.

1. Для начала нам понадобится объект Tea...

```
Tea myTea = new Tea();
```

2. Затем вызываем шаблонный метод:

```
myTea.prepareRecipe();
```

который следует алгоритму приготовления напитков...

3. Сначала кипятим воду:

```
boilWater();
```

Эта операция выполняется в CaffeineBeverage.

4. Затем необходимо заварить чай — только субкласс знает, как это правильно делается:

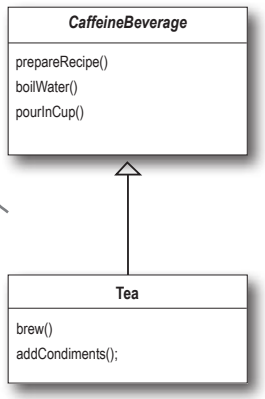
```
brew();
```

5. Чай переливается в чашку; эта операция выполняется одинаково для всех напитков, поэтому она тоже выполняется в CaffeineBeverage:

```
pourInCup();
```

6. В чай кладутся добавки, специфические для конкретного напитка, поэтому операция реализуется в субклассе:

```
addCondiments();
```



Что дает Шаблонный Метод?

Тривиальная реализация Tea и Coffee

Новый класс CaffeineBeverage на базе Шаблонного Метода

Алгоритм определяется классами Coffee и Tea.	Алгоритм определяется классом CaffeineBeverage.
Частичное дублирование кода в классах Coffee и Tea.	Класс CaffeineBeverage обеспечивает повторное использование кода между субклассами.
Модификация алгоритма требует открытия субклассов и внесения множественных изменений.	Алгоритм находится в одном месте, в котором вносятся изменения в коде алгоритма.
Добавление новых классов в такой структуре требует значительной работы.	Структура классов на базе Шаблонного Метода обеспечивает простое добавление новых классов — они лишь должны реализовать пару методов.
Знание алгоритма и его реализации распределено по многим классам.	Вся информация об алгоритме сосредоточена в классе CaffeineBeverage, а субклассы предоставляют полную реализацию.

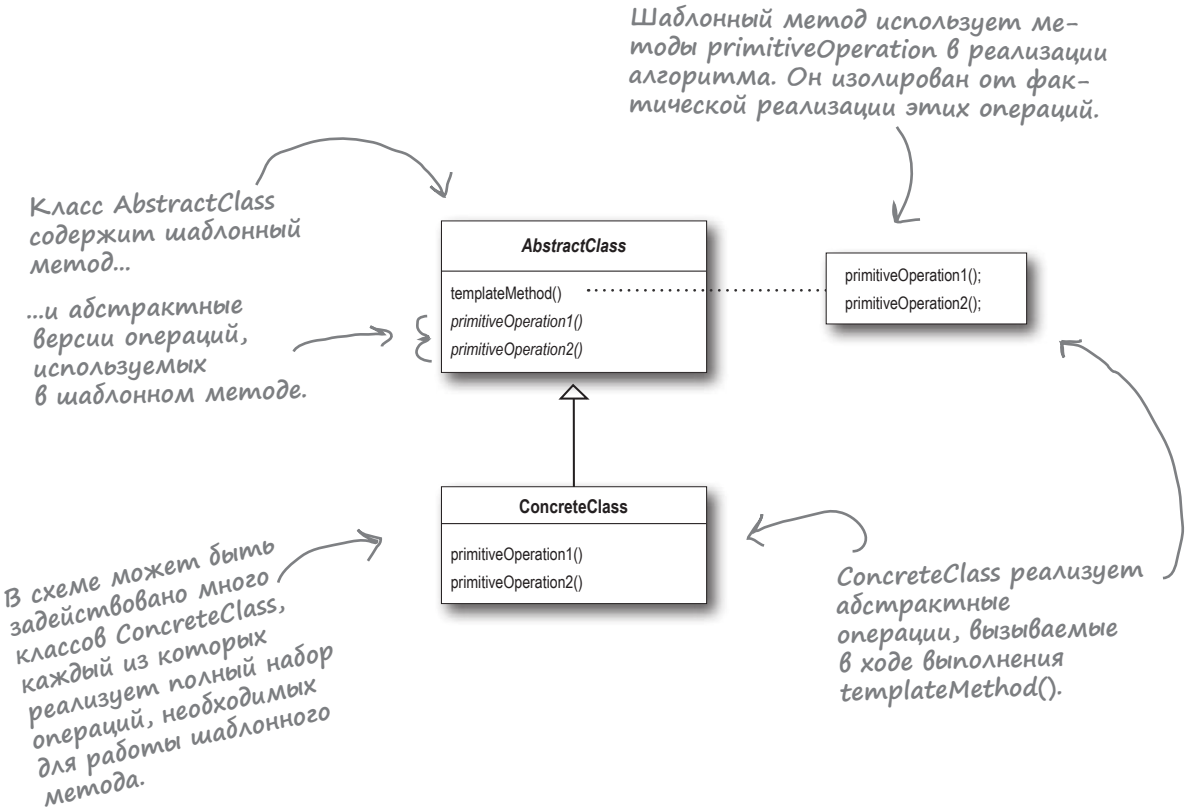
Определение паттерна Шаблонный Метод

Вы уже видели, как паттерн Шаблонный Метод применяется на практике. Теперь обратимся к официальному определению и уточним некоторые подробности:

> **Паттерн Шаблонный Метод** задает «скелет» алгоритма в методе, оставляя определение реализации некоторых шагов субклассам. Субклассы могут переопределять некоторые части алгоритма без изменения его структуры.

Основной задачей паттерна является создание шаблона алгоритма. Что такое «шаблон алгоритма»? Как было показано ранее, это метод; а конкретнее — метод, определяющий алгоритм в виде последовательности шагов. Один или несколько шагов определяются в виде абстрактных методов, реализуемых субклассами. Таким образом гарантируется неизменность структуры алгоритма при том, что часть реализации предоставляется субклассами.

Рассмотрим следующую диаграмму классов:



Код под увеличительным стеклом

Давайте повнимательнее присмотримся к определению AbstractClass, включая шаблонный метод и примитивные операции.

Абстрактный класс; он должен субклассироваться классами, предоставляющими реализацию операций.

Шаблонный метод; объявляется с ключевым словом final, чтобы субклассы не могли изменить последовательность шагов в алгоритме.

```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    void concreteOperation() {
        // реализации
    }
}
```

Шаблонный метод определяет последовательность шагов, каждый из которых представлен методом.

В данном примере две примитивные операции должны реализоваться конкретными субклассами.

Конкретная операция, определенная в абстрактном классе. Вскоре такие методы будут описаны более подробно...

Код под микроскопом

А сейчас мы еще подробнее рассмотрим методы, которые могут определяться в абстрактном классе:

В templateMethod() включен вызов нового метода.

```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    final void concreteOperation() {
        // реализация
    }

    void hook() {}

}
```

Примитивы-методы никуда не делись; они объявлены абстрактными и реализуются конкретными субклассами.

Конкретная операция определяется в абстрактном классе. Эта объявлена с ключевым словом final, чтобы субклассы не могли переопределить ее. Она может использоваться как напрямую шаблонным методом, так и субклассами.

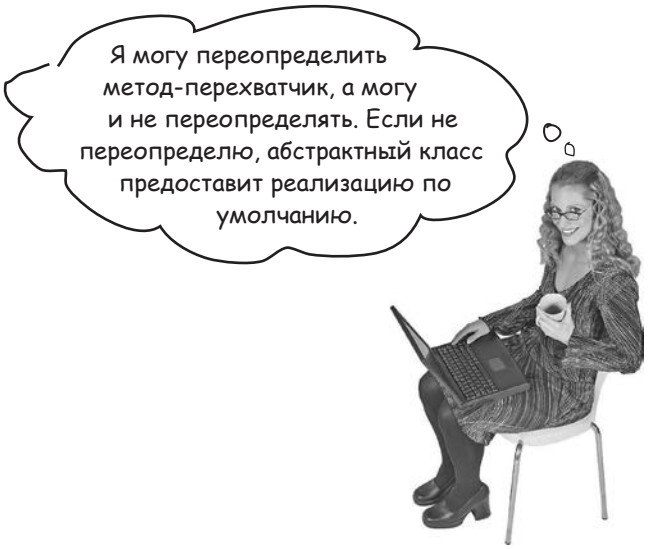
Конкретный метод, который не делает ничего!

Субклассы могут переопределять такие методы (называемые «перехватчиками»), но не обязаны это делать. Пример использования методов-перехватчиков представлен на следующей странице.

Перехватчики в паттерне Шаблонный Метод

«Перехватчиком» называется метод, объявленный абстрактным классом, но имеющий пустую реализацию (или реализацию по умолчанию). Он дает возможность субклассу «подключаться» к алгоритму в разных точках. Впрочем, субкласс также может проигнорировать имеющийся перехватчик.

Рассмотрим пример возможного применения перехватчиков (другие примеры будут описаны позднее):




```
public abstract class CaffeineBeverageWithHook {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        if (customerWantsCondiments()) {
            addCondiments();
        }
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }

    boolean customerWantsCondiments() {
        return true;
    }
}
```

Добавляем условную конструкцию, результат которой определяется вызовом конкретного метода customerWantsCondiments(). Только если вызов вернет true, мы вызываем addCondiments().

Метод с (почти) пустой реализацией по умолчанию: просто возвращает true, и не делает ничего более.

Перехватчик: субкласс может переопределить этот метод, но не обязан этого делать.

Использование перехватчиков

Чтобы использовать метод-перехватчик, мы переопределяем его в субклассе. В данном случае перехватчик управляет выполнением классом CaffeineBeverage определенной части алгоритма, а именно добавками к напитку.

Как узнать, нужно ли класть клиенту в кофе сахар/молоко? Да просто спросить!

```
public class CoffeeWithHook extends CaffeineBeverageWithHook {

    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }

    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }

    public boolean customerWantsCondiments() {
        String answer = getUserInput();

        if (answer.toLowerCase().startsWith("y")) {
            return true;
        } else {
            return false;
        }
    }

    private String getUserInput() {
        String answer = null;

        System.out.print("Would you like milk and sugar with your coffee (y/n)? ");

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try {
            answer = in.readLine();
        } catch (IOException ioe) {
            System.err.println("IO error trying to read your answer");
        }
        if (answer == null) {
            return "no";
        }
        return answer;
    }
}
```

Здесь вы переопределяете перехватчик и задаете нужную функциональность.

Предлагаем пользователю принять решение и возвращаем true или false в зависимости от полученных данных.

В этом фрагменте мы спрашиваем пользователя, нужно ли добавить в напиток сахар/молоко. Входные данные читаются из командной строки.

Проверяем, как работает код

Вода закипает... Следующая тестовая программа приготовит горячий чай и кофе.

```
public class BeverageTestDrive {
    public static void main(String[] args) {

        TeaWithHook teaHook = new TeaWithHook();
        CoffeeWithHook coffeeHook = new CoffeeWithHook();

        System.out.println("\nMaking tea...");
        teaHook.prepareRecipe();

        System.out.println("\nMaking coffee...");
        coffeeHook.prepareRecipe();
    }
}
```

Создаем чай.

Создаем кофе.

Вызываем prepareRecipe() для обоих!

Запускаем...

```
File Edit Window Help send-more-honesttea

%java BeverageTestDrive

Making tea...
Boiling water
Steeping the tea
Pouring into cup
Would you like lemon with your tea (y/n)? y
Adding Lemon

Making coffee...
Boiling water
Dripping Coffee through filter
Pouring into cup
Would you like milk and sugar with your coffee (y/n)? n
%
```

Чашка горячего чая... и, конечно, с лимоном!

И кофе тоже — но без добавок, вредных для фигуры!

И знаете что? Мы с вами согласимся. Но признайте: до того, как вам пришла в голову подобная мысль, это был классный пример использования перехватчиков для условного управления выполнением алгоритма в абстрактном классе. Верно?

Несомненно, вы сможете придумать много других, более реалистичных сценариев использования шаблонных методов и перехватчиков в своем коде.

Часто Задаваемые Вопросы

В: Как при создании шаблонного метода определить, когда использовать абстрактные методы, а когда — перехватчики?

О: Используйте абстрактные методы, если субкласс ДОЛЖЕН предоставить реализацию метода или шага алгоритма. Перехватчики используются для необязательных частей алгоритма.

В: Для чего нужны перехватчики?

О: Как мы уже говорили, при помощи перехватчика субкласс может реализовать необязательную часть алгоритма. Если эта часть не важна для реализации субкласса, он ее пропускает. Также перехватчик может дать субклассу возможность среагировать на предстоящий или только что выполненный шаг шаблонного метода. Ранее мы уже рассмотрели пример, в котором перехватчик давал субклассу возможность принять решение вместо абстрактного класса.

В: Должен ли субкласс реализовать все абстрактные методы абстрактного суперкласса?

О: Да, каждый конкретный субкласс определяет полный набор абстрактных методов и предоставляет полную реализацию неопределенных шагов алгоритма шаблонного метода.

В: Вероятно, количество абстрактных методов должно быть минимальным, иначе их реализация в субклассе потребует слишком большого объема работы.

О: Об этом стоит помнить при написании шаблонных методов. Иногда эта цель достигается за счет укрупнения шагов алгоритма, но это приводит к потере гибкости.

Также следует помнить, что некоторые шаги могут быть необязательными; реализуйте их в виде перехватчиков (вместо абстрактных методов), чтобы упростить реализацию субклассов.

Голливудский принцип

Представляем новый принцип проектирования, который называется «Голливудский принцип».

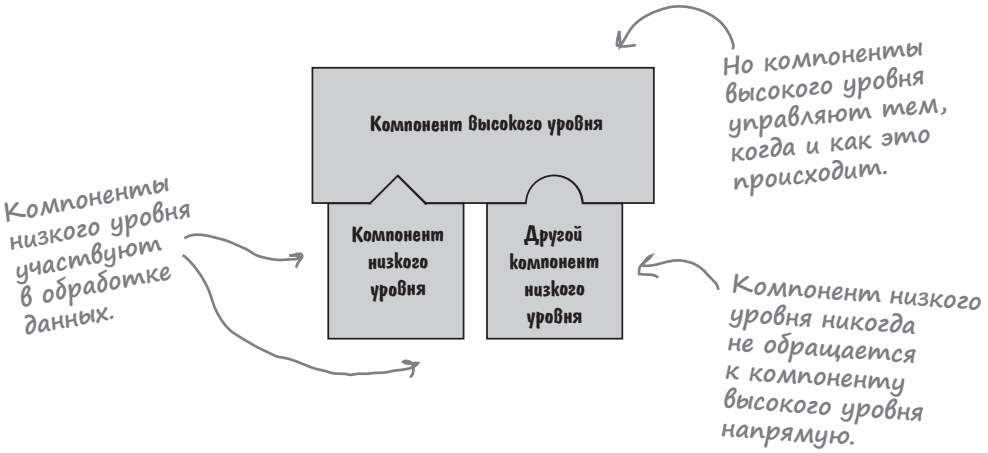
Голливудский принцип

Не вызывайте нас — мы вас сами вызовем.

Легко запоминается, верно? Но какое отношение этот принцип имеет к ОО-проектированию?

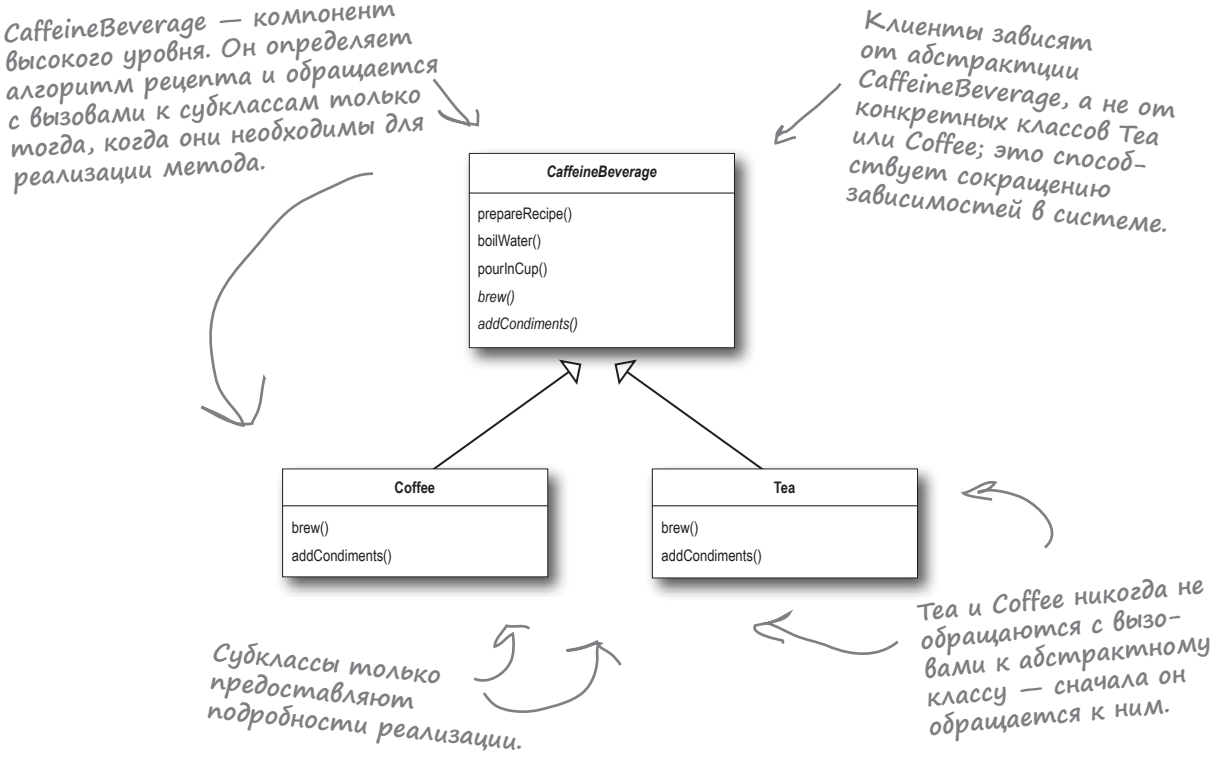
Голливудский принцип помогает предотвратить «разложение зависимостей» — явление, при котором компоненты высокого уровня зависят от компонентов низкого уровня, которые зависят от компонентов низкого уровня, которые зависят... и т. д. Разобраться в архитектуре такой системы очень трудно.

Голливудский принцип позволяет компонентам низкого уровня подключаться к системе, но компоненты высокого уровня сами определяют, когда и как они должны использоваться. Иначе говоря, компоненты высокого уровня запрещают компонентам низкого уровня «проявлять инициативу».



Голливудский принцип и Шаблонный Метод

Связь между Голливудским принципом и Шаблонным Методом достаточно очевидна: при проектировании с использованием паттерна Шаблонный Метод мы фактически запрещаем субклассам обращаться с вызовами к суперклассу. Каким образом? Давайте еще раз взглянем на архитектуру CaffeineBeverage:



МОЗГОВОЙ ШТУРМ

Какие еще паттерны используют Голливудский принцип?

Фабричный Метод, Наблюдатель; другие варианты?

Часто Задаваемые Вопросы

В: Как Голливудский принцип связан с принципом инверсии зависимостей, который мы рассматривали несколько глав назад?

О: Принцип инверсии зависимостей учит нас избегать использования конкретных классов и по возможности работать с абстракциями. Голливудский принцип направлен на построение инфраструктур и компонентов, в которых компоненты низкого уровня могут участвовать в вычислениях без формирования зависимостей между компонентами низкого и высокого уровня. Таким образом, оба принципа обеспечивают логическую изоляцию, но принцип инверсии зависимостей является более сильным и общим утверждением относительно того, как избегать зависимостей в архитектуре.

В: Запрещено ли компоненту низкого уровня вызывать методы компонента высокого уровня?

О: Нет. Более того, компонент низкого уровня часто в конечном итоге вызывает метод, определенный на более высоком уровне иерархии наследования. Мы лишь хотим избежать создания циклических зависимостей между компонентами высокого и низкого уровня.

Кто и что делает?

Соедините каждый паттерн с его описанием:

Паттерн	Описание
Шаблонный Метод	Инкапсуляция взаимозаменяемых вариантов поведения и выбор нужного варианта посредством делегирования
Стратегия	Субклассы определяют реализацию шагов алгоритма
Фабричный Метод	Субклассы решают, какие конкретные классы создавать

Шаблонные методы на практике

Паттерн Шаблонный Метод очень часто применяется на практике; вы найдете множество примеров его применения в готовом коде. Будьте осмотрительны, потому что многие реализации шаблонных методов заметно отличаются от «канонических» реализаций этого паттерна.

Столь широкое распространение этого паттерна объясняется тем, что он идеально подходит для создания инфраструктур, которые управляют общим ходом выполнения некоторой задачи, но при этом дают возможность пользователю указать, что конкретно должно происходить на каждом шаге алгоритма.

Давайте проведем небольшую экскурсию на природе (то есть в Java API)...

Сортировка на базе Шаблонного Метода

Какая операция часто выполняется с массивами? Конечно, сортировка!

Хорошо понимая этот факт, проектировщики класса Java Arrays предоставили нам удобный шаблонный метод для выполнения сортировки. Давайте посмотрим, как он работает:

Мы немного упростили код, чтобы было проще объяснять. Полную версию можно загрузить на сайте Sun.

Функциональность сортировки обеспечивается совместной работой двух методов.

Вспомогательный метод sort() создает копию массива и передает ее вместе с приемным массивом методу mergeSort(). Также передается длина массива и признак начала сортировки с первого элемента.

```
public static void sort(Object[] a) {
    Object aux[] = (Object[])a.clone();
    mergeSort(aux, a, 0, a.length, 0);
}
```

Метод mergeSort() содержит алгоритм сортировки, реализация которого зависит от метода compareTo(). За техническими подробностями обращайтесь к исходному коду Sun.

Шаблонный Метод.

```
private static void mergeSort(Object src[], Object dest[],
             int low, int high, int off)
{

    // a lot of other code here
    for (int i=low; i<high; i++){
        for (int j=i; j>low &&
             ((Comparable)dest[j-1]).compareTo((Comparable)dest[j])>0; j--)
        {
            swap(dest, j, j-1);
        }
    }
    // and a lot of other code here

}
```

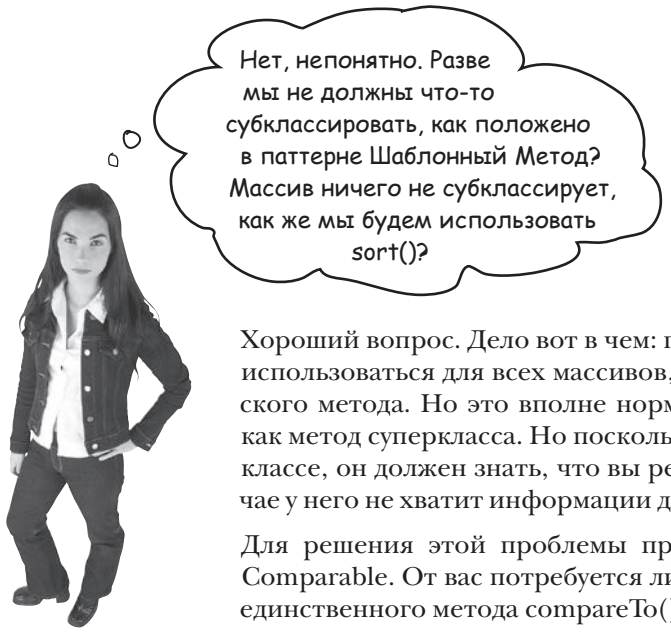
Конкретный метод, уже определенный в классе Arrays.

compareTo() — метод, который необходимо реализовать для «заполнения пробелов» в Шаблонном Методе.

Сортируем уток...

Массив объектов Duck, который нужно отсортировать.

Допустим, у вас имеется массив объектов Duck, и этот массив нужно отсортировать. Как это сделать? Шаблонный метод sort из класса Arrays определяет алгоритм, но мы должны указать ему, как сравнить два объекта, а для этого нужно реализовать метод compareTo()... Понятно?

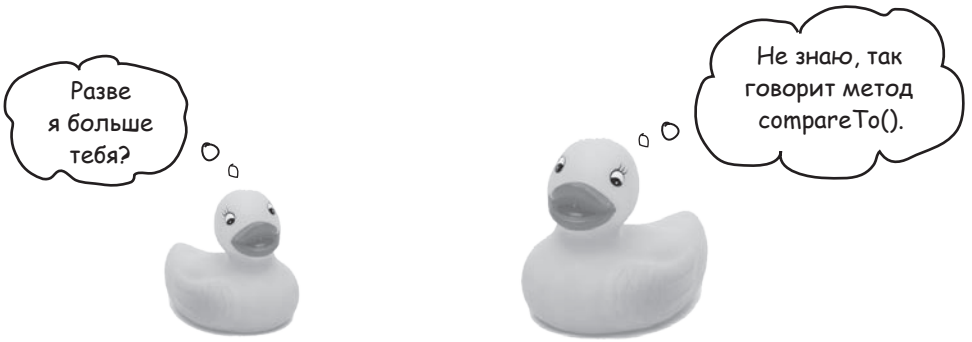


Хороший вопрос. Дело вот в чем: проектировщики sort() хотели, чтобы метод мог использоваться для всех массивов, поэтому они реализовали sort() в виде статического метода. Но это вполне нормальное решение — оно работает почти так же, как метод суперкласса. Но поскольку метод sort() все-таки не определяется в суперклассе, он должен знать, что вы реализовали метод compareTo(); в противном случае у него не хватит информации для определения полного алгоритма сортировки.

Для решения этой проблемы проектировщики воспользовались интерфейсом Comparable. От вас потребуется лишь реализовать этот интерфейс, состоящий из единственного метода compareTo().

Что делает метод compareTo()?

Метод compareTo() сравнивает два объекта и возвращает информацию об их соотношении (первый объект меньше второго, больше либо равен ему). Метод sort() использует его для сравнения объектов в массиве.



Сравнение объектов Duck

Итак, для сортировки объектов Duck необходимо реализовать метод compareTo(); тем самым вы предоставите классу Arrays информацию, необходимую для завершения алгоритма сортировки.

Реализация класса Duck выглядит так:

При отсутствии реального субклассирования класс должен реализовать интерфейс Comparable.

```
public class Duck implements Comparable {
    String name;
    int weight;

    public Duck(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    public String toString() {
        return name + " weighs " + weight;
    }

    public int compareTo(Object object) {

        Duck otherDuck = (Duck)object;

        if (this.weight < otherDuck.weight) {
            return -1;
        } else if (this.weight == otherDuck.weight) {
            return 0;
        } else { // this.weight > otherDuck.weight
            return 1;
        }
    }
}
```

Переменные экземпляров.

Объект Duck просто выводит значения переменных name и weight.

Метод необходим для сортировки...

Метод compareTo() получает другой объект Duck и сравнивает его с ТЕКУЩИМ объектом Duck.

Здесь определяется правило сравнения объектов Duck. Если значение переменной weight ТЕКУЩЕГО объекта Duck меньше значения weight объекта otherDuck, метод возвращает -1; если они равны — возвращает 0; а если больше — возвращает 1.

Пример сортировки

Тестовая программа для сортировки объектов Duck...

```
public class DuckSortTestDrive {
    public static void main(String[] args) {
        Duck[] ducks = {
                        new Duck("Daffy", 8),
                        new Duck("Dewey", 2),
                        new Duck("Howard", 7),
                        new Duck("Louie", 2),
                        new Duck("Donald", 10),
                        new Duck("Huey", 2)
         };

        System.out.println("Before sorting:");
        display(ducks);

        Arrays.sort(ducks);

        System.out.println("\nAfter sorting:");
        display(ducks);
    }

    public static void display(Duck[] ducks) {
        for (Duck d : ducks) {
            System.out.println(d);
        }
    }
}
```

Создаем массив объектов Duck.

Мы вызываем статический метод sort класса Arrays и передаем ему массив объектов Duck.

Выводим значения name и weight.

Сортировка!

(Повторно) выводим значения name и weight.

Давайте начнем сортировку!

```
File Edit Window Help DonaldNeedsToGoOnADiet
%java DuckSortTestDrive

Before sorting:
Daffy weighs 8
Dewey weighs 2
Howard weighs 7
Louie weighs 2
Donald weighs 10
Huey weighs 2

After sorting:
Dewey weighs 2
Louie weighs 2
Huey weighs 2
Howard weighs 7
Daffy weighs 8
Donald weighs 10
%
```

До сортировки

После сортировки

Как сортируются объекты Duck

За сценой

Давайте проследим за тем, как работает метод sort() класса Arrays. Нас прежде всего интересует то, как шаблонный метод определяет алгоритм и как в некоторых точках алгоритма он обращается к Duck за реализацией текущего шага...

```
for (int i=low; i<high; i++){
        ... compareTo() ...
        ... swap() ...
}
```

1. Прежде всего понадобится массив объектов Duck:

```
Duck[] ducks = {new Duck("Daffy", 8), ... };
```

2. Затем мы вызываем шаблонный метод sort() класса Arrays и передаем ему созданный массив:

```
Arrays.sort(ducks);
```

Метод sort() (и вспомогательный метод mergeSort()) определяют процедуру сортировки.

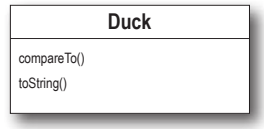
Метод sort() определяет алгоритм, и никакой класс этого изменить не может. Он рассчитывает на то, что класс, реализующий Comparable, предоставит реализацию compareTo().

3. Чтобы отсортировать массив, метод последовательно сравнивает его элементы до тех пор, пока весь список не будет приведен к порядку сортировки. Чтобы узнать, как сравнивать два объекта Duck, метод sort обращается за помощью к методу compareTo() класса Duck. Метод compareTo() вызывается для первого объекта Duck и получает объект Duck, с которым он сравнивается:

```
ducks[0].compareTo(ducks[1]);
```

Первый объект Duck

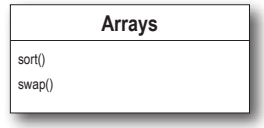
Объект Duck, с которым он сравнивается



Не связаны наследованием (в отличие от типичного шаблонного метода).

4. Если объекты Duck нарушают порядок сортировки, они меняются местами при помощи конкретного метода swap() класса Arrays:

```
swap()
```



5. Метод sort продолжает сравнивать и менять местами объекты Duck, пока элементы массива не будут располагаться в правильном порядке!

Часто Задаваемые Вопросы

В: А этот пример вообще относится к паттерну Шаблонный Метод?

О: В описании паттерна говорится об определении алгоритма с возможностью определения реализации отдельных его шагов в субклассах — сортировка массива под это описание определенно не подходит! Но, как мы знаем, на практике паттерны часто приходится подгонять под ограничения контекста и реализации.

Проектировщики метода Arrays sort() столкнулись с препятствием: в общем случае субклассировать массив Java невозможно, а они хотели, чтобы метод sort мог использоваться для всех массивов (хотя разные массивы относятся к разным классам). Исходя из этого, они определили статический метод и поручили определение способа сортируемым объектам.

Таким образом, несмотря на отличия от «канонических» шаблонных методов, эта реализация вполне соответствует духу паттерна.

В: Такая реализация сортировки напоминает скорее паттерн Стратегия. Почему мы считаем ее Шаблонным Методом?

О: Вероятно, такое впечатление возникает из-за того, что в паттерне Стратегия используется композиция. И это в каком-то смысле верно, ведь мы *используем* объект Arrays для сортировки массива. Однако в паттерне Стратегия включаемый класс реализует весь алгоритм, а алгоритм сортировки, реализованный в Arrays, неполон; недостающий метод compareTo() должен быть предоставлен классом. В этом состоит его сходство с паттерном Шаблонный Метод.

В: Встречаются ли другие примеры шаблонных методов в Java API?

О: Да, встречаются. Например, в библиотеке java.io класс InputStream содержит метод read(), который реализуется субклассами и используется шаблонным методом read(byte b[], int off, int len).

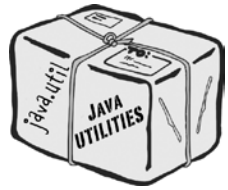
МОЗГОВОЙ ШТУРМ

Всем известно, что композиции следует отдавать предпочтение перед наследованием. Авторы реализации sort() решили не использовать наследование, а вместо этого реализовали sort() как статический метод, связываемый с Comparable посредством композиции на стадии выполнения. В чем преимущества такого решения? Недостатки? Как бы вы подошли к решению этой проблемы? Не усложняют ли эту задачу особенности массивов Java?

МОЗГОВОЙ^2 ШТУРМ

Вспомните другой паттерн, который представляет собой специализированную версию Шаблонного Метода. В этой специализации примитивные операции используются для создания и возвращения объектов. Что это за паттерн?

Шаблонный метод в JFrame



Экскурсия по шаблонным методам продолжается... Следующая остановка — JFrame!

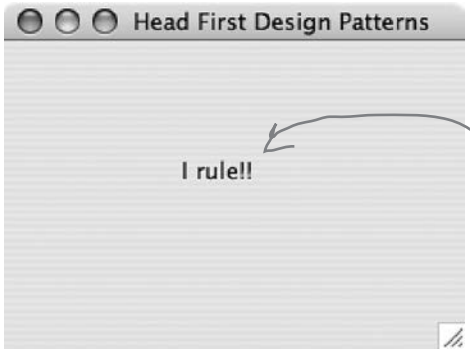
Для тех, кто еще не знаком с JFrame, поясняем: это основной контейнер Swing, наследующий метод paint(). По умолчанию paint() не делает ничего, потому что является *перехватчиком*! Переопределяя paint(), можно подключиться к алгоритму, используемому JFrame для перерисовки своей части экрана, и включить в JFrame свой графический вывод. Простейший пример использования JFrame для переопределения метода paint():

Расширяем класс JFrame, который содержит метод update(), управляющий перерисовкой экрана. Чтобы подключиться к этому алгоритму, мы переопределяем метод paint().

```
public class MyFrame extends JFrame {

    public MyFrame(String title) {
        super(title);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        this.setSize(300,300);
        this.setVisible(true);
    }

    public void paint(Graphics graphics) {
        super.paint(graphics);
        String msg = "I rule!!";
        graphics.drawString(msg, 100, 100);
    }

    public static void main(String[] args) {
        MyFrame myFrame = new MyFrame("Head First Design Patterns");
    }
}
```

Подробности несущественны! Просто некая инициализация...

Алгоритм перерисовки JFrame вызывает paint(). По умолчанию метод paint() не делает ничего... это перехватчик. Переопределенная версия paint() выводит сообщение в окне.



Сообщение, которое выводится на панели вследствие перехвата метода paint().

Аплеты

И последняя остановка нашей экскурсии: аплеты.

Как вам, вероятно, известно, аплетом называется мини-программа, выполняемая на веб-странице. Аплеты субклассируют Applet, а этот класс предоставляет несколько полезных перехватчиков. Рассмотрим некоторые из них:

```
public class MyApplet extends Applet {
    String message;

    public void init() {
        message = "Hello World, I'm alive!";
        repaint();
    }

    public void start() {
        message = "Now I'm starting up...";
        repaint();
    }

    public void stop() {
        message = "Oh, now I'm being stopped...";
        repaint();
    }

    public void destroy() {
        // Аплет уничтожается...
    }

    public void paint(Graphics g) {
        g.drawString(message, 5, 15);
    }
}
```

Перехватчик init позволяет выполнить необходимые действия при инициализации приложения.

repaint() — конкретный метод класса Applet, при помощи которого компоненты высокого уровня оповещаются о необходимости перерисовки аплета.

Перехватчик start позволяет приложению выполнить необходимые действия непосредственно перед отображением аплета на веб-странице.

При переходе к другой странице перехватчик stop дает приложению возможность сделать все необходимое для завершения работы.

Перехватчик destroy используется перед уничтожением приложения (например, при закрытии браузера). Здесь можно что-нибудь вывести, но какой смысл?

Посмотрите-ка сюда! Наш старый знакомый — метод paint()! Аплет тоже использует его как перехватчика.

Конкретные аплеты широко используют перехватчики для определения собственного поведения. Поскольку эти методы реализуются в виде перехватчиков, их реализация аплетом не обязательна.

Беседа у камина

Сравнение Шаблонного Метода и Стратегии.

Шаблонный Метод

Послушай, что ты делаешь в моей главе? Я думал, что ты общаешься только с занудами вроде Фабричного Метода.

Да я же пошутил! Хотя серьезно, что ты здесь делаешь? Уже восемь глав о тебе ни слуху, ни духу!

Раз уж тебя так долго не было, стоит напомнить читателю, кто ты такой.

Выходит, у нас много общего. Но моя задача несколько иная: я определяю общую структуру алгоритма, поручая часть работы субклассам. Это позволяет мне выбирать разные реализации отдельных шагов алгоритма, сохраняя под контролем его структуру. А ты, похоже, свои алгоритмы не контролируешь...

Стратегия

Полегче — ведь вы с Фабричным Методом вроде как родственники?

Я слышал, что твоя глава подходит к концу, и решил посмотреть, как дела. У нас много общего, и я решил, что смогу помочь...

Не знаю, право, — после главы 1 меня часто останавливают на улице и спрашивают: «Скажите, а вы случайно не тот паттерн...» Так что меня хорошо знают. Только ради тебя: я определяю семейство алгоритмов и обеспечиваю их взаимозаменяемость. Инкапсуляция позволяет легко использовать разные алгоритмы на стороне клиента.

Я бы не стал это называть *так*... Но как бы то ни было, я не ограничиваюсь применением наследования для реализаций алгоритмов. Я предоставляю клиентам выбор разных реализаций алгоритмов через композицию.

Шаблонный Метод

Да, я помню. Но я лучше контролирую свой алгоритм и не допускаю дублирования кода. Собственно, если все части моего алгоритма одинаковы, кроме, допустим, одной строки, мои классы намного эффективнее твоих. Весь дублирующийся код выделяется в суперкласс и совместно используется всеми субклассами.

Рад за тебя, но не будем забывать, что *меня* используют чаще других паттернов. Почему? Потому что я предоставляю основной механизм повторного использования кода, позволяющий задавать поведение в субклассах. Сам понимаешь, этот механизм идеально подходит для создания инфраструктур.

Это почему? Мой суперкласс объявляется абстрактным.

Да-да, я уже говорил: рад за тебя. Спасибо, что зашел, но мне нужно добраться до конца этой главы.

Понятно. Не вызывайте нас, мы вас сами вызовем...

Стратегия

Иногда ты бываешь более эффективным (ненамного) и требуешь меньшего количества объектов. И *иногда* бываешь менее сложным по сравнению с моей моделью делегирования... но я более гибок, потому что я использую композицию. Со мной клиенты могут изменять алгоритмы во время выполнения, просто переключаясь на другой объект стратегии. Не зря же именно *меня* выбрали для главы 1!

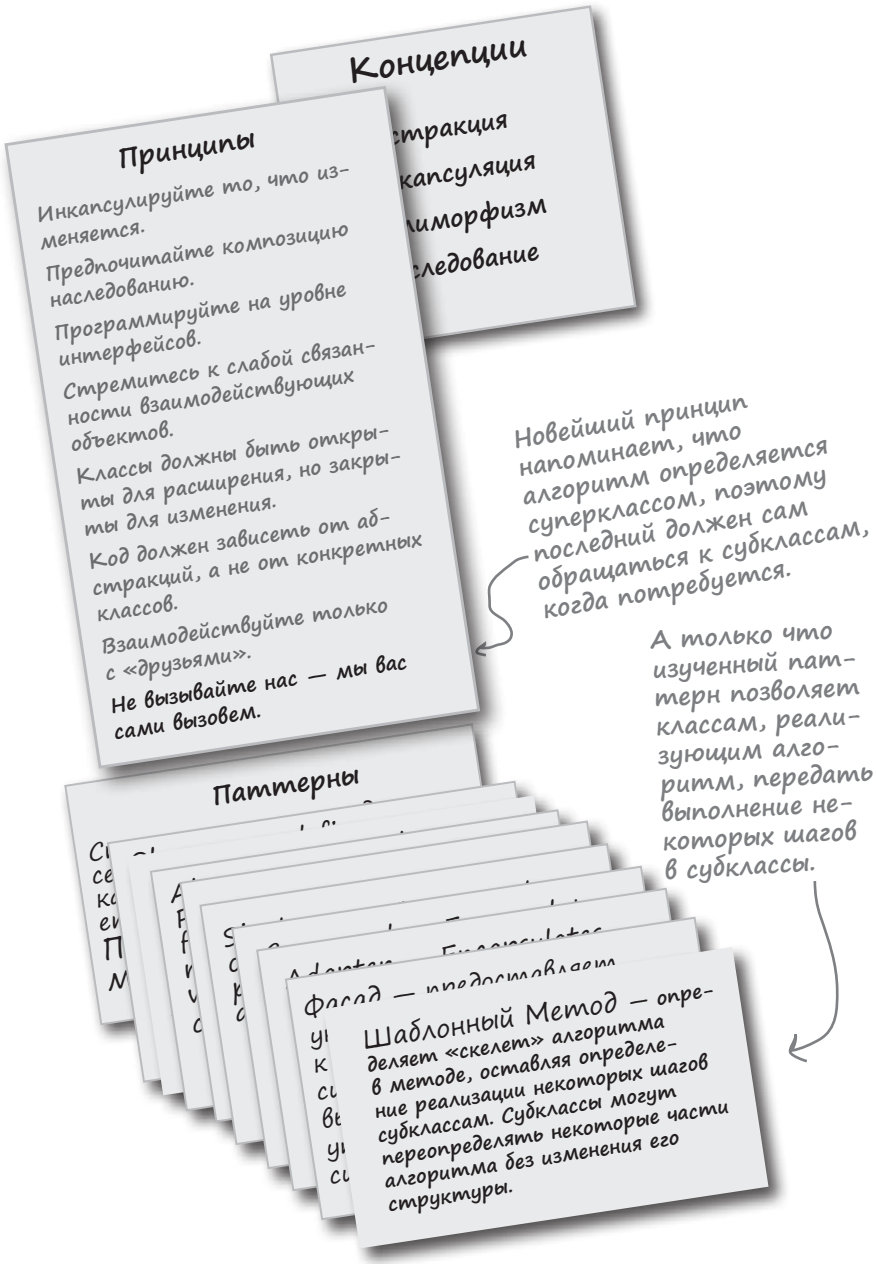
Да, пожалуй... А как насчет зависимостей? У тебя их явно больше, чем у меня.

Но тебе приходится зависеть от реализации в суперклассе методов, являющихся частью твоего алгоритма. Я не завишу ни от кого; я могу реализовать весь алгоритм самостоятельно!

Ладно, ладно, не обижайся. Не буду мешать, но если тебе что-то понадобится — только скажи. Всегда готов помочь.

Новые инструменты

Ваш инструментарий дополнился паттерном Шаблонный Метод. Этот паттерн позволяет организовать повторное использование кода с сохранением контроля над алгоритмами.

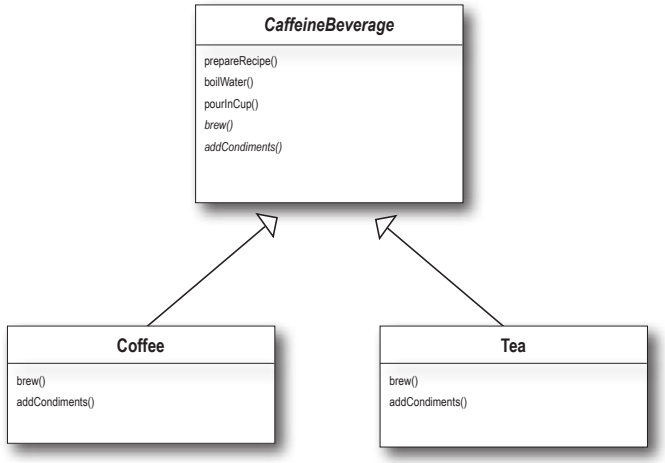


КЛЮЧЕВЫЕ МОМЕНТЫ

- Шаблонный Метод определяет основные шаги алгоритма, оставляя субклассам возможность определения реализации этих шагов.
- Паттерн Шаблонный Метод играет важную роль в повторном использовании кода.
- Абстрактный класс Шаблонного Метода может определять конкретные методы, абстрактные методы и перехватчики.
- Абстрактные методы реализуются субклассами.
- Перехватчики не делают ничего или определяют поведение по умолчанию в абстрактном классе, но могут переопределяться в субклассах.
- Чтобы субкласс не мог изменить алгоритм в Шаблонном Методе, объявите последний с ключевым словом final.
- Голливудский принцип указывает на то, что решения должны приниматься модулями высокого уровня, которые знают, как и когда обращаться с вызовами к модулям низкого уровня.
- Паттерны Стратегия и Шаблонный метод инкапсулируют алгоритмы; один использует наследование, а другой — композицию.
- Фабричный Метод является специализированной версией Шаблонного Метода.

Возьми в руку карандаш
Решение

Нарисуйте новую диаграмму классов после перемещения реализации prepareRecipe() в класс CaffeineBeverage:



Кто и что делает?
Решение

Соедините каждый паттерн с его описанием:

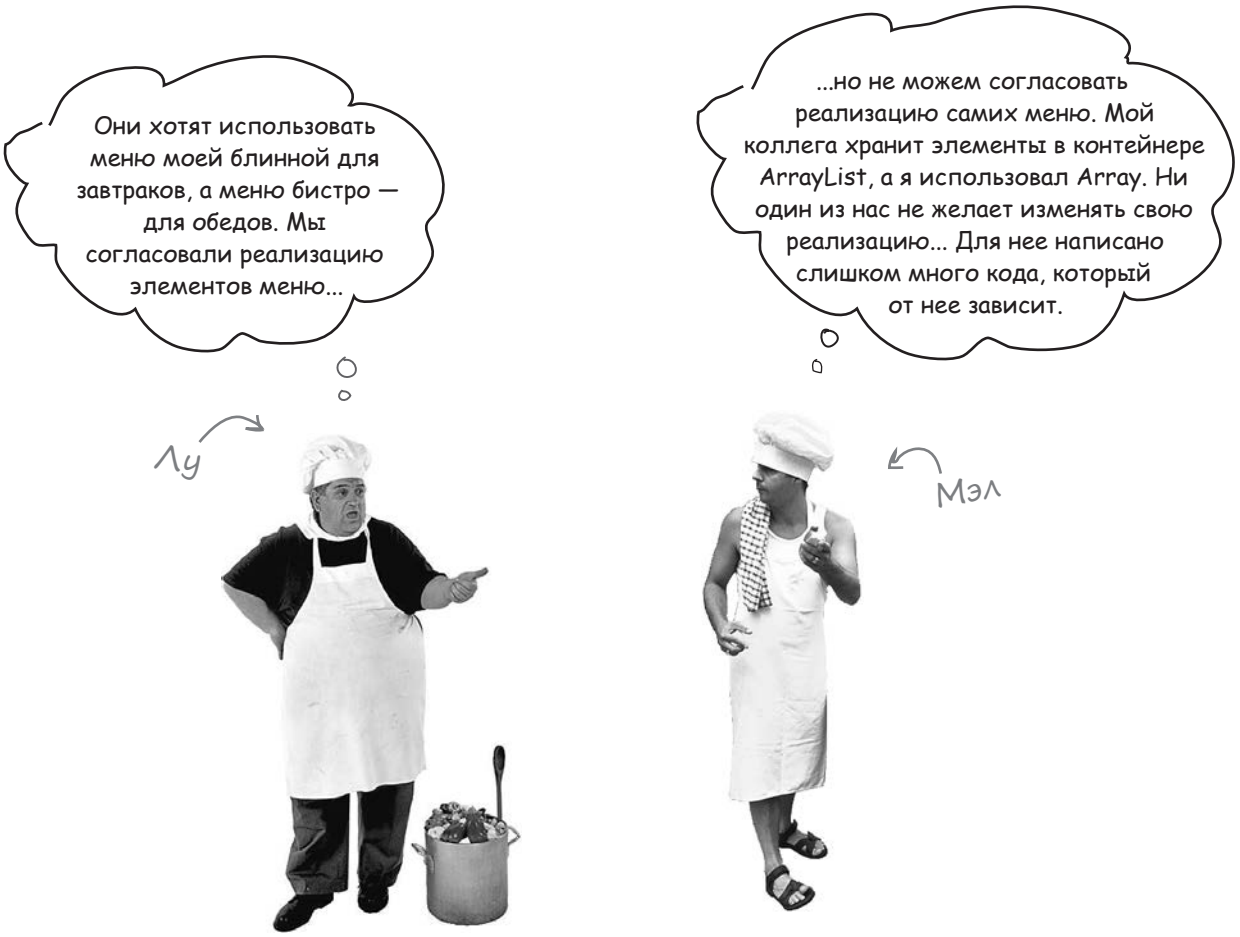
Паттерн	Описание
Шаблонный Метод	Инкапсуляция взаимозаменяемых вариантов поведения и выбор нужного варианта посредством делегирования
Стратегия	Субклассы определяют реализацию шагов алгоритма
Фабричный Метод	Субклассы решают, какие конкретные классы создавать

Управляемые коллекции

Существует много способов создания коллекций. Объекты можно разместить в контейнере Array, Stack, List, Hashmap — выбирайте сами. Каждый способ обладает своими достоинствами и недостатками. Но в какой-то момент клиенту потребуется перебрать все эти объекты, и, когда это произойдет, собираетесь ли вы раскрывать реализацию коллекции? Надеемся, нет! Это было бы крайне непрофессионально. В этой главе вы узнаете, как предоставить клиенту механизм перебора объектов без раскрытия информации о способе их хранения. Также в ней будут описаны способы создания суперколлекций. А если этого недостаточно, вы узнаете кое-что новое относительно обязанностей объектов.

Сенсация в Объективиле: бистро объединяется с блинной!

Отличные новости! Теперь мы можем заказать аппетитный завтрак с блинчиками и обед в одном месте! Но, похоже, у поваров возникла небольшая проблема...



Проверяем элементы меню

По крайней мере, Лу и Мэл согласовали реализацию класса MenuItem. Рассмотрим содержимое обоих меню, а заодно познакомимся с реализацией.

В меню бистро много обеденных блюд, а меню блинной состоит из завтраков. С каждым элементом меню связывается название, описание и цена.

```
public class MenuItem {
    String name;
    String description;
    boolean vegetarian;
    double price;

    public MenuItem(String name,
                    String description,
                    boolean vegetarian,
                    double price)
    {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }
}
```

Объект MenuItem содержит несколько полей: имя, описание, флаг вегетарианского блюда и цена. Все эти значения передаются конструктору для инициализации объекта MenuItem.

Методы для чтения/записи полей элемента меню.

Две реализации меню

Давайте разберемся, о чем спорят Лу и Мэл. Оба повара потратили немало времени и сил на написание кода хранения элементов меню, и другого кода, который от него зависит.

Реализация меню блинной.

```
public class PancakeHouseMenu {
    ArrayList<MenuItem> menuItems;

    public PancakeHouseMenu() {
        menuItems = new ArrayList<MenuItem>();

        addItem("K&B's Pancake Breakfast",
            "Pancakes with scrambled eggs, and toast",
            true,
            2.99);

        addItem("Regular Pancake Breakfast",
            "Pancakes with fried eggs, sausage",
            false,
            2.99);

        addItem("Blueberry Pancakes",
            "Pancakes made with fresh blueberries",
            true,
            3.49);

        addItem("Waffles",
            "Waffles, with your choice of blueberries or strawberries",
            true,
            3.59);
    }
    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.add(menuItem);
    }

    public ArrayList<MenuItem> getMenuItems() {
        return menuItems;
    }

    // другие методы
}
```

Лу хранит элементы меню в ArrayList.

Каждый элемент меню включается в ArrayList в конструкторе.

Для каждого объекта MenuItem задается имя, описание, признак вегетарианского блюда и цена.

Чтобы добавить новый элемент меню, Лу создает новый объект MenuItem, задает все необходимые аргументы и включает созданный объект в ArrayList.

Метод getMenuItems() возвращает список элементов меню.

Лу написал большой объем кода, зависящего от реализации ArrayList. И он не хочет переписывать весь код заново!

Какой еще Arraylist... Я выбрал НОРМАЛЬНЫЙ массив, чтобы ограничить максимальный размер меню.

А вот реализация Мэла.

```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    public DinerMenu() {
        menuItems = new MenuItem[MAX_ITEMS];

        addItem("Vegetarian BLT",
            "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99);
        addItem("BLT",
            "Bacon with lettuce & tomato on whole wheat", false, 2.99);
        addItem("Soup of the day",
            "Soup of the day, with a side of potato salad", false, 3.29);
        addItem("Hotdog",
            "A hot dog, with saurkraut, relish, onions, topped with cheese",
            false, 3.05);
        // a couple of other Diner Menu items added here
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        if (numberOfItems >= MAX_ITEMS) {
            System.err.println("Sorry, menu is full!  Can't add item to menu");
        } else {
            menuItems[numberOfItems] = menuItem;
            numberOfItems = numberOfItems + 1;
        }
    }

    public MenuItem[] getMenuItems() {
        return menuItems;
    }

    // other menu methods here
}
```

Мэл выбрал другой подход: он использует массив Array, чтобы ограничить максимальный размер меню.

Мэл тоже создает элементы меню в конструкторе при помощи вспомогательного метода addItem().

Метод addItem() получает все параметры, необходимые для создания MenuItem, и создает объект. Он также проверяет, не нарушает ли новый объект максимальный размер массива.

Мэл ограничивает размер меню, чтобы не запоминать слишком много рецептов.

Метод getMenuItems() возвращает массив элементов меню.

Мэл ТОЖЕ написал большой объем кода, зависящего от выбранной им реализации меню. И он слишком занят, чтобы переписывать этот код заново.

Какие проблемы создает наличие двух разных реализаций меню?

Чтобы понять, какие сложности возникают с двумя разными представлениями меню, мы попробуем реализовать клиента, использующего оба меню. Допустим, новая компания, возникшая в результате слияния, наняла вас для создания официантки с поддержкой Java (не забывайте, что вы находитесь в Объективиле!). Спецификация требует, чтобы официантка могла при необходимости напечатать сокращенное меню и даже определить, является ли блюдо вегетарианским, не обращаясь к повару!

Сначала просмотрим спецификацию, а затем шаг за шагом разберемся, что потребуется для ее реализации...

Официантка с поддержкой Java.

Спецификация официантки с поддержкой Java

Официантка с поддержкой Java: проект «Элис»

```
printMenu()
   — выводит каждый элемент меню

printBreakfastMenu()
   — выводит только блюда завтраков

printLunchMenu()
   — выводит только обеденные блюда

printVegetarianMenu()
   — выводит все вегетарианские блюда

isItemVegetarian(name)
   — по названию блюда возвращает true, если
оно является вегетарианским, или false в про-
тивном случае
```

Спецификация официантки.

Начнем с реализации метода printMenu():

1. Чтобы вывести полное меню, необходимо вызвать метод getMenuItems() для всех элементов обеих реализаций. Обратите внимание: методы возвращают разные типы:

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList<MenuItem> breakfastItems = pancakeHouseMenu.getMenuItems();

DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();
```

Методы внешне похожи, но вызовы возвращают разные типы.

Здесь проявляются различия реализации: блюда для завтрака хранятся в ArrayList, а обеденные блюда — в Array.

2. Чтобы вывести меню блинной, мы перебираем элементы контейнера ArrayList, а для вывода меню бистро перебираются элементы Array.

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}

for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}
```

Теперь нам придется написать два разных цикла для перебора двух реализаций меню...

...один цикл для ArrayList...

другой — для Array.

3. Реализация каждого метода будет представлять собой вариацию на эту тему — содержимое двух меню будет перебираться в двух разных циклах. А если вдруг добавится новый ресторан со своей реализацией, то в программе будут использоваться *три* разных цикла.

Возьми в руку карандаш

Какие из следующих утверждений относятся к нашей реализации printMenu()?

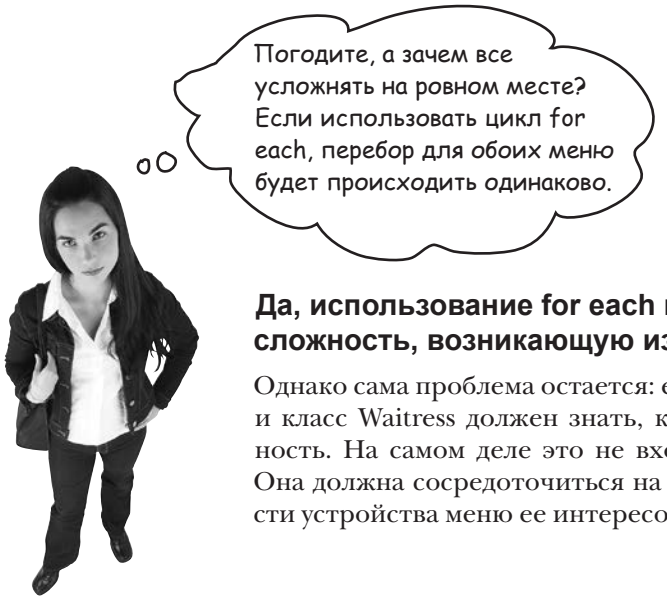
- ❑ A. Мы программируем для конкретных реализаций PancakeHouseMenu и DinerMenu, а не для интерфейсов.
- ❑ B. Официантка не реализует Java Waitress API, а следовательно, не соответствует стандарту.
- ❑ C. Если мы решим перейти с DinerMenu на другое меню с реализацией на базе Hashtable, нам придется изменять большой объем кода.
- ❑ D. Официантка должна знать, как в каждом объекте меню организована внутренняя коллекция элементов, а это нарушает инкапсуляцию.
- ❑ E. В реализации присутствует дублирование кода: метод printMenu() содержит два разных цикла для перебора двух разновидностей меню. А при появлении третьего меню понадобится еще один цикл.
- ❑ F. Реализация не использует язык MXML (Menu XML), что снижает ее универсальность.

Что дальше?

Мы оказались в затруднительном положении. Мэл и Лу не желают изменять свои реализации, потому что им придется переписать большой объем кода в соответствующих классах меню. Но если ни один из них не уступит, наша реализация официантки окажется сложной в сопровождении и расширении.

Хорошо бы найти механизм, позволяющий им реализовать единый интерфейс для своих меню (они и так достаточно близки, если не считать возвращаемого типа метода getMenuItems()). Это позволит нам свести к минимуму конкретные ссылки, а также избавиться от повторения циклов при переборе элементов меню.

Заманчиво, верно? Но как это сделать?



Да, использование for each позволит замаскировать сложность, возникающую из-за разных видов перебора.

Однако сама проблема остается: есть две разных реализации меню, и класс Waitress должен знать, как реализована каждая разновидность. На самом деле это не входит в обязанности официантки. Она должна сосредоточиться на обслуживании клиентов, а тонкости устройства меню ее интересовать вообще *не должны*.

Даже если использовать цикл for each для перебора меню, Waitress все равно нужно знать тип каждого меню.

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList<MenuItem> breakfastItems = pancakeHouseMenu.getMenuItems();

DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();

for (MenuItem menuItem : breakfastItems) {
    System.out.print(menuItem.getName());
    System.out.println("\t\t" + menuItem.getPrice());
    System.out.println("\t" + menuItem.getDescription());
}
for (MenuItem menuItem : lunchItems) {
    System.out.print(menuItem.getName());
    System.out.println("\t\t" + menuItem.getPrice());
    System.out.println("\t" + menuItem.getDescription());
}
```

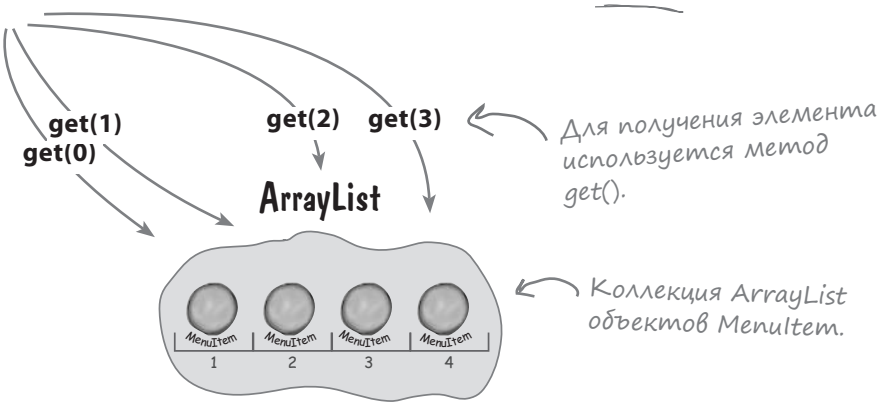
Наша цель — по возможности оградить Waitress от конкретных реализаций меню. Как вы вскоре увидите, существует другой, более эффективный способ.

Как инкапсулировать перебор элементов?

Инкапсулируйте то, что изменяется — это едва ли не самое важное из всего, о чем говорится в книге. Понятно, что изменяется в данном случае: механизм перебора для разных коллекций объектов (элементов меню). Но как его инкапсулировать? Давайте подробно разберем эту идею...

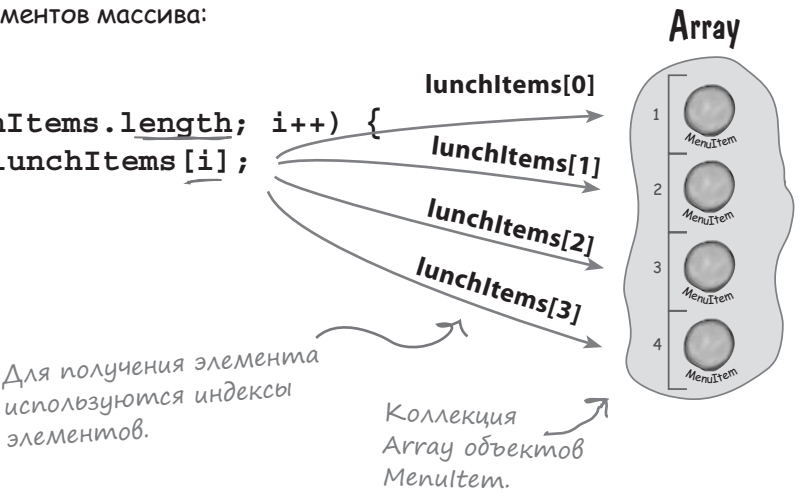
❶ Для перебора элементов ArrayList используются методы size() и get():

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
}
```



❷ Для перебора элементов массива используется поле length объекта Array и синтаксис выборки элементов массива:

```
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}
```

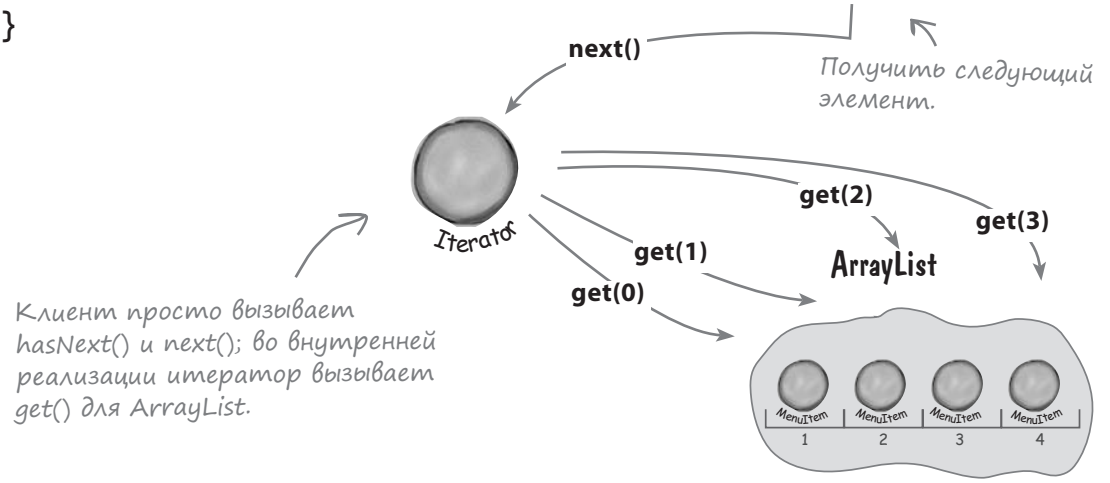


❸ Почему бы не создать объект (назовем его **итератором**), инкапсулирующий механизм перебора объектов в коллекции? Попробуем сделать это для ArrayList:

Запрашиваем у breakfastMenu итератор для перебора объектов MenuItem.

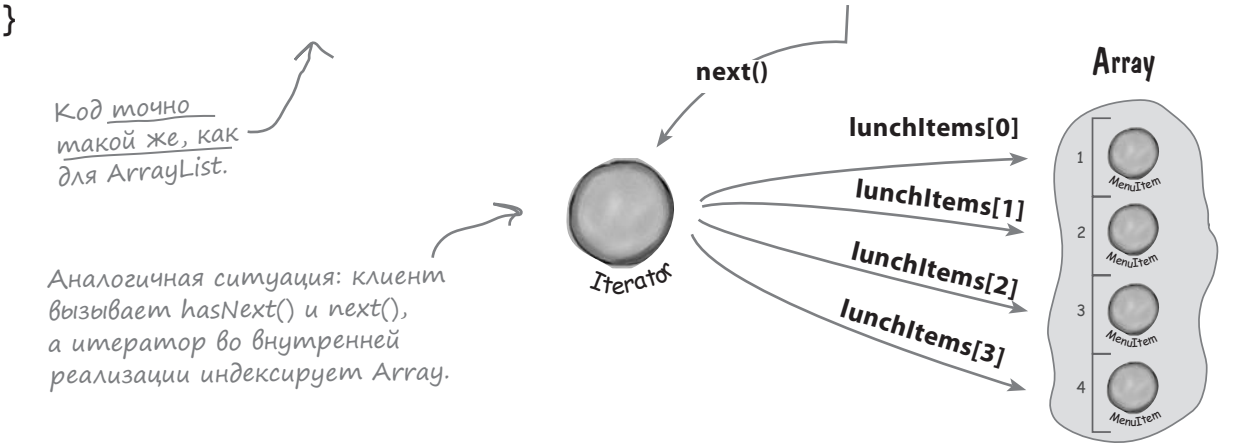
```
Iterator iterator = breakfastMenu.createIterator();

while (iterator.hasNext()) {
    MenuItem menuItem = iterator.next();
}
```

Пока остаются элементы...



❹ Теперь сделаем то же для Array:

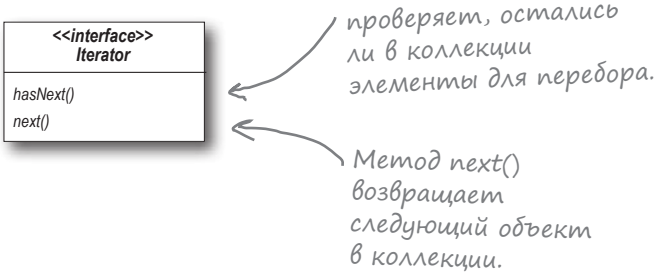
```
Iterator iterator = lunchMenu.createIterator();

while (iterator.hasNext()) {
    MenuItem menuItem = iterator.next();
}
```



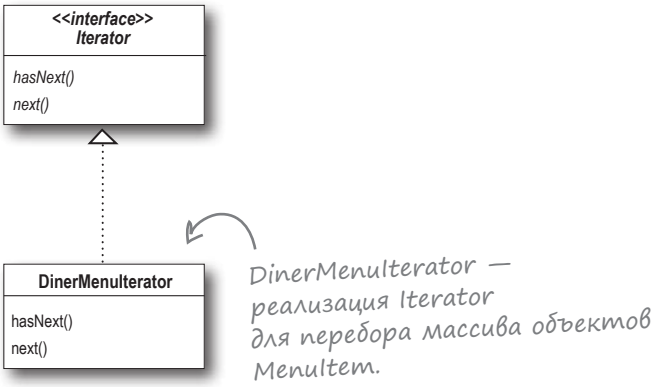
Паттерн Итератор

Похоже, наш план инкапсуляции перебора элементов вполне реален. И как вы, вероятно, уже догадались, для решения этой задачи существует паттерн проектирования, который называется Итератор.

Первое, что необходимо знать о паттерне Итератор, — то, что он зависит от специального интерфейса (допустим, Iterator). Одна из возможных форм интерфейса Iterator:



При наличии такого интерфейса мы можем реализовать итераторы для любых видов коллекций объектов: массивов, списков, хеш-карт*... Допустим, мы хотим реализовать итератор для коллекции Array из нашего примера. Реализация будет выглядеть так:



Давайте реализуем этот итератор и свяжем его с коллекцией DinerMenu, чтобы вы лучше поняли, как работает механизм перебора...

* Хеш-карта — та же хеш-таблица, только без синхронизации и с возможностью хранения null.

Добавление итератора в DinerMenu

Чтобы добавить итератор в DinerMenu, сначала необходимо определить интерфейс Iterator:

```
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

Два метода интерфейса:

hasNext() возвращает флаг, который указывает, остались ли в коллекции элементы для перебора...

...а метод next() возвращает следующий элемент.

Теперь необходимо реализовать конкретный итератор для коллекции DinerMenu:

```
public class DinerMenuIterator implements Iterator {
    MenuItem[] items;
    int position = 0;

    public DinerMenuIterator(MenuItem[] items) {
        this.items = items;
    }

    public MenuItem next() {
        MenuItem menuItem = items[position];
        position = position + 1;
        return menuItem;
    }

    public boolean hasNext() {
        if (position >= items.length || items[position] == null) {
            return false;
        } else {
            return true;
        }
    }
}
```

Реализуем интерфейс Iterator.

В переменной position хранится текущая позиция перебора в массиве.

Конструктор получает массив объектов, для перебора которых создается итератор.

Метод next() возвращает следующий элемент массива и увеличивает текущую позицию.

Метод hasNext() возвращает true, если в массиве еще остались элементы для перебора.

Так как для меню бистро выделен массив максимального размера, нужно проверить не только достижение границы массива, но и равенство следующего элемента null (признак последнего элемента).

Переработка DinerMenu с использованием итератора

Итак, у нас есть итератор. Пора интегрировать его с DinerMenu; для этого необходимо лишь добавить один метод, который создает объект DinerMenuIterator и возвращает его клиенту:

```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    // конструктор

    // addItem

    public MenuItem[] getMenuItems() {
        return menuItems;
    }

    public Iterator createIterator() {
        return new DinerMenuIterator(menuItems);
    }

    // другие методы
}
```

Метод getMenuItems() нам больше не понадобится. Более того, его присутствие нежелательно, потому что он раскрывает внутреннюю реализацию!

Метод createIterator() создает объект DinerMenuIterator для массива menuItems и возвращает его клиенту.

Метод возвращает интерфейс Iterator. Клиенту не нужно знать ни то, как коллекция menuItems хранится в DinerMenu, ни то, как реализован DinerMenuIterator. Клиент просто использует итератор для перебора элементов.

Упражнение

Самостоятельно реализуйте итератор для меню блинной (PancakeHouseIterator) и внесите изменения, необходимые для его интеграции с PancakeHouseMenu.

Изменение кода Waitress

Теперь поддержку итераторов необходимо интегрировать в реализацию официантки. Попутно мы избавимся от избыточности в коде. Процесс интеграции весьма прямолинеен: сначала мы создаем метод printMenu(), которому передается Iterator, а затем вызываем createIterator() для каждого меню, получаем Iterator и передаем его новому методу.

Новая версия с поддержкой итераторов.

В конструкторе передаются два объекта меню.

```
public class Waitress {
    PancakeHouseMenu pancakeHouseMenu;
    DinerMenu dinerMenu;

    public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();
        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }

    // другие методы
}
```

Метод printMenu() теперь создает два итератора, по одному для каждого меню.

А затем вызывает перегруженный метод printMenu() для каждого итератора.

Проверяем, остались ли еще элементы.

Получаем следующий элемент.

Выводим название, цену и описание текущего элемента.

В этой версии используется один цикл.

Перегруженный метод printMenu() использует Iterator для перебора и вывода элементов меню.

Тестирование кода

Система готова, можно переходить к тестированию. Напишем небольшую тестовую программу и посмотрим, как работает официантка...

```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();

        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu);

        waitress.printMenu();
    }
}
```

Создаем новые меню.

Создаем объект Waitress и передаем ему созданные объекты меню.

А затем выводим их содержимое.

Запускаем тест...

```
File Edit Window Help GreenEggs&Ham
% java DinerMenuTestDrive

MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles, with your choice of blueberries or strawberries

LUNCH
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hotdog, 3.05 -- A hot dog, with saurkraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

%
```

Сначала перебираем меню завтраков...

А затем меню обедов — и все в одном цикле.

Что мы сделали?

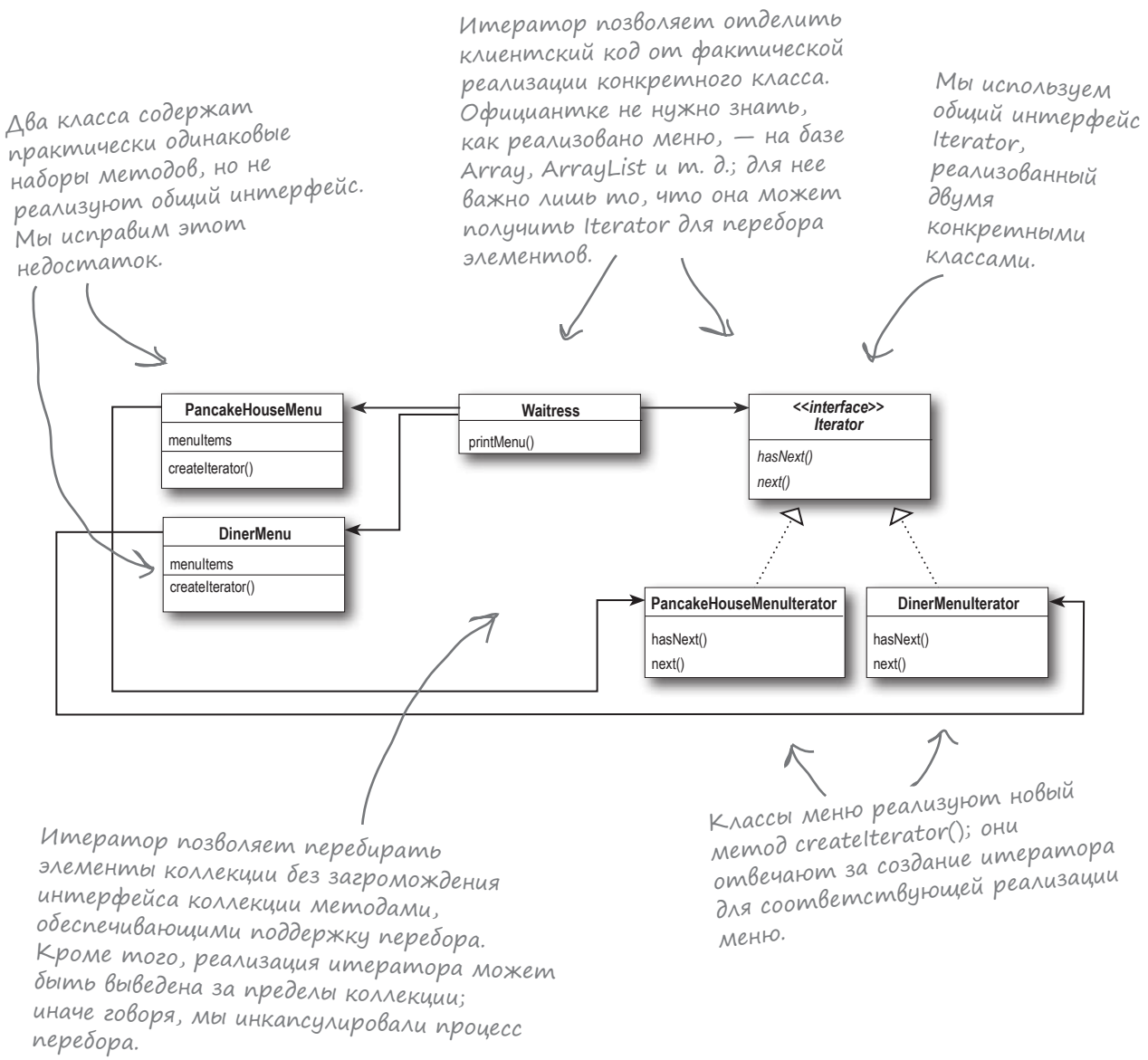
Прежде всего мы осчастливили поваров обоих заведений. Проблема с различиями успешно решена, а существующий код сохранен. После создания PancakeHouseMenuIterator и DinerMenuIterator им осталось лишь добавить метод createIterator() — и ничего более.

Заодно мы упростили свою работу — класс Waitress стал значительно проще в сопровождении и расширении. Давайте еще раз внимательно рассмотрим, что же было сделано, и проанализируем последствия:

Старая реализация, сложная в сопровождении	Новая реализация на базе итераторов
Плохая инкапсуляция работы с меню; мы видим, что одна реализация использует Array, а другая — ArrayList.	Реализации надежно инкапсулированы. Клиентский код не знает, как классы меню хранят свои коллекции элементов.
Для перебора элементов необходимы два цикла.	Достаточно одного цикла, который полиморфно обрабатывает элементы любой коллекции, реализующей Iterator.
Клиентский код привязан к конкретным классам ((MenuItem[ ] и ArrayList).	Клиентский код использует интерфейс (Iterator).
Клиентский код привязан к двум конкретным классам меню, несмотря на почти полное совпадение их интерфейсов.	Интерфейсы двух классов меню теперь полностью совпадают... Но код Waitress все еще привязан к двум конкретным классам меню. Эту проблему необходимо решить.

Текущее состояние дел

Прежде чем переходить к усовершенствованиям, рассмотрим текущую архитектуру «в перспективе».

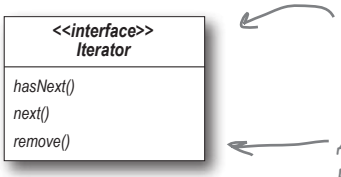


Вносим усовершенствования...

Итак, наборы методов PancakeHouseMenu и DinerMenu полностью совпадают, но мы еще не определили для них общий интерфейс. Сейчас мы сделаем это.

Законный вопрос: почему мы не воспользовались интерфейсом Iterator языка Java? Это было сделано для того, чтобы вы могли при необходимости построить итератор «с нуля». Теперь, когда это было сделано, мы переключимся на интерфейс Iterator языка Java, потому что он обладает рядом преимуществ по сравнению с нашей доморощенной реализацией. Что это за преимущества? Скоро увидите.

Для начала рассмотрим интерфейс java.util.Iterator:



Почти не отличается от предыдущей реализации.

Добавлен метод для удаления последнего элемента, возвращаемого методом next() коллекции.

Проще простого: нужно сменить интерфейс, расширяемый классами PancakeHouseMenuIterator и DinerMenuIterator, верно? Почти... Хотя на самом деле еще проще. Пакет java.util не только содержит собственный интерфейс Iterator, но и у класса ArrayList имеется метод iterator(), возвращающий итератор, — то есть для ArrayList реализовывать итератор вообще не нужно. Тем не менее для DinerMenu это сделать все равно придется, так как класс Array не содержит метода iterator() (или другого способа создания итератора для массива).

Часто Задаваемые Вопросы

В: А если мне не нужна возможность удаления элементов из коллекции?

О: Реализация метода remove() не считается обязательной. Но, разумеется, сам метод должен присутствовать, так как он является частью интерфейса Iterator. Если вы не разрешаете remove() в своем итераторе, инициируйте исключение java.lang.UnsupportedOperationException. В документации Iterator API указано, что это исключение может инициироваться в remove(), и любой нормальный клиент будет проверять его при вызове метода remove().

В: Как метод remove() работает с несколькими программными потоками, которые могут использовать разные итераторы для одной коллекции объектов?

О: Поведение remove() для коллекций, изменяющихся в процессе перебора, не определено. Будьте внимательны при программировании параллельного доступа к коллекции в многопоточной модели.

Интеграция с java.util.Iterator

Начнем с класса PancakeHouseMenu; перевести его на java.util.Iterator будет совсем несложно. Достаточно удалить класс PancakeHouseMenuIterator, добавить директиву import java.util.Iterator в начало PancakeHouseMenu и изменить одну строку в PancakeHouseMenu:

```
public Iterator<MenuItem> createIterator() {
    return menuItems.iterator();
}
```

Вместо создания собственного итератора мы просто вызываем метод iterator() для объекта menuItems.

На этом доработка PancakeHouseMenu завершена.

Теперь необходимо внести изменения, позволяющие DinerMenu работать с java.util.Iterator.

```
import java.util.Iterator;

public class DinerMenuIterator implements Iterator {
    MenuItem[] list;
    int position = 0;

    public DinerMenuIterator(MenuItem[] list) {
        this.list = list;
    }

    public MenuItem next() {
        // реализация
    }

    public boolean hasNext() {
        // реализация
    }

    public void remove() {
        if (position <= 0) {
            throw new IllegalStateException
                ("You can't remove an item until you've done at least one next()");
        }
        if (list[position-1] != null) {
            for (int i = position-1; i < (list.length-1); i++) {
                list[i] = list[i+1];
            }
            list[list.length-1] = null;
        }
    }
}
```

Сначала импортируем java.util.Iterator — интерфейс, который мы собираемся реализовать.

Текущая реализация вообще не изменяется...

...но мы должны реализовать remove(). Так как в данном случае используется массив фиксированного размера, при вызове remove() элементы просто сдвигаются на одну позицию.

Работа почти завершена...

Осталось только дать классам Menu общий интерфейс и немного переработать код официантки. Интерфейс Menu очень прост: возможно, когда-нибудь мы дополним его новыми методами (скажем, addItem), но пока этот метод не будет включен в открытый интерфейс:

```
public interface Menu {
    public Iterator<MenuItem> createIterator();
}
```

Простой интерфейс с единственным методом, который возвращает клиентам итератор для элементов меню.

Теперь мы должны добавить директиву `implements Menu` в определения классов PancakeHouseMenu и DinerMenu, а также обновить код Waitress:

Класс Waitress тоже использует java.util.Iterator.

```
import java.util.Iterator;

public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;

    public Waitress(Menu pancakeHouseMenu, Menu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void printMenu() {
        Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem)iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }

    // другие методы
}
```

Конкретные классы Menu заменяются интерфейсом Menu.

Ничего не меняется.

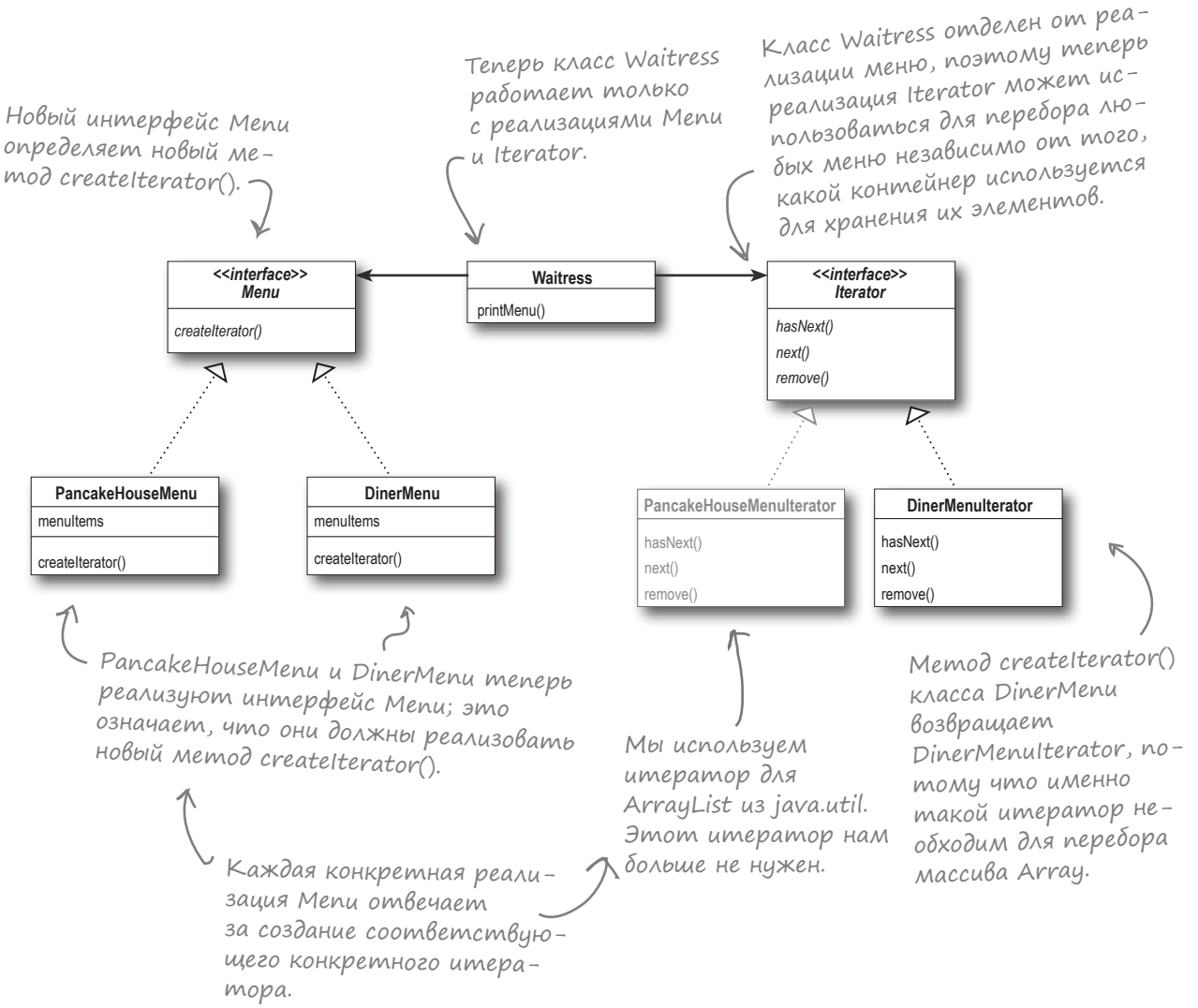
Что нам это дает?

Классы PancakeHouseMenu и DinerMenu реализуют интерфейс Menu. Класс Waitress может обращаться к объекту меню как к реализации интерфейса, а не как к экземпляру конкретного класса. Таким образом, мы сокращаем зависимости между Waitress и конкретными классами — «программируем на уровне интерфейса, а не реализации».

Решение проблемы зависимости Waitress от конкретных классов Menu.

Новый интерфейс Menu состоит из единственного метода createIterator(), реализуемого классами PancakeHouseMenu и DinerMenu. Каждый класс несет ответственность за создание конкретного итератора, соответствующего внутренней реализации коллекции.

Решение проблемы зависимости Waitress от реализации MenuItems.



Определение паттерна Итератор

Вы уже видели, как паттерн Итератор реализуется в самостоятельно написанных итераторах. Также было показано, как итераторы поддерживаются в некоторых классах коллекций языка Java (таких, как ArrayList). Пора ознакомиться с формальным определением паттерна:

> **Паттерн Итератор** предоставляет механизм последовательного перебора элементов коллекции без раскрытия ее внутреннего представления.

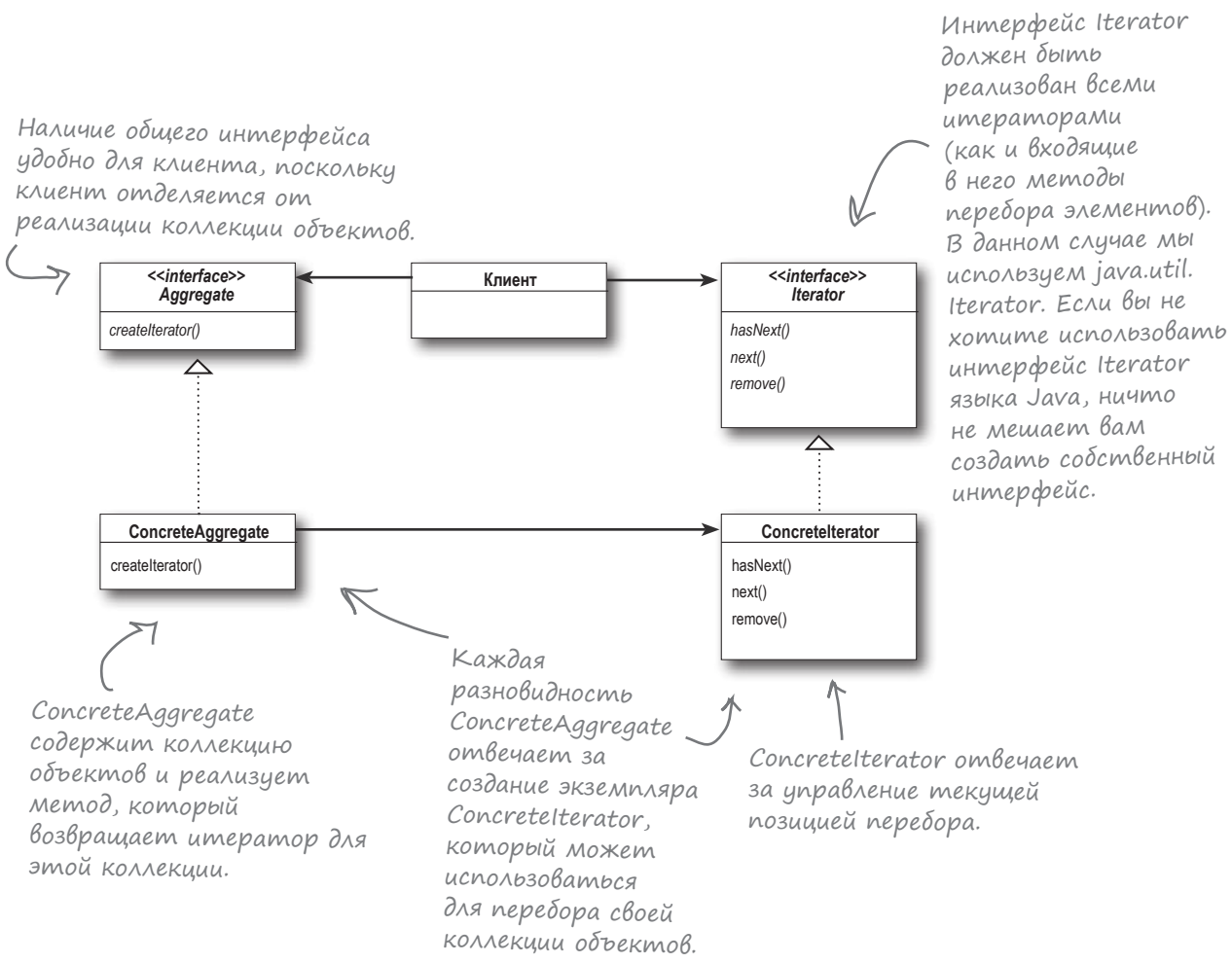
Итак, паттерн позволяет перебирать элементы коллекции, не зная, как реализована коллекция. Мы уже рассмотрели пример с двумя реализациями меню. Однако применение итераторов в ваших собственных архитектурах приводит и к другим, не менее важным последствиям: при наличии универсального механизма перебора элементов можно написать полиморфный код, который работает с *любыми* коллекциями — как метод printMenu(), который работает с элементами, хранящимися в Array, ArrayList или в любой другой коллекции, способной создать Iterator.

Применение паттерна Итератор имеет и другое важное последствие для архитектуры системы: ответственность за перебор элементов передается от объекта коллекции объекту итератора. Это обстоятельство не только упрощает интерфейс и реализацию коллекции, но и избавляет коллекцию от посторонних обязанностей (ее главной задачей является управление объектами, а не перебор).

Следующая диаграмма классов поможет лучше понять суть итератора...

Паттерн Итератор обеспечивает перебор элементов коллекции без раскрытия реализации.

Кроме того, перебор элементов выполняется объектом итератора, а не самой коллекцией. Это упрощает интерфейс и реализацию коллекции, а также способствует более логичному распределению обязанностей.



МОЗГОВОЙ ШТУРМ

Диаграмма классов паттерна Итератор очень похожа на диаграмму классов другого паттерна, описанного ранее. Что это за паттерн? Подсказка: создаваемый экземпляр выбирается субклассом.

Часто Задаваемые Вопросы

В: В других книгах я видел классы итераторов с методами first(), next(), isDone() и currentItem(). Почему здесь используются другие методы?

О: Это «классические» имена методов. Со временем они изменились, и теперь в java.util.Iterator входят методы next(), hasNext() и даже remove().

Классические методы next() и currentItem() в java.util были объединены. Метод isDone() превратился в hasNext(), а у метода first() нет прямого аналога. Дело в том, что в Java новый итератор обычно запрашивается перед началом перебора. Впрочем, принципиальных различий между этими интерфейсами нет, а вы можете наделить свои итераторы новыми аспектами поведения (примером такого расширения служит метод remove() в java.util.Iterator).

В: Говорят, итераторы бывают «внутренними» и «внешними». Что это такое? И какую разновидность мы реализовали в своем примере?

О: Мы реализовали *внешний* итератор — клиент управляет перебором, вызывая метод next() для перехода к следующему элементу. Перебором элементов при использовании внутреннего итератора управляет сам итератор. В этом случае вы должны указать, что ему делать с текущим элементом в ходе перебора (передавая итератору ссылку на выполняемую операцию). Внутренние итераторы уступают внешним в гибкости, поскольку клиент не управляет перебором. С другой стороны, кто-то сочтет, что ими проще пользоваться.

В: Можно ли реализовать итератор, который перебирает элементы не только в прямом, но и в обратном направлении?

О: Конечно. В этом случае итератор обычно дополняется двумя новыми методами: для перехода к предыдущему элементу и для проверки достижения начала коллекции. Библиотека Java Collection Framework предоставляет другой тип интерфейса итераторов, который называется ListIterator. В нем стандартный интерфейс Iterator дополняется методом previous() и еще несколькими методами. Итератор поддерживается всеми коллекциями, реализующими интерфейс List.

В: Кто определяет порядок перебора в неупорядоченных коллекциях — таких, как Hashtable?

О: Итератор не устанавливает определенного порядка перебора. Сама коллекция может быть неупорядоченной; она даже может содержать дубликаты. Таким образом, порядок перебора определяется свойствами коллекции и реализацией. В общем случае не следует делать никаких допущений относительно порядка перебора, если только в описании коллекции явно не указано обратное.

В: Вы упомянули о возможности написания «полиморфного кода» с использованием итератора; можно объяснить подробнее?

О: При написании метода, которому в параметре передается реализация Iterator, мы применяем *полиморфный перебор*. Иначе говоря, такой код может перебирать элементы любой коллекции, если только она поддерживает Iterator. При этом нас не интересует, как реализована коллекция — мы все равно можем написать код для перебора ее элементов.

В: Если я работаю на Java, вероятно, мне стоит всегда использовать интерфейс java.util.Iterator, чтобы я мог использовать свои реализации итераторов с классами, поддерживающими итераторы Java?

О: Вероятно. Общий интерфейс Iterator безусловно упростит использование ваших коллекций с коллекциями Java (такими, как ArrayList и Vector).

В: Я видел в Java интерфейс Enumeration, он тоже реализует паттерн Iterator?

О: Мы уже говорили на эту тему в главе, посвященной паттерну Адаптер. Помните? Java.util.Enumeration — старая реализация Iterator, которая была заменена java.util.Iterator. Интерфейс Enumeration содержит два метода: hasMoreElements() (соответствует hasNext()) и nextElement() (соответствует next()). Вероятно, вам стоит использовать интерфейс Iterator вместо Enumeration, так как он поддерживается большим количеством классов Java.

Принцип одной обязанности

А если все-таки разрешить классам коллекций реализовать как управление объектами, так и методы перебора? Да, это приведет к увеличению количества методов коллекции, ну и что? Чем это плохо?

Чтобы понять, чем это плохо, необходимо сначала осознать один факт: поручая классу не только его непосредственную задачу (управление коллекцией объектов), но и дополнительные задачи (перебор), мы создаем две возможные причины для изменения. Теперь измениться может как внутренняя реализация коллекции, так и механизм перебора. Как видите, наш старый знакомый — ИЗМЕНЕНИЕ — снова оказывается в центре очередного принципа проектирования:

Принцип проектирования

Класс должен иметь только одну причину для изменения.

Мы знаем, что изменений в классах следует по возможности избегать — модификация кода обычно сопровождается массой проблем. Наличие двух причин для изменения повышает вероятность того, что класс изменится в будущем, а если это все же произойдет — изменения повлияют на два аспекта архитектуры.

Что делать? Принцип указывает на то, что каждому классу должна быть выделена одна — и только одна! — обязанность.

Как это часто бывает, в реальной жизни все несколько сложнее: разделение обязанностей в архитектуре является одной из самых сложных задач. Наш мозг склонен объединять аспекты поведения даже в том случае, если в действительности речь идет о двух разных обязанностях. Единственный путь к ее успешному решению — анализ архитектуры и отслеживание возможных причин изменения классов в ходе роста системы.

Каждая обязанность класса является областью потенциальных изменений. Несколько обязанностей — несколько причин для изменения.

Принцип рекомендует ограничить каждый класс одной обязанностью.

Связность — термин, часто используемый для оценки логического единства функций класса или модуля.

Мы говорим, что модуль или класс обладает *высокой связностью*, если он спроектирован для выполнения группы взаимосвязанных функций. Классы с *низкой связностью* проектируются на основе набора разрозненных функций.

Концепция связности является более общей, чем принцип одной обязанности, но эти два понятия тесно связаны. Классы, соответствующие принципу, обычно обладают высокой связностью, и более просты в сопровождении, чем классы с многими обязанностями и низкой связностью.

МОЗГОВОЙ ШТУРМ

Проанализируйте следующие классы и определите, какие из них обладают множественными обязанностями.

Game
- login()
- signup()
- move()
- fire()
- rest()

Person
- setName()
- setAddress()
- setPhoneNumber()
- save()
- load()

Phone
- dial()
- hangUp()
- talk()
- sendData()
- flash()

GumballMachine
- getCount()
- getState()
- getLocation()

DeckOfCards
- hasNext()
- next()
- remove()
- addCard()
- removeCard()
- shuffle()

ShoppingCart
- add()
- remove()
- checkOut()
- saveForLater()

Iterator
- *hasNext()*
- *next()*
- *remove()*

ОСТОРОЖНО, ОПАСНАЯ ЗОНА! БЕРЕГИТЕСЬ НЕОБОСНОВАННЫХ ДОПУЩЕНИЙ!

МОЗГОВОЙ ШТУРМ 2

Определите, какой связностью — высокой или низкой — обладают следующие классы.

Game
- login()
- signup()
- move()
- fire()
- rest()
- getHighScore()
- getName()

GameSession
- login()
- signup()

PlayerActions
- move()
- fire()
- rest()

Player
- getHighScore()
- getName()

Хорошо, что вы занялись паттерном Итератор, потому что в бюро поглощений и слияний Объективиля состоялась очередная сделка... Мы объединяемся с кафе и будем поддерживать его меню.

А мы-то думали, что проблем и так хватает. Что будем делать?

Не вешай нос. Я уверен, что мы сможем адаптировать их меню для паттерна Итератор.

Знакомство с классом CafeMenu

Перед вами код меню кафе. Вроде бы его интеграция в нашу инфраструктуру не вызовет особых проблем... Давайте проверим.

Класс CafeMenu не реализует наш интерфейс Menu, но эта проблема легко решается.

Меню кафе хранится в коллекции HashMap. Поддерживает ли эта коллекция Iterator? Скоро узнаем...

```
public class CafeMenu {
    HashMap<String, MenuItem> menuItems = new HashMap<String, MenuItem>();

    public CafeMenu() {
        addItem("Veggie Burger and Air Fries",
            "Veggie burger on a whole wheat bun, lettuce, tomato, and fries",
            true, 3.99);
        addItem("Soup of the day",
            "A cup of the soup of the day, with a side salad",
            false, 3.69);
        addItem("Burrito",
            "A large burrito, with whole pinto beans, salsa, guacamole",
            true, 4.29);
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.put(menuItem.getName(), menuItem);
    }

    public Map<String, MenuItem> getItems() {
        return menuItems;
    }
}
```

Элементы CafeMenu, как и элементы других меню, инициализируются в конструкторе.

Здесь мы создаем новый элемент MenuItem и добавляем его в хеш-таблицу menuItems.

Ключ — название элемента меню.

Значение — объект menuItem.

А этот метод нам не понадобится.

Возьми в руку карандаш

Прежде чем заглядывать на следующую страницу, запишите три основные задачи, которые необходимо решить для интеграции кода в нашу инфраструктуру:

1. _____

2. _____

3. _____

Переработка кода CafeMenu

Класс CafeMenu легко интегрируется в нашу инфраструктуру. Почему? Потому что HashMap принадлежит к числу коллекций Java, поддерживающих Iterator. Но процедура интеграции несколько отличается от ArrayList...

Класс CafeMenu реализует интерфейс Menu, чтобы класс Waitress мог использовать его, как и две другие реализации Menu.

```
public class CafeMenu implements Menu {
    HashMap<String, MenuItem> menuItems = new HashMap<String, MenuItem>();

    public CafeMenu() {
        // constructor code here
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.put(menuItem.getName(), menuItem);
    }

    public Map<String, MenuItem> getItems() {
        return menuItems;
    }

    public Iterator<MenuItem> createIterator() {
        return menuItems.values().iterator();
    }
}
```

Мы используем HashMap – стандартную структуру данных для хранения значений.

Как и прежде, мы избавляемся от getItems(), чтобы не раскрывать реализацию menuItems клиентскому коду Waitress.

Реализация метода createIterator(). Обратите внимание: мы получаем итератор не для всей коллекции HashMap, а только для значений.

Код под увеличительным стеклом

Коллекция HashMap немного сложнее ArrayList, потому что в ней хранятся пары «ключ–значение». Тем не менее мы можем получить итератор для набора значений (относящихся к классу MenuItem).

```
public Iterator<MenuItem> createIterator() {
    return menuItems.values().iterator();
}
```

Сначала получаем набор значений для Hashtable, который представляет собой коллекцию всех объектов в таблице.

К счастью, коллекция поддерживает метод iterator(), возвращающий объект типа java.util.Iterator.

Поддержка CafeMenu в классе Waitress

Пока все было просто; а как насчет изменения класса Waitress для поддержки новой реализации Menu? В коде Waitress, рассчитанном на работу с Iterator, это тоже должно быть просто.

```
public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;
    Menu cafeMenu;

    public Waitress(Menu pancakeHouseMenu, Menu dinerMenu, Menu cafeMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
        this.cafeMenu = cafeMenu;
    }

    public void printMenu() {
        Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
        Iterator<MenuItem> cafeIterator = cafeMenu.createIterator();

        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
        System.out.println("\nDINNER");
        printMenu(cafeIterator);
    }

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
}
```

Меню кафе передается вместе с другими меню в конструкторе. Там оно сохраняется в переменной экземпляра.

Чтобы вывести меню, достаточно создать итератор и передать его printMenu(). И все!

Ничего не меняется.

Вывод полного меню

Давайте обновим тестовую программу и убедимся в том, что все работает правильно.

```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();
        CafeMenu cafeMenu = new CafeMenu();

        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu, cafeMenu);

        waitress.printMenu();
}
```

Создаем CafeMenu...

... и передаем его Waitress.

Теперь тестовая программа выводит содержимое всех трех меню.

Результат тестирования; обратите внимание на новое меню кафе!

```
File  Edit  Window  Help  Kathy&BertLikePancakes
% java DinerMenuTestDrive

MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles, with your choice of blueberries or strawberries

LUNCH
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hotdog, 3.05 -- A hot dog, with saurkraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

DINNER
Soup of the day, 3.69 -- A cup of the soup of the day, with a side salad
Burrito, 4.29 -- A large burrito, with whole pinto beans, salsa, guacamole
Veggie Burger and Air Fries, 3.99 -- Veggie burger on a whole wheat bun,
 lettuce, tomato, and fries
%
```

Сначала перебираем меню блинной.

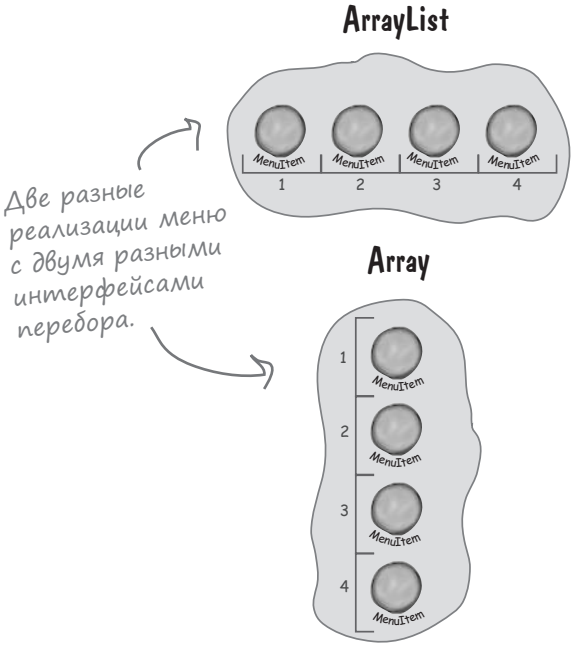
Затем меню бистро.

И наконец новое меню кафе — все это в одном коде перебора.

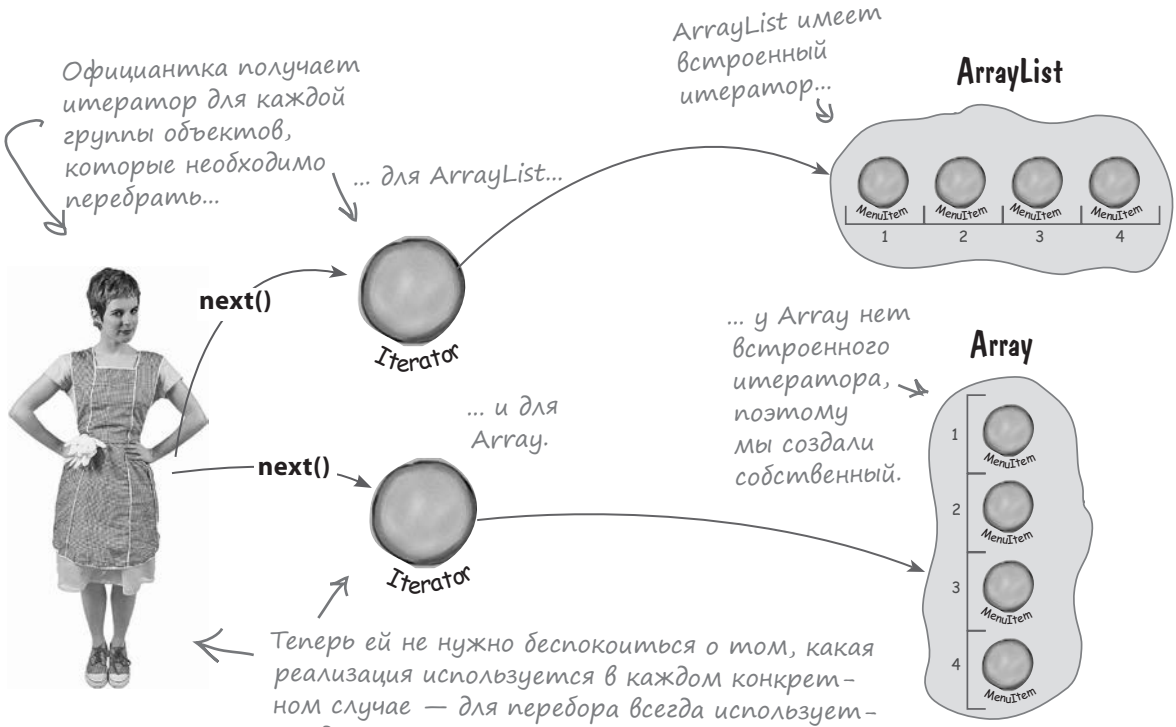
Что мы сделали?

Мы хотели, чтобы официантка могла легко перебирать элементы меню...

... ничего не зная о реализации самих меню.

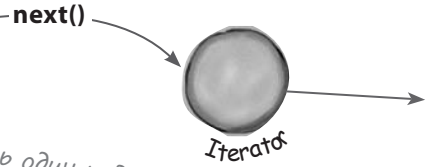


Мы отделили официантку от реализации...



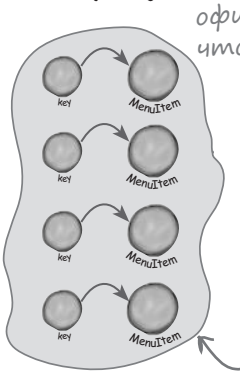
... и упростили дальнейшие расширения

Работа с итератором отделяет официантку от реализации меню, поэтому при необходимости мы легко можем добавить новые меню.



Теперь один код может использоваться для перебора любых групп объектов, а подробности внутренней реализации остаются скрытыми от клиентского кода.

Мы без труда добавляем новую реализацию меню; а благодаря наличию итератора официантка знает, что с ней делать.



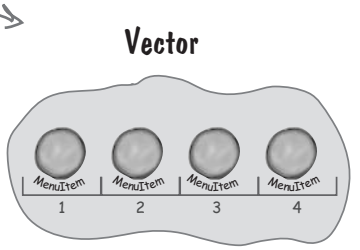
Создать реализацию Iterator для значений Hashtable было несложно; нужный итератор возвращается вызовом values.iterator().

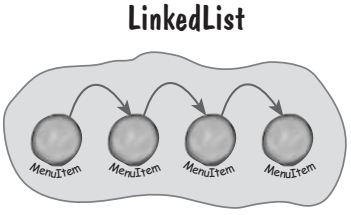
Но это еще не все!

Java предоставляет в ваше распоряжение многочисленные классы коллекций для хранения и выборки групп объектов (такие, как Vector и LinkedList).

В основном их интерфейсы различаются.

Но почти все они предоставляют возможность получения Iterator.

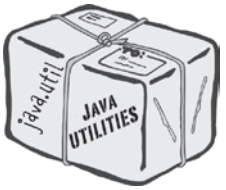




...и другие!

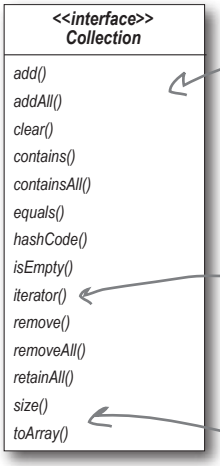
Но даже если Iterator и не поддерживается, это тоже нормально, потому что теперь вы знаете, как построить собственную реализацию.

Итераторы и коллекции



В своем примере мы использовали пару классов из библиотеки Java Collections Framework. Эта «библиотека» в действительности представляет собой обычный набор классов и интерфейсов, в числе которых использованный нами класс ArrayList и многие другие (Vector, LinkedList, Stack, PriorityQueue и т. д.). Каждый из этих классов реализует интерфейс java.util.Collection, содержащий полезные методы для работы с группами объектов.

Давайте познакомимся с этим интерфейсом хотя бы в общих чертах:

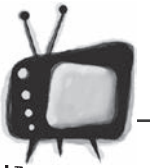


Как видите, здесь есть немало полезного. Вы можете добавлять и удалять элементы из коллекции, даже не зная, как она реализована.

Наш старый знакомый — метод iterator(). С его помощью можно получить итератор для любого класса, реализующего интерфейс Collection.

Также полезны методы size() (получение количества элементов) и toArray() (преобразование коллекции в массив).

Каждый объект Collection знает, как создать свой итератор. Вызов iterator() для ArrayList возвращает конкретный итератор, предназначенный для ArrayList, но при этом используемый им конкретный класс остается скрытым; перебор коллекции осуществляется через интерфейс Iterator.



Будьте осторожны!

Hashtable — один из классов с к*освенной* поддержкой итераторов. Как было показано при реализации CafeMenu, для такого класса можно получить Iterator, но только после предварительной выборки его субколлекции values. И это вполне логично; в Hashtable хранятся два вида объектов, ключи и значения. Чтобы перебрать значения, необходимо сначала извлечь их из Hashtable и только потом получить итератор.

Магниты с кодами

Повара решили, что содержимое их меню должно чередоваться. Другими словами, в понедельник, среду, пятницу и воскресенье будут предлагаться одни блюда, а во вторник, четверг и субботу — другие. Код нового «чередующего» итератора для DinerMenu уже был написан, но кто-то для шутки разрезал его на куски и разместил на холодильнике. Сможете ли вы собрать его заново? Некоторые фигурные скобки упали на пол. Они слишком малы, чтобы их подбирать — добавьте столько скобок, сколько считаете нужным!

```
MenuItem menuItem = items[position];
position = position + 2;
return menuItem;
```

```
import java.util.Iterator;
import java.util.Calendar;
```

```
public Object next() {
```

```
{
```

```
public AlternatingDinerMenuIterator(MenuItem[] items)
```

```
this.items = items;
position = Calendar.DAY_OF_WEEK % 2;
```

```
public void remove() {
```

```
implements Iterator<MenuItem>
```

```
MenuItem[] items;
int position;
```

```
}
```

```
public class AlternatingDinerMenuIterator
```

```
public boolean hasNext() {
```

```
throw new UnsupportedOperationException(
    "Alternating Diner Menu Iterator does not support remove()");
```

```
if (position >= items.length || items[position] == null) {
    return false;
} else {
    return true;
}
```

```
}
```

Официантка, на выход!

Код Waitress был значительно усовершенствован, но следует признать, что три вызова printMenu() выглядят довольно уродливо.

Давайте откровенно признаем: каждый раз, когда в системе будет появляться новое меню, нам придется открывать код Waitress и добавлять новый фрагмент. Пожалуй, это является нарушением принципа открытости/закрытости.

Три вызова createIterator().

```
public void printMenu() {
    Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
    Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
    Iterator<MenuItem> cafeIterator = cafeMenu.createIterator();

    System.out.println("MENU\n----\nBREAKFAST");
    printMenu(pancakeIterator);

    System.out.println("\nLUNCH");
    printMenu(dinerIterator);

    System.out.println("\nDINNER");
    printMenu(cafeIterator);
}
```

Три вызова printMenu.

Каждый раз, когда в системе добавляется или удаляется меню, этот код открывается для изменений.

Мы отлично справились с логической изоляцией реализации меню и инкапсуляцией перебора в итераторе. Однако мы продолжаем работать с меню как с отдельными независимыми объектами. Хорошо бы найти способ работать с ними как с единым целым.

МОЗГОВОЙ ШТУРМ

Классу Waitress по-прежнему приходится трижды вызывать printMenu(), по одному разу для каждого меню. Можете ли вы придумать способ объединения меню, чтобы было достаточно одного вызова? Или чтобы для перебора всех меню классу Waitress передавался всего один итератор?

Похоже, идея удачная. Давайте попробуем:

```
public class Waitress {
    ArrayList<Menu> menus;

    public Waitress(ArrayList<Menu> menus) {
        this.menus = menus;
    }

    public void printMenu() {
        Iterator<Menu> menuIterator = menus.iterator();
        while(menuIterator.hasNext()) {
            Menu menu = menuIterator.next();
            printMenu(menu.createIterator());
        }
    }

    void printMenu(Iterator<Menu> iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
}
```

Передаем коллекцию ArrayList с элементами Menu.

Перебираем объекты меню, передавая итератор каждого объекта перегруженному методу printMenu().

Здесь код не изменяется.

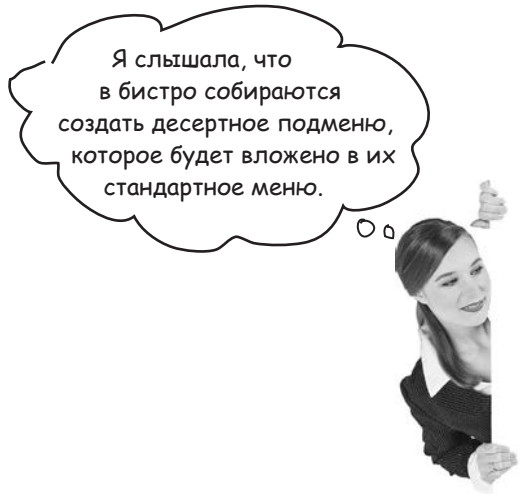
Выглядит неплохо. Правда, мы потеряли названия меню, но их можно добавить в объекты меню.

А когда мы уже торжествовали победу...

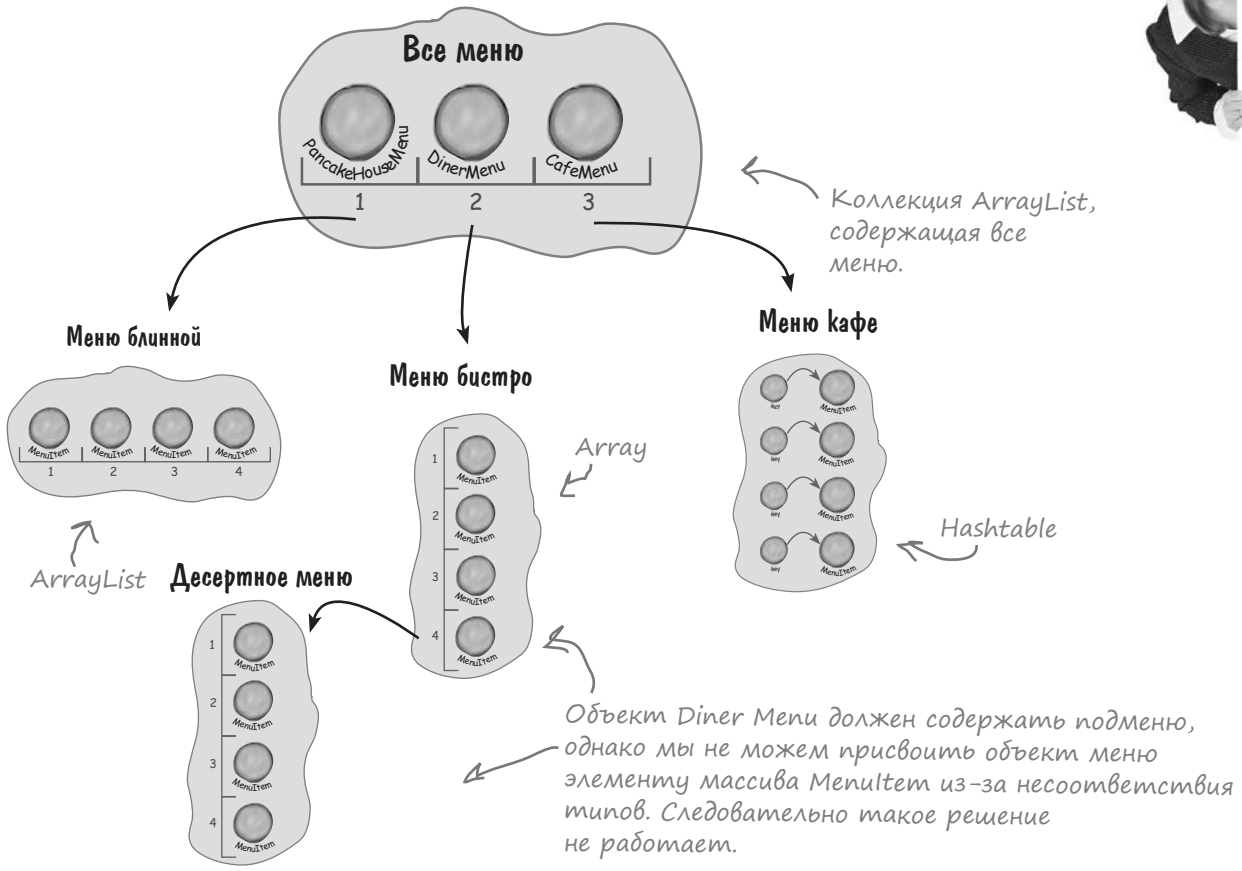
Было решено добавить десертное подменю.

Что делать? Теперь мы должны поддерживать не только несколько меню, но и меню внутри других меню.

Было бы хорошо, если бы десертное меню можно было сделать элементом коллекции DinerMenu, но в текущей реализации такое решение работать не будет.



Чего мы хотим (примерно):



Ничего не выйдет!

Коллекцию десертного меню не удастся присвоить объекту MenuItem.

Придется вносить изменения!

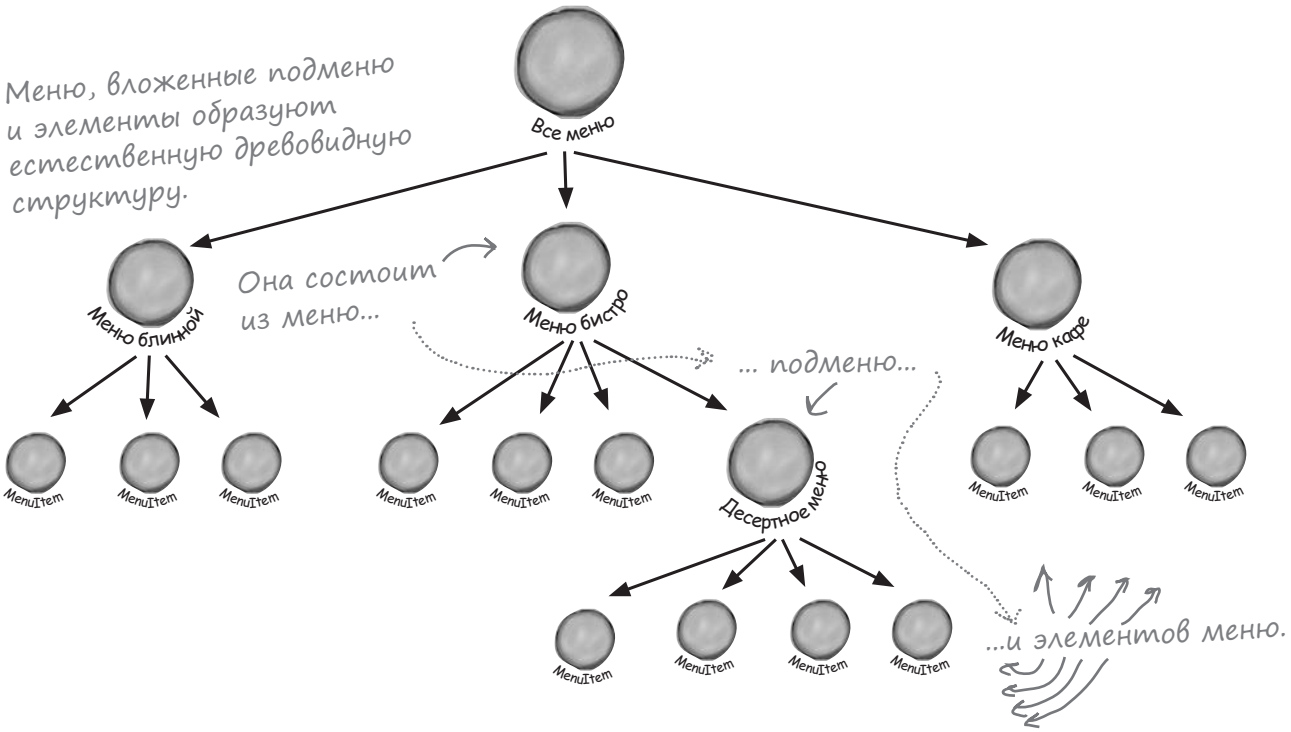
Что нам нужно?

Пора принять ответственное решение и переработать реализацию, чтобы она стала достаточно общей для работы с любыми меню (а теперь и подменю). Да, вы правильно поняли — поварам придется изменить реализацию своих меню.

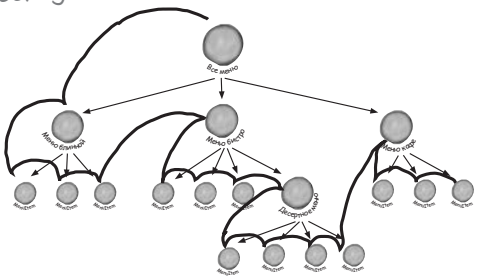
Дело в том, что наша архитектура достигла критического уровня сложности. Если мы не переработаем ее сейчас, она уже не сможет адаптироваться к дальнейшим объединениям или появлению подменю.

Итак, каким требованиям должна удовлетворять новая архитектура?

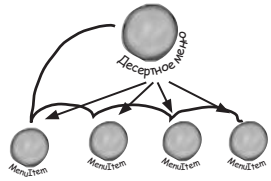
- Древовидная структура для поддержки меню, подменю и элементов.
- Механизм перебора элементов в каждом меню, по крайней мере не менее удобный, чем с использованием итераторов.
- Более гибкие средства перебора элементов, чтобы, например, мы могли перебрать только элементы десертного подменю или же все меню бистро вместе с десертным подменю.



Как и прежде, нам понадобится механизм перебора всех узлов дерева.



А также более гибкие средства перебора — например, по элементам одного меню.



МОЗГОВОЙ ШТУРМ

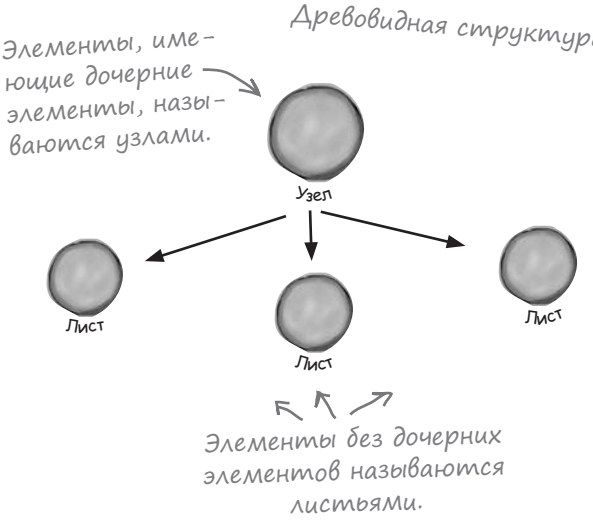
А как бы вы подошли к реализации новых требований к архитектуре? Подумайте, прежде чем перевернуть страницу.

Определение паттерна Компоновщик

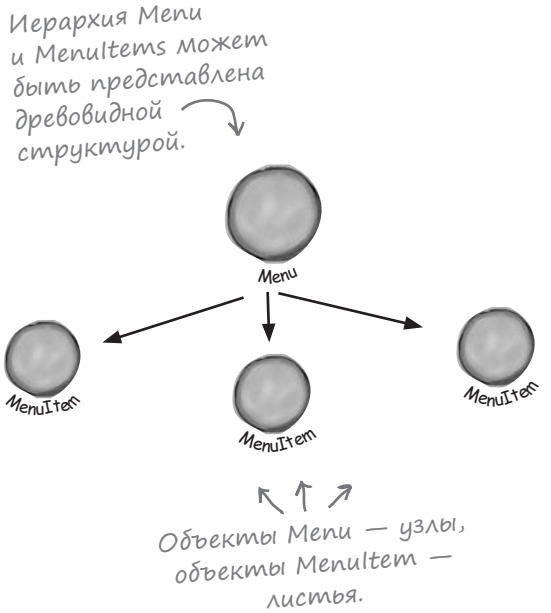
Да, вы не ошиблись, мы собираемся привлечь очередной паттерн для решения этой проблемы. Мы еще не закончили с паттерном Итератор — он остается частью нашего решения. Тем не менее проблема управления меню вышла на новый уровень, недоступный для паттерна Итератор. Итак, мы сделаем шаг назад и решим ее при помощи паттерна Компоновщик.

Не будем ходить вокруг да около — начнем прямо с формального определения:

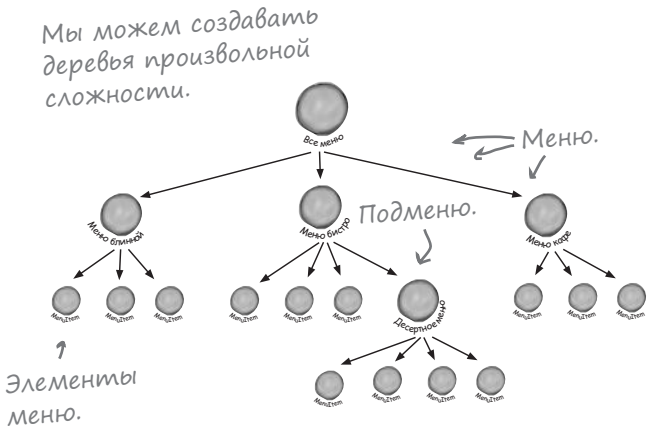
> **Паттерн Компоновщик** объединяет объекты в древовидные структуры для представления иерархий «часть/целое». Компоновщик позволяет клиенту выполнять однородные операции с отдельными объектами и их совокупностями.

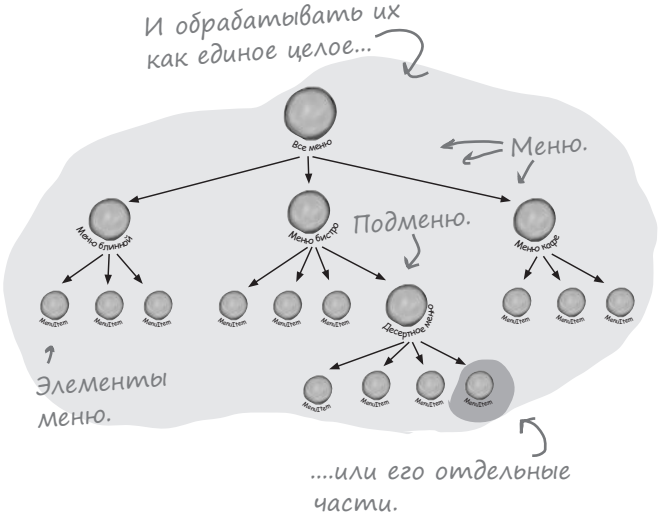


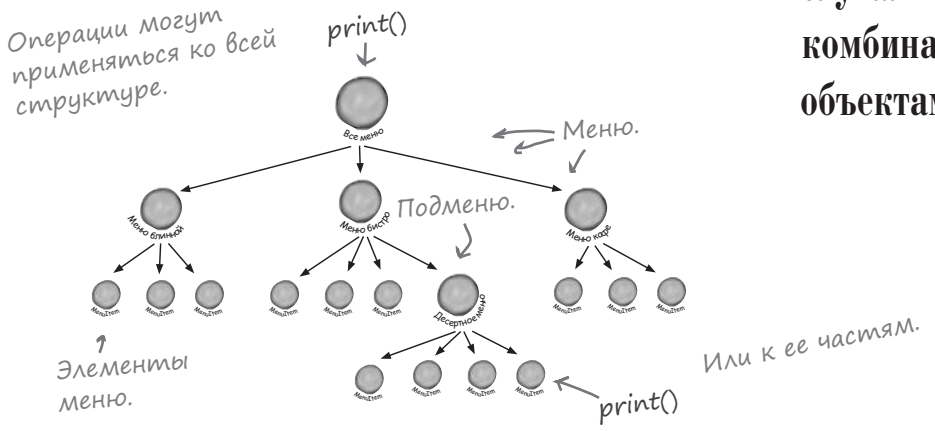
Рассмотрим это определение в контексте наших меню: паттерн дает возможность создать древовидную структуру, которая может работать с вложенными группами меню ***и*** элементами меню. Размещая меню и элементы в одной структуре, мы создаем иерархию «часть/целое». Иначе говоря, дерево объектов, которое состоит из отдельных частей (меню и элементы меню), но при этом может рассматриваться как единое целое (одно большое «суперменю»).



Построив «суперменю», мы можем использовать этот паттерн для того, чтобы «выполнять однородные операции с отдельными объектами и их комбинациями». Что это означает? Если у вас есть древовидная структура из меню, подменю (и, возможно, под-подменю) с элементами, любое меню представляет собой «комбинацию», так как оно может содержать другие меню и команды меню. Отдельными объектами являются только элементы меню — они не могут содержать других объектов. Применение в архитектуре паттерна Компоновщик позволит нам написать простой код, который применяет одну и ту же операцию (например, вывод!) ко всей структуре меню.

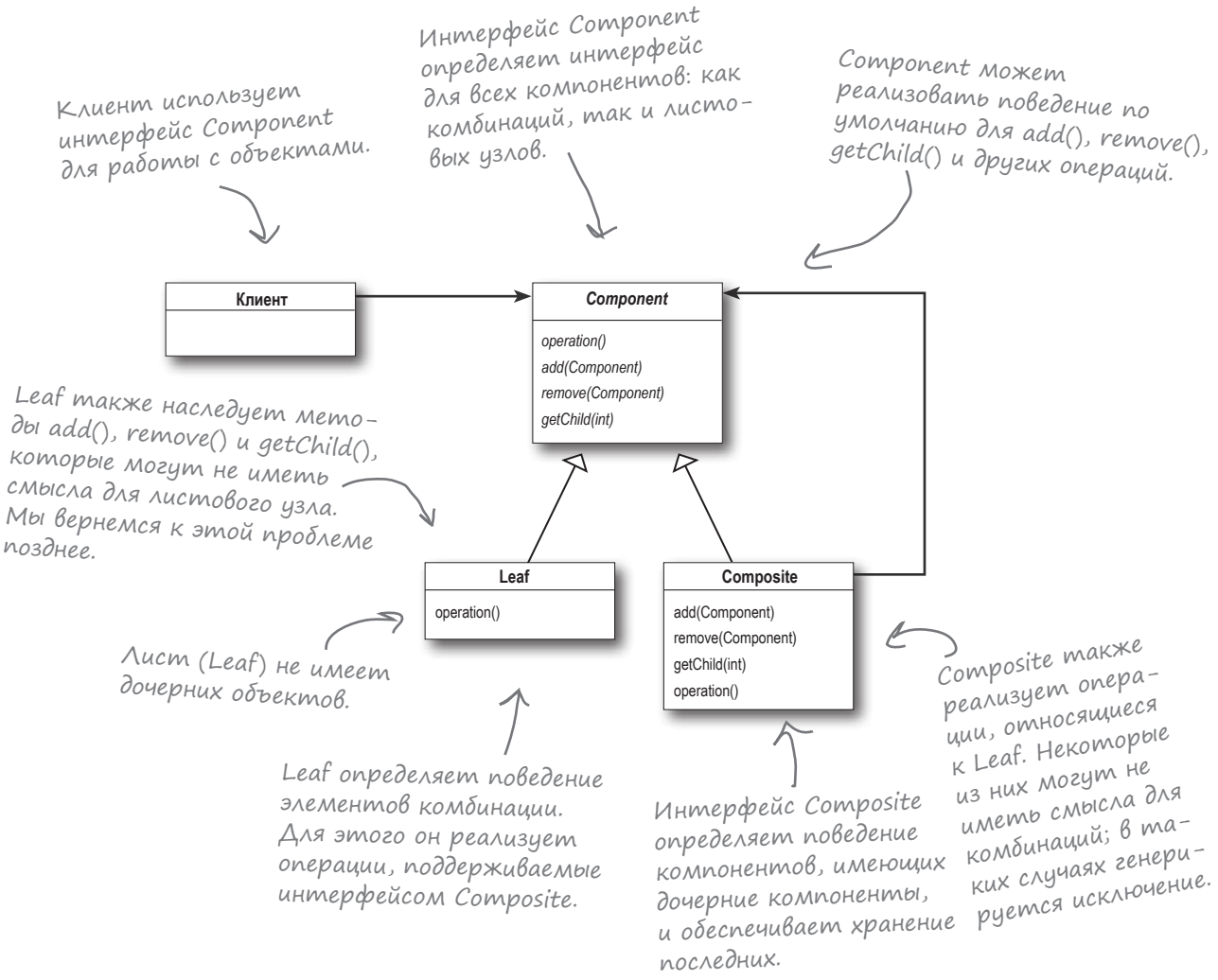






Паттерн Компоновщик позволяет создавать древовидные структуры, узлами которых являются как комбинации, так и отдельные объекты.

В такой структуре одни и те же операции могут применяться и к комбинациям, и к отдельным объектам. Иначе говоря, во многих случаях различия между комбинациями и отдельными объектами игнорируются.



Часто Задаваемые Вопросы

В: Компоненты, комбинации, деревья? Я уже запутался.

О: Комбинация состоит из компонентов. Компоненты бывают двух видов: комбинации и листовые элементы. Заметили рекурсию? Комбинация содержит дочерние компоненты, которые могут быть другими комбинациями или листьями.

При такой организации данных образуется древовидная структура (инвертированное дерево), корнем которой является комбинация, а ветвями — комбинации, завершаемые листовыми узлами.

В: А при чем здесь итераторы?

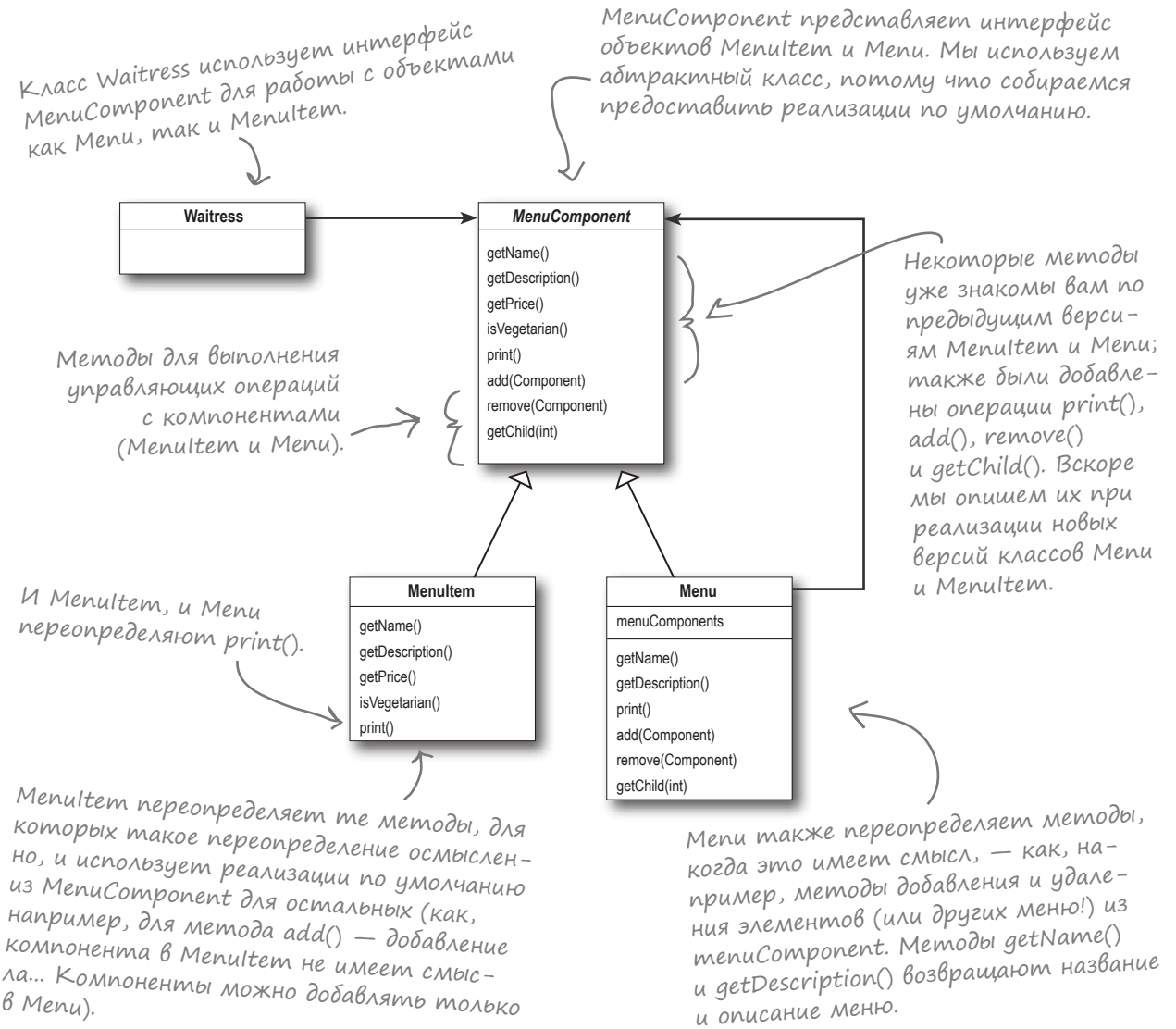
О: Напомню, что мы заново реализуем систему меню на базе нового решения: паттерна Компоновщик. Так что не ждите, что итератор, как по волшебству, превратится в комбинацию. Тем не менее эти два паттерна хорошо работают в сочетании друг с другом.

Вскоре мы рассмотрим пару возможных применений итераторов в реализации комбинаций.

Проектирование меню с использованием паттерна Компоновщик

Как же нам применить паттерн Компоновщик при проектировании системы меню? Для начала необходимо определить интерфейс компонента; этот интерфейс, общий для меню и элементов меню, позволяет выполнять с ними однородные операции. Другими словами, *один и тот же* метод вызывается как для меню, так и для их элементов.

Возможно, вызов некоторых методов для меню или элементов меню ***не имеет смысла***, но с этим мы разберемся позднее (притом совсем скоро). А пока в общих чертах посмотрим, как система меню укладывается в структуру паттерна Компоновщик:



Реализация MenuComponent

Начнем с абстрактного класса MenuComponent; напомним, что его задачей является определение интерфейса листьев и комбинационных узлов. Логично спросить: «Разве MenuComponent в этом случае не играет сразу две роли?» Да, возможно, и мы еще вернемся к этому вопросу. А пока мы определим реализацию по умолчанию для некоторых методов; если MenuItem (лист) или Menu (комбинация) не хотят реализовывать такие методы (например, getChild() для листового узла), они смогут воспользоваться стандартным поведением:

Все компоненты должны реализовать интерфейс MenuComponent; но так как листьям и узлам отводятся разные роли, не всегда можно определить реализацию по умолчанию для каждого метода, для которого она имеет смысл. Иногда лучшее, что можно сделать в реализации по умолчанию, — инициировать исключение времени выполнения.

MenuComponent предоставляет реализации по умолчанию для всех методов.

```
public abstract class MenuComponent {

    public void add(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public void remove(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public MenuComponent getChild(int i) {
        throw new UnsupportedOperationException();
    }

    public String getName() {
        throw new UnsupportedOperationException();
    }
    public String getDescription() {
        throw new UnsupportedOperationException();
    }
    public double getPrice() {
        throw new UnsupportedOperationException();
    }
    public boolean isVegetarian() {
        throw new UnsupportedOperationException();
    }

    public void print() {
        throw new UnsupportedOperationException();
    }
}
```

Так как одни методы имеют смысл только для MenuItem, а другие — только для Menu, реализация по умолчанию инициирует UnsupportedOperationException. Если объект MenuItem или Menu не поддерживает операцию, ему не нужно ничего делать — он просто наследует реализацию по умолчанию.

Группа «комбинационных» методов — то есть методов для добавления, удаления и получения MenuComponent.

Группа «методов операций», используемых MenuItem. Как вы вскоре увидите при анализе кода Menu, некоторые из этих методов также могут использоваться в Menu.

Метод print() реализуется как в Menu, так и в MenuItem, но мы предоставляем реализацию по умолчанию.

Реализация MenuItem

Ну что ж, займемся классом MenuItem. Напомним, что это класс листового узла на диаграмме классов паттерна Компоновщик, и он реализует поведение элементов комбинации.


```
public class MenuItem extends MenuComponent {
    String name;
    String description;
    boolean vegetarian;
    double price;

    public MenuItem(String name,
                    String description,
                    boolean vegetarian,
                    double price)
    {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }

    public void print() {
        System.out.print("  " + getName());
        if (isVegetarian()) {
            System.out.print("(v)");
        }
        System.out.println(", " + getPrice());
        System.out.println("     -- " + getDescription());
    }
}
```

Прежде всего класс должен расширять интерфейс MenuComponent.

Конструктор получает название, описание и т. д. и сохраняет ссылки на полученные данные. Он почти не отличается от конструктора в старой реализации элементов меню.

Get-методы не отличаются от предыдущей реализации.

А этот метод отличается от предыдущей реализации. Мы переопределяем метод print() класса MenuComponent. Для MenuItem этот метод выводит полное описание элемента меню: название, описание, цену и признак вегетарианского блюда.

Реализация комбинационного меню

После класса элемента меню MenuItem остается создать класс комбинационного узла, который мы назовем Menu. Напомним, что класс комбинации может содержать элементы MenuItem ***или*** другие Menu. Этот класс не реализует пару методов MenuComponent (getPrice() и isVegetarian()), потому что эти методы не имеют особого смысла для Menu.

Menu расширяет MenuComponent (как и MenuItem).

Menu может иметь любое количество потомков типа MenuComponent; для их хранения будет использоваться внутренняя коллекция ArrayList.

```
public class Menu extends MenuComponent {
    ArrayList<MenuComponent> menuComponents = new ArrayList<MenuComponent>();
    String name;
    String description;

    public Menu(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }

    public void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }

    public MenuComponent getChild(int i) {
        returnmenuComponents.get(i);
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("----------------------");
    }
}
```

Отличается от нашей старой реализации: с каждым меню связывается название и описание.

Включение в Menu объектов MenuItem или других объектов Menu. Так как и MenuItem, и Menus расширяют MenuComponent, хватает одного метода.

Также поддерживаются операции удаления и получения MenuComponent.

Get-методы для получения названия и описания.

Обратите внимание: мы не переопределяем getPrice() или isVegetarian(), потому что эти методы не имеют смысла для Menu. При попытке вызвать их для Menu будет выдано исключение UnsupportedOperationException.

При выводе объекта Menu выводится его название и описание.

Минутку, я не понял реализацию print(). Вроде как к комбинациям должны применяться те же операции, что и к листьям. Применив эту реализацию print() к комбинации, я получу только название и описание меню, а не содержимое КОМБИНАЦИИ.

Отличное замечание! Поскольку меню является комбинацией и содержит как объекты MenuItem, так и другие объекты Menu, метод print() должен выводить все его содержимое. А если он этого не делает, нам придется перебирать всю комбинацию и выводить каждый элемент самостоятельно. Такой подход противоречит самой цели создания комбинационных структур.

Как вы вскоре убедитесь, правильно реализовать print() несложно, потому что мы можем положиться на способность каждого компонента вывести себя. Решение получается элегантным и рекурсивным. Смотрите сами:

Исправление метода print()

```
public class Menu extends MenuComponent {
    ArrayList<MenuComponent> menuComponents = new ArrayList<MenuComponent>();
    String name;
    String description;

    // код конструктора

    // другие методы

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");

        Iterator<MenuComponent> iterator = menuComponents.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                                  iterator.next();
            menuComponent.print();
        }
    }
}
```

Нужно лишь изменить метод print(), чтобы он выводил не только информацию о меню, но и все компоненты: другие меню и элементы меню.

Смотрите! Мы используем итератор для перебора всех компонентов объекта Menu... Ими могут быть другие объекты Menu или объекты MenuItem.

Так как и Menu, и MenuItem реализуют метод print(), мы просто вызываем print(), а все остальное они сделают сами.

ВНИМАНИЕ: если в процессе перебора мы встретим другой объект меню, его метод print() начнет новый перебор, и т. д.

Готовимся к тестированию...

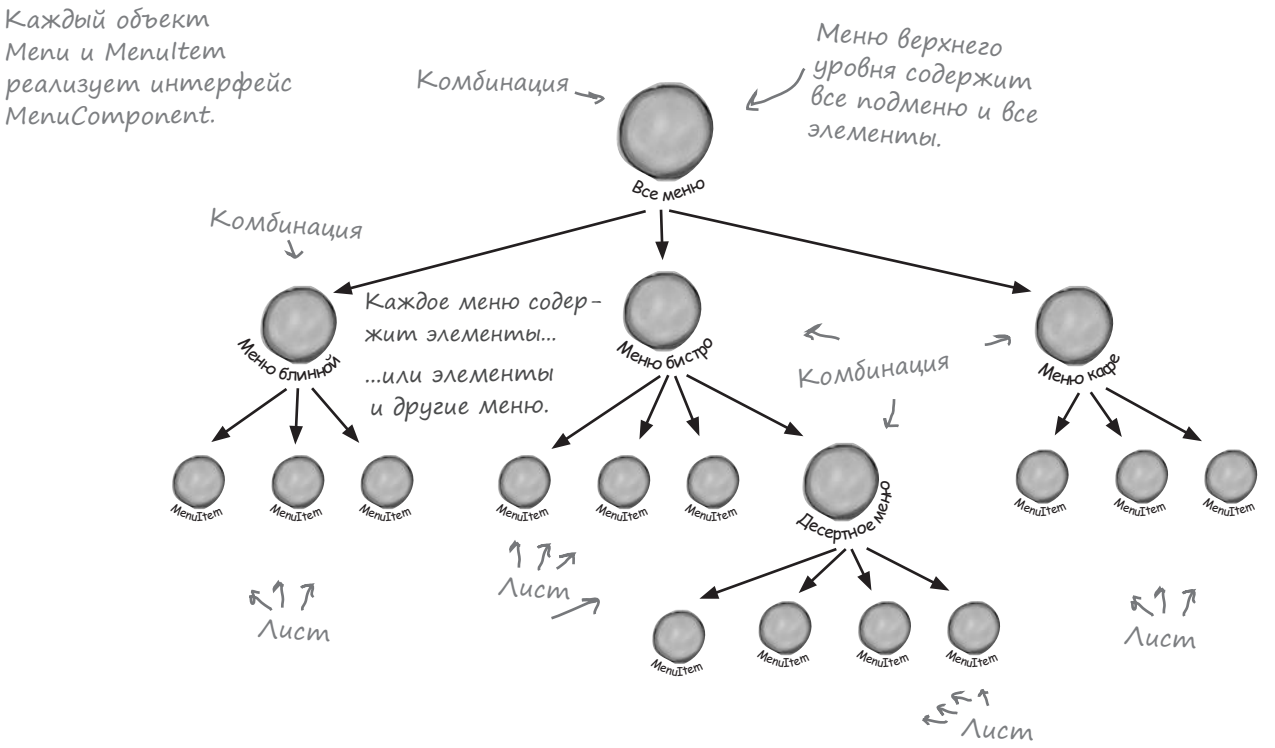
Пора переходить к тестированию, но сначала необходимо обновить класс Waitress — в конце концов, он является основным клиентом нашего кода!

```
public class Waitress {
    MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    public void printMenu() {
        allMenus.print();
    }
}
```

Да! Код Waitress настолько несложен. Мы просто передаем компонент меню верхнего уровня — тот, который содержит остальные меню. Мы назвали его allMenus.

А чтобы вывести всю иерархию меню — все меню и все их элементы — достаточно вызвать метод print() для меню верхнего уровня.

И последнее, о чем стоит сказать до написания тестовой программы. Давайте посмотрим, как будет выглядеть комбинация меню во время выполнения:



Теперь можно и тестировать...

Осталось написать тестовую программу. В отличие от предыдущей версии, все меню будут созданы непосредственно в ней. Мы могли бы потребовать у каждого повара его новое меню, но сначала протестируем все вместе. Код:

```
public class MenuTestDrive {
    public static void main(String args[]) {
        MenuComponent pancakeHouseMenu =
            new Menu("PANCAKE HOUSE MENU", "Breakfast");
        MenuComponent dinerMenu =
            new Menu("DINER MENU", "Lunch");
        MenuComponent cafeMenu =
            new Menu("CAFE MENU", "Dinner");
        MenuComponent dessertMenu =
            new Menu("DESSERT MENU", "Dessert of course!");

        MenuComponent allMenus = new Menu("ALL MENUS", "All menus combined");

        allMenus.add(pancakeHouseMenu);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        // добавление других элементов

        dinerMenu.add(new MenuItem(
            "Pasta",
            "Spaghetti with Marinara Sauce, and a slice of sourdough bread",
            true,
            3.89));

        dinerMenu.add(dessertMenu);

        dessertMenu.add(new MenuItem(
            "Apple Pie",
            "Apple pie with a flakey crust, topped with vanilla icecream",
            true,
            1.59));

        // добавление других элементов

        Waitress waitress = new Waitress(allMenus);

        waitress.printMenu();
    }
}
```

Сначала создаем все объекты меню.

Также нам понадобится меню верхнего уровня — назовем его allMenus.

Для включения каждого меню в allMenus используется комбинационный метод add().

Затем в меню добавляются остальные элементы. Здесь приведен только один пример; за остальными обращайтесь к полному исходному коду.

В меню также включается другое меню. Для dinerMenu важно только то, что все хранящиеся в нем объекты являются MenuComponent.

Создаем элемент в десертном меню...

Сконструированная иерархия меню передается классу Waitress. И как вы уже видели, вывести содержимое меню в этой версии проще простого.

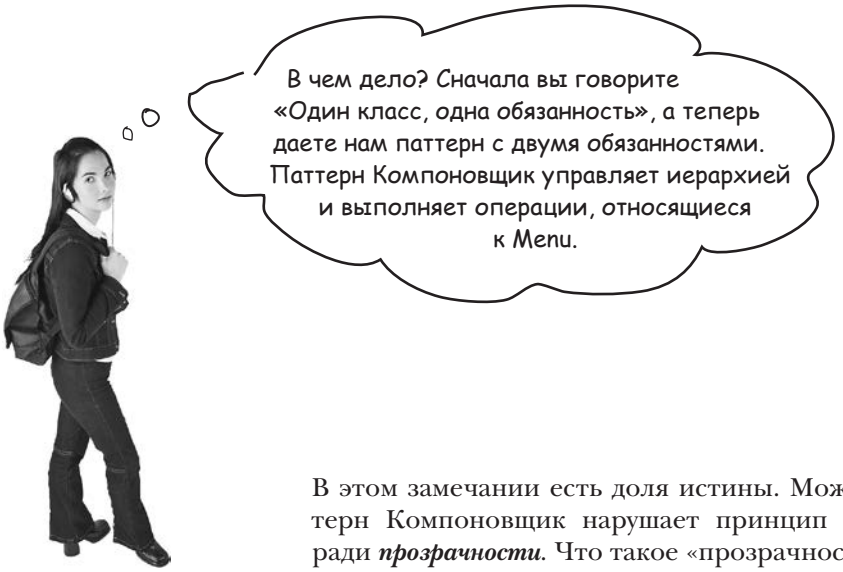
Готовимся к тестовому запуску...

ВНИМАНИЕ: приведены результаты для полной версии исходного кода.

```
File Edit Window Help GreenEggs&Spam
% java MenuTestDrive

ALL MENUS, All menus combined
---------------------

PANCAKE HOUSE MENU, Breakfast
---------------------
  K&B's Pancake Breakfast(v), 2.99
     -- Pancakes with scrambled eggs, and toast
  Regular Pancake Breakfast, 2.99
     -- Pancakes with fried eggs, sausage
  Blueberry Pancakes(v), 3.49
     -- Pancakes made with fresh blueberries, and blueberry syrup
  Waffles(v), 3.59
     -- Waffles, with your choice of blueberries or strawberries

DINER MENU, Lunch
---------------------
  Vegetarian BLT(v), 2.99
     -- (Fakin') Bacon with lettuce & tomato on whole wheat
  BLT, 2.99
     -- Bacon with lettuce & tomato on whole wheat
  Soup of the day, 3.29
     -- A bowl of the soup of the day, with a side of potato salad
  Hotdog, 3.05
     -- A hot dog, with saurkraut, relish, onions, topped with cheese
  Steamed Veggies and Brown Rice(v), 3.99
     -- Steamed vegetables over brown rice
  Pasta(v), 3.89
     -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

DESSERT MENU, Dessert of course!
---------------------
  Apple Pie(v), 1.59
     -- Apple pie with a flakey crust, topped with vanilla icecream
  Cheesecake(v), 1.99
     -- Creamy New York cheesecake, with a chocolate graham crust
  Sorbet(v), 1.89
     -- A scoop of raspberry and a scoop of lime

CAFE MENU, Dinner
---------------------
  Veggie Burger and Air Fries(v), 3.99
     -- Veggie burger on a whole wheat bun, lettuce, tomato, and fries
  Soup of the day, 3.69
     -- A cup of the soup of the day, with a side salad
  Burrito(v), 4.29
     -- A large burrito, with whole pinto beans, salsa, guacamole
%
```

Все содержимое меню... И чтобы вывести его, мы просто вызвали print() для меню верхнего уровня.

Новое десертное меню выводится в числе других компонентов меню бистро.



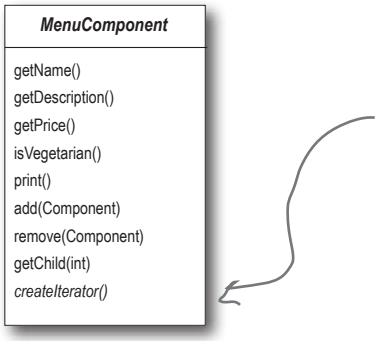
В этом замечании есть доля истины. Можно сказать, что паттерн Компоновщик нарушает принцип одной обязанности ради ***прозрачности***. Что такое «прозрачность»? Благодаря тому, что мы включили в интерфейс Component операции управления как дочерними узлами, так ***и*** листьями, клиент выполняет одинаковые операции с комбинациями и листовыми узлами. Таким образом, вид узла становится прозрачным для клиента. Однако при этом приходится частично жертвовать ***безопасностью***, потому что клиент может попытаться выполнить с элементом неподходящую или бессмысленную операцию (например, попытаться добавить меню в листовой элемент). Речь идет о сознательном архитектурном решении; мы также могли пойти по другому пути и разделить обязанности на интерфейсы. Это повысило бы безопасность архитектуры (любые неподходящие операции с элементами будут обнаруживаться на стадии компиляции), но тогда код утратит прозрачность, в нем придется использовать условные конструкции с оператором `instanceof`.

Итак, перед нами классический случай компромиссного решения. Руководствуйтесь архитектурными принципами, но всегда обращайте внимание на то, какое влияние они оказывают на нашу архитектуру. Иногда приходится намеренно выбирать решения, которые на первый взгляд противоречат принципам. В других случаях оценка зависит от точки зрения. Например, присутствие в листовых узлах операций управления дочерними узлами (add(), remove() и getChild()) выглядит противоестественно, но лист можно рассматривать как узел с нулем дочерних узлов.

Возвращение к итераторам

Несколько страниц назад мы пообещали показать, как использовать итераторы в паттерне Компоновщик. Итераторы уже использовались во внутренней реализации метода print(), но есть и другие применения: например, клиентский код может перебрать все элементы комбинации, чтобы выбрать из всего меню вегетарианские блюда.

Чтобы реализовать итератор для комбинации, мы включим в каждый компонент метод createIterator(). Начнем с абстрактного класса MenuComponent:



Метод createIterator() включается в MenuComponent. Это означает, что он должен быть реализован классами Menu и MenuItem, а вызов createIterator() для комбинации будет распространяться на все дочерние элементы.

Теперь необходимо реализовать этот метод в классах Menu и MenuItem:

```
public class Menu extends MenuComponent {
    Iterator<MenuComponent> iterator = null;
    // остальной код не изменяется

    public Iterator<MenuComponent> createIterator() {
        if (iterator == null) {
            iterator = new CompositeIterator(menuComponents.iterator());
        }
        return iterator;
    }
}

public class MenuItem extends MenuComponent {

    // остальной код не изменяется

    public Iterator<MenuComponent> createIterator() {
        return new NullIterator();
    }

}
```

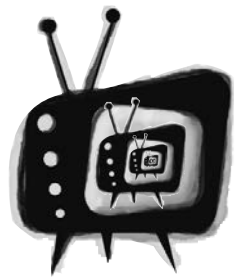
Здесь используется новый итератор CompositeIterator. Он умеет перебирать элементы любой комбинации.

Передаем итератор текущей комбинации.

Переходим к MenuItem...

Стоп! А что такое NullIterator? Через пару страниц узнаете.

Итератор CompositeIterator



Смотри в оба: впереди рекурсия!

CompositeIterator — НЕТРИВИАЛЬНЫЙ итератор. Он перебирает элементы MenuItem в комбинации, а также следит за тем, чтобы в перебор были включены все дочерние меню (а также их дочерние меню, и т. д.).

Код итератора приведен ниже. Будьте внимательны: объем кода невелик, но в отдельных местах разобраться непросто. Повторяйте про себя: «Рекурсия — мой друг, рекурсия — мой друг...»

Как и все итераторы, реализует интерфейс java.util.Iterator.

```
import java.util.*;

public class CompositeIterator implements Iterator {
    Stack<Iterator<MenuComponent>> stack = new Stack<Iterator<MenuComponent>>();

    public CompositeIterator(Iterator iterator) {
        stack.push(iterator);
    }

    public Object next() {
        if (hasNext()) {
            Iterator<MenuComponent> iterator = stack.peek();
            MenuComponent component = iterator.next();

            stack.push(component.createIterator());

            return component;
        } else {
            return null;
        }
    }

    public boolean hasNext() {
        if (stack.empty()) {
            return false;
        } else {
            Iterator<MenuComponent> iterator = stack.peek();
            if (!iterator.hasNext()) {
                stack.pop();
                return hasNext();
            } else {
                return true;
            }
        }
    }

}
```

Здесь передается итератор комбинации верхнего уровня. Мы сохраняем его в стеке.

Когда клиент запрашивает следующий элемент, мы сначала проверяем его существование вызовом hasNext()...

Если следующий элемент существует, мы извлекаем текущий итератор из стека и получаем следующий элемент.

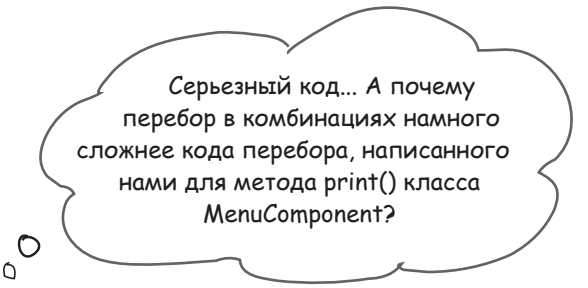
Затем итератор этого компонента заносится в стек. Если компонент относится к классу Menu — обнаружена очередная комбинация, которую необходимо включить в перебор; соответственно, мы заносим его в стек. В любом случае метод возвращает компонент.

Чтобы проверить наличие следующего компонента, мы проверяем, пуст ли стек.

Если стек не пуст, читаем из стека верхний итератор и проверяем, есть ли в стеке следующий элемент. Если нет, метод извлекает компонент из стека и рекурсивно вызывает hasNext().

А если есть, метод возвращает true.

Мы не поддерживаем удаление, поэтому не определяем реализацию и оставляем поведение по умолчанию из java.util.iterator.



При написании метода print() класса MenuComponent мы использовали итератор для перебора всех узлов компонента. Если узел был объектом Menu (а не MenuItem), то для его обработки рекурсивно вызывался метод print(). Иначе говоря, объект MenuComponent выполнял перебор сам, в своей *внутренней* реализации.

В этом коде реализуется ***внешний*** итератор, поэтому нам приходится учитывать много дополнительных обстоятельств. Для начала внешний итератор должен сохранять текущую позицию перебора, чтобы внешний клиент мог управлять перебором, вызывая методы hasNext() и next(). Но в нашей ситуации код должен сохранять текущую позицию в комбинационной рекурсивной структуре. По этой причине для отслеживания текущей позиции при перемещении вверх-вниз по комбинационной иерархии используются стеки.

Нарисуйте диаграмму с объектами Menu и MenuItem. Затем представьте, что вы — итератор CompositeIterator, который должен обрабатывать вызовы hasNext() и next(). Нарисуйте маршрут перебора структуры при выполнении следующего кода:

```
public void testCompositeIterator(MenuComponent component) {
       CompositeIterator iterator = new CompositeIterator(component.iterator);

       while(iterator.hasNext()) {
              MenuComponent component = iterator.next();
       }
}
```

Пустой итератор

Итак, что же это такое пустой итератор? Это можно представить себе так: у объектов MenuItem нет компонентов для перебора, верно? Как же реализовать для них метод createIterator()? Возможны два варианта:

Еще один пример «паттерна» Пустой Объект.

Вариант 1:

Вернуть null

Но тогда в клиентский код придется включить условную конструкцию, которая будет проверять, вернул метод null или нет.

Вариант 2:

Вернуть итератор, который всегда возвращает false при вызове hasNext()

Такой план выглядит более разумно. Метод, как и положено, возвращает итератор, а клиенту не нужно проверять полученное значение. Фактически мы создаем «пустой» итератор, не выполняющий операций.

Самый ленивый итератор, который только можно представить: не хочет сделать ни одного шага.

Конечно, второй вариант выглядит предпочтительно. Реализуем его под именем NullIterator:

```
import java.util.Iterator;

public class NullIterator implements <MenuComponent> {

    public Object next() {
        return null;
    }

    public boolean hasNext() {
        return false;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

При вызове next() возвращается null.

Но самое важное, что при вызове hasNext() всегда возвращается false.

И конечно, NullIterator не поддерживает удаление. Реализовывать не обязательно; можно просто доверить обработку реализации по умолчанию java.util.iterator.remove.

Дайте мне вегетарианское меню

Теперь мы можем перебирать все элементы Menu. Давайте воспользуемся этим обстоятельством и реализуем в классе Waitress метод, который будет отбирать из всего меню вегетарианские блюда:

```
public class Waitress {
    MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    public void printMenu() {
        allMenus.print();
    }

    public void printVegetarianMenu() {
        Iterator<MenuComponent> iterator = allMenus.createIterator();
        System.out.println("\nVEGETARIAN MENU\n----");
        while (iterator.hasNext()) {
            MenuComponent menuComponent = iterator.next();
            try {
                if (menuComponent.isVegetarian()) {
                    menuComponent.print();
                }
            } catch (UnsupportedOperationException e) {}
        }
    }
}
```

Метод printVegetarianMenu() получает комбинацию allMenu и создает для нее итератор (наш класс CompositeIterator).

Перебор всех элементов комбинации.

Для каждого элемента вызывается метод isVegetarian(), и если он возвращает true — для элемента вызывается метод print().

Метод print() вызывается только для MenuItem, и никогда для комбинаций. А вы видите, почему?

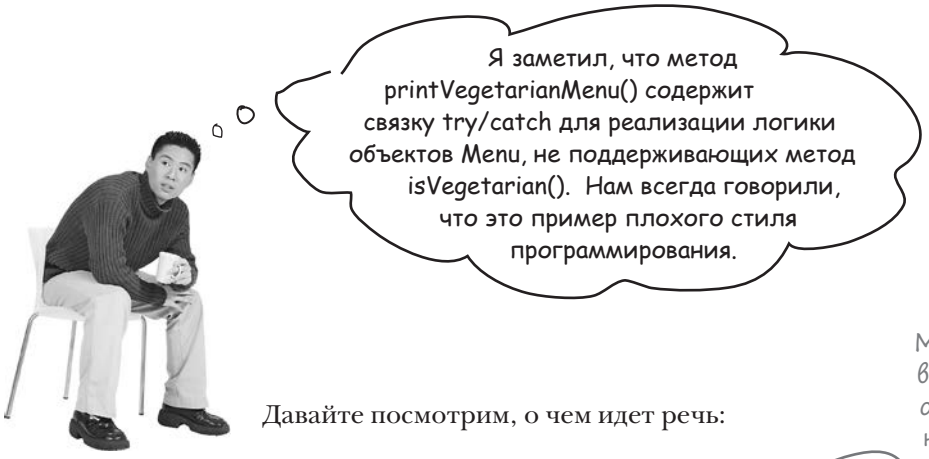
Реализация isVegetarian() для Menu всегда генерирует исключение. Если это произойдет, мы перехватываем исключение и продолжаем перебор.

Магия итераторов и композиций

Ого! Чтобы код пришел к своему текущему состоянию, нам пришлось изрядно потрудиться. Но зато теперь у нас имеется обобщенная структура меню, которая обеспечит потребности растущей империи быстрого питания. Теперь вы сможете заказать себе вегетарианские блюда:

```
File Edit Window Help HaveUhuggedYurIteratorToday?
% java MenuTestDrive

VEGETARIAN MENU
----
  K&B's Pancake Breakfast(v), 2.99
     -- Pancakes with scrambled eggs, and toast
  Blueberry Pancakes(v), 3.49
     -- Pancakes made with fresh blueberries, and blueberry syrup
  Waffles(v), 3.59
     -- Waffles, with your choice of blueberries or strawberries
  Vegetarian BLT(v), 2.99
     -- (Fakin') Bacon with lettuce & tomato on whole wheat
  Steamed Veggies and Brown Rice(v), 3.99
     -- Steamed vegetables over brown rice
  Pasta(v), 3.89
     -- Spaghetti with Marinara Sauce, and a slice of sourdough bread
  Apple Pie(v), 1.59
     -- Apple pie with a flakey crust, topped with vanilla icecream
  Cheesecake(v), 1.99
     -- Creamy New York cheesecake, with a chocolate graham crust
  Sorbet(v), 1.89
     -- A scoop of raspberry and a scoop of lime
  Veggie Burger and Air Fries(v), 3.99
     -- Veggie burger on a whole wheat bun, lettuce, tomato, and fries
  Burrito(v), 4.29
     -- A large burrito, with whole pinto beans, salsa, guacamole
%
```

Вегетарианское меню складывается из вегетарианских элементов каждого меню.



Давайте посмотрим, о чем идет речь:

```
try {
    if (menuComponent.isVegetarian()) {
        menuComponent.print();
    }
} catch (UnsupportedOperationException) {}
```

Метод isVegetarian() вызывается для всех объектов MenuComponent, но объекты Menu инициируют исключение, потому что они не поддерживают эту операцию.

Если компонент не поддерживает операцию, мы попросту игнорируем перехваченное исключение.

В общем случае это замечание справедливо; конструкция try/catch предназначена для обработки ошибок, а не для реализации программной логики. Какие еще возможны варианты? Можно перед вызовом isVegetarian() проверить фактический тип компонента меню оператором instanceof и убедиться, что он относится к типу MenuItem. Но тогда мы потеряем ***прозрачность***, потому что это будет означать, что Menu и MenuItem обрабатываются по-разному.

Также можно изменить метод isVegetarian() в Menu, чтобы он возвращал false. Это простое решение, сохраняющее прозрачность.

В нашем решении мы выбираем логическую ясность: перехват исключения ясно показывает, что эта операция для Menu не поддерживается (что принципиально отличается от возвращения false методом isVegetarian()). Кроме того, такое решение позволяет предоставить разумную реализацию метода isVegetarian() для Menu; она будет работать с существующим кодом.

Нам этот вариант кажется более предпочтительным.

ПАТТЕРНЫ ДЛЯ ВСЕХ

Интервью недели: паттерн Компоновщик о проблемах реализации

HeadFirst: Сегодня у нас в гостях паттерн Компоновщик. Почему бы вам не рассказать немного о себе?

Компоновщик: Да, конечно... Я — паттерн, используемый при работе с коллекциями объектов, связанных отношениями «часть–целое», когда вы хотите выполнять однородные операции с такими объектами.

HeadFirst: А здесь подробнее, пожалуйста... Что вы подразумеваете под отношениями «часть–целое»?

Компоновщик: Представьте графический интерфейс; в нем используются многочисленные компоненты верхнего уровня (Frame, Panel и т. д.), содержащие другие компоненты: меню, текстовые панели, полосы прокрутки и кнопки. Таким образом интерфейс делится на части, но в нашем представлении он обычно воспринимается как единое целое. Вы приказываете компоненту верхнего уровня перерисовать себя и рассчитываете на то, что он правильно прорисует все свои части. Компоненты, содержащие другие компоненты, называются комбинационными (или составными) объектами, а компоненты, не содержащие других компонентов, — листовыми объектами.

HeadFirst: А что подразумевается под однородностью операций? Наличие общих методов, которые могут вызываться для комбинаций и для листьев?

Компоновщик: Точно. Я приказываю комбинационному или листовому объекту отобразить себя на экране, и он делает то, что от него требуется. Комбинационный объект для этого приказывает всем своим компонентам отобразить себя.

HeadFirst: Из этого следует, что все объекты обладают одинаковым интерфейсом. А если объекты комбинации решают разные задачи?

Компоновщик: Чтобы комбинация была прозрачной с точки зрения клиента, все ее объекты должны реализовать единый интерфейс; в противном случае клиенту придется беспокоиться о том, какой интерфейс реализует каждый объект, что противоречит основному предназначению паттерна. Разумеется, отсюда следует, что вызовы некоторых методов для некоторых объектов не имеют смысла.

HeadFirst: И что делать в таких случаях?

Компоновщик: Есть пара возможных решений: иногда можно не делать ничего, вернуть null или false — что лучше подойдет для вашего приложения. В других случаях выбирается более активный путь — метод инициирует исключение. Конечно, клиенту придется проделать дополнительную работу и убедиться в том, что при вызове метода не произошло ничего непредвиденного.

HeadFirst: Если клиент не знает, с каким объектом он имеет дело, то как он без проверки типа определит, какие методы для него можно вызывать?

Компоновщик: При некоторой изобретательности можно структурировать методы так, что реализация по умолчанию будет делать что-то осмысленное. Например, вызов getChild() для ком-

бинации имеет смысл. Но он имеет смысл и для листа, если рассматривать последний как объект без дочерних компонентов.

HeadFirst: Ага... умно. Но некоторых клиентов настолько беспокоит эта проблема, что они требуют определения разных интерфейсов для разных объектов, чтобы избежать вызова бессмысленных методов. Можно ли это назвать паттерном Компоновщик?

Компоновщик: Да. Эта версия паттерна Компоновщик намного безопаснее, но в ней клиент должен проверить тип каждого объекта перед вызовом метода, чтобы выполнить правильное преобразование.

HeadFirst: Расскажите нам подробнее о структуре комбинаций и листовых объектов.

Компоновщик: Обычно создается древовидная структура — некое подобие иерархии. Корнем дерева является комбинация верхнего уровня, а ее потомки являются либо комбинациями, либо листовыми узлами.

HeadFirst: Дочерние узлы содержат информацию о своих родителях?

Компоновщик: Да, компонент может хранить указатель на родительский узел для упрощения перебора. А если у вас имеется ссылка на дочерний узел, подлежащий удалению, для выполнения операции необходимо перейти к родителю. Наличие ссылки упрощает такой переход.

HeadFirst: Есть ли еще что-то, о чем необходимо помнить при реализации паттерна Компоновщик?

Компоновщик: Да, есть. Например, порядок дочерних компонентов. Если дочерние компоненты комбинации должны храниться в определенном порядке, то понадобится более сложная схема добавления и удаления, а также перемещения по иерархии.

HeadFirst: Верно, это я как-то не учел.

Компоновщик: А кэширование?

HeadFirst: Кэширование?

Компоновщик: Да. Если перемещение по комбинационной структуре обходится слишком дорого (например, из-за ее сложности), полезно реализовать кэширование комбинационных узлов. Например, если вы постоянно перебираете комбинацию с ее дочерними узлами для вычисления некоторого значения, стоит реализовать механизм кэширования для временного хранения результата, чтобы избежать лишних вычислений.

HeadFirst: Да, при реализации паттерна Компоновщик приходится учитывать больше обстоятельств, чем видно на первый взгляд. Прежде чем завершать наше интервью, позвольте задать еще один вопрос: что вы считаете своим главным преимуществом?

Компоновщик: Могу с уверенностью сказать, что я упрощаю жизнь своим клиентам. Им не приходится беспокоиться о том, работают ли они с комбинационным объектом или листом, а значит, не придется использовать конструкции if, чтобы убедиться в том, что они вызывают правильные методы для правильных объектов. Часто для выполнения операции со всей структурой достаточно вызова одного метода.

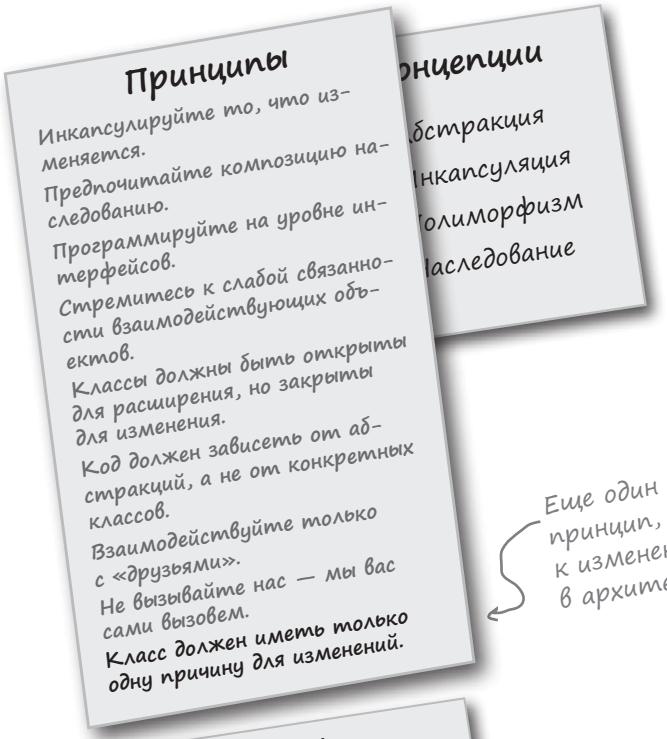
HeadFirst: Да, это важное преимущество. Несомненно, вы — весьма полезный паттерн для управления коллекциями. Однако наше время подходит к концу... Спасибо всем, кто был с нами. До встречи на следующем интервью из серии «Паттерны для всех».

Соедините каждый паттерн с его описанием:

Паттерн	Описание
Стратегия	Клиент выполняет одинаковые операции с коллекциями и отдельными объектами
Адаптер	Предоставляет механизм перебора коллекций без раскрытия внутренней реализации.
Итератор	Упрощает интерфейс группы классов
Фасад	Изменяет интерфейс одного или нескольких классов
Компоновщик	Оповещает группу объектов об изменениях состояния
Наблюдатель	Инкапсулирует взаимозаменяемые варианты поведения и выбирает один из них посредством делегирования

Новые инструменты

Два новых инструмента — два полезных способа работы с коллекциями объектов.



КЛЮЧЕВЫЕ МОМЕНТЫ

- Итератор предоставляет доступ к элементам коллекции без раскрытия ее внутренней структуры.
- Итератор обеспечивает перебор элементов коллекции и его инкапсуляцию в отдельном объекте.
- При использовании итераторов коллекция избавляется от одной обязанности (поддержки операций перебора данных).
- Итератор предоставляет общий интерфейс перебора элементов коллекции, что позволяет применять полиморфизм в коде, использующем элементы коллекции.
- Каждому классу следует по возможности назначать только одну обязанность.
- Паттерн Компоновщик предоставляет структуру для хранения как отдельных объектов, так и комбинаций.
- Паттерн Компоновщик позволяет клиенту выполнять однородные операции с комбинациями и отдельными объектами.
- В реализации паттерна Компоновщик приходится искать баланс между прозрачностью, безопасностью и вашими потребностями.

Возьми в руку карандаш
Решение

Какие из следующих утверждений относятся к нашей реализации printMenu()?

- ☑ A. Мы программируем для конкретных реализаций PancakeHouseMenu и DinerMenu, а не для интерфейсов.
- ☐ B. Официантка не реализует Java Waitress API, а следовательно, не соответствует стандарту.
- ☑ C. Если мы решим перейти с DinerMenu на другое меню с реализацией на базе Hashtable, нам придется изменять большой объем кода.
- ☑ D. Официантка должна знать, как в каждом объекте меню организована внутренняя коллекция элементов, а это нарушает инкапсуляцию.
- ☑ E. В реализации присутствует дублирование кода: метод printMenu() содержит два разных цикла для перебора двух разновидностей меню. А при появлении третьего меню понадобится еще один цикл.
- ☐ F. Реализация не использует язык MXML (Menu XML), что снижает ее универсальность.

Возьми в руку карандаш
Решение

А как бы вы подошли к реализации новых требований к архитектуре? Подумайте, прежде чем перевернуть страницу.

1. Реализовать интерфейс Menu.
2. Избавиться от getItems().
3. Добавить метод createIterator() и возвращать итератор для перебора значений из Hashtable.

Магниты с кодами. Решение

Восстановленный итератор для DinerMenu с чередованием элементов

```
import java.util.Iterator;
import java.util.Calendar;

public class AlternatingDinerMenuIterator implements Iterator<MenuItem> {

    MenuItem[] items;
    int position;

    public AlternatingDinerMenuIterator(MenuItem[] items) {

        this.items = items;
        position = Calendar.DAY_OF_WEEK % 2;

    }

    public boolean hasNext() {

        if (position >= items.length || items[position] == null) {
            return false;
        } else {
            return true;
        }

    }

    public MenuItem next() {

        MenuItem menuItem = items[position];
        position = position + 2;
        return menuItem;

    }

    public void remove() {

        throw new UnsupportedOperationException(
            "Alternating Diner Menu Iterator does not support remove()");

    }

}
```

Обратите внимание: эта реализация Iterator не поддерживает remove().

Кто и что делает? Решение

Соедините каждый паттерн с его описанием:

Состояние дел

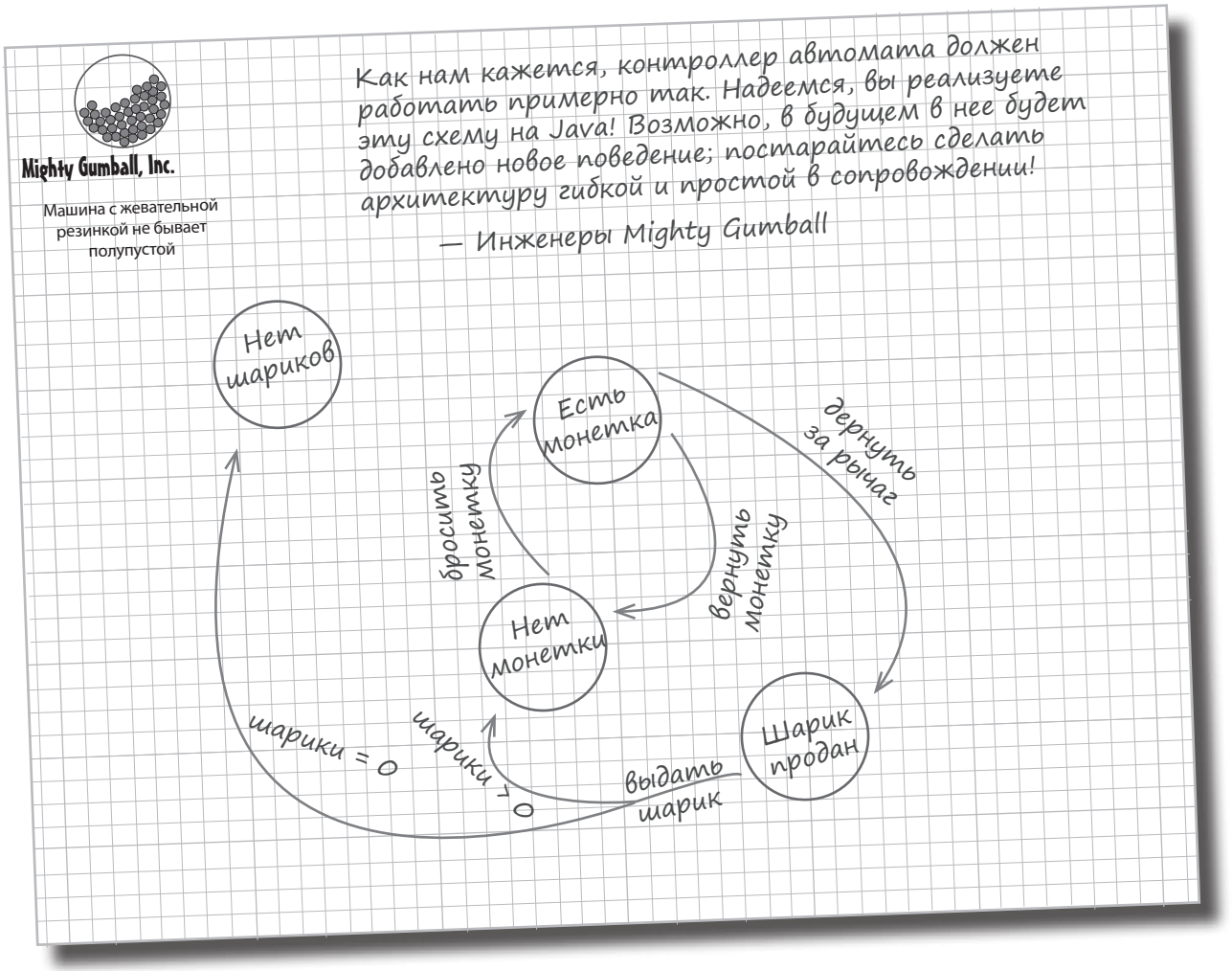
Малоизвестный факт: паттерны Стратегия и Состояние — близнецы, разлученные при рождении. Как известно, паттерн Стратегия организовал успешный бизнес в области взаимозаменяемых алгоритмов. Паттерн Состояние выбрал, пожалуй, более благородный путь: он помогает объектам управлять своим поведением посредством изменения внутреннего состояния.

Техника на грани фантастики

Тостеры с поддержкой Java остались в прошлом. В наши дни Java встраивается в *настоящую* технику — например, в автоматы для продажи шариков жевательной резинки. Да, фирмы-производители обнаружили, что установка процессора в их изделиях повышает объем продаж, позволяет следить за наличием товара по сети и более точно оценить качество обслуживания.

Но производители автоматов хорошо разбираются в жевательной резинке, а не в программировании, поэтому они обратились к вам за помощью:

Во всяком случае они так говорят, хотя, скорее всего, им просто надоела техника прошлого века и они хотят сделать свою работу более интересной.



Разговор в офисе

Джуди: Похоже на диаграмму состояний.

Джо: Верно, каждый кружок — это состояние...

Джуди: ...а стрелка — переход.

Фрэнк: Эй, не торопитесь — я слишком давно изучал диаграммы состояния. Можете в двух словах напомнить, что это такое?

Джуди: Конечно, Фрэнк. Посмотри на кружки; это состояния. Вероятно, «Нет монетки» — исходное состояние автомата, потому что он ждет, пока в него бросят монетку. Каждое состояние представляет собой некую конфигурацию автомата, а переход в другое состояние происходит в результате некой операции.

Джо: Точно. Чтобы перейти в другое состояние, нужно что-то сделать — скажем, бросить в автомат монетку. Видишь стрелку от кружка «Нет монетки» к кружку «Есть монетка»?

Фрэнк: Вижу...

Джо: Если автомат находится в состоянии «Нет монетки» и ты бросишь в него монетку, то он перейдет в состояние «Есть монетка». Происходит *переход состояния*.

Фрэнк: Ага, понятно! А если автомат находится в состоянии «Есть монетка», я могу либо дернуть за рычаг для перехода в состояние «Шарик продан», либо нажать кнопку возврата и вернуться в состояние «Нет монетки».

Джуди: Точно!

Фрэнк: Вроде бы ничего страшного. Четыре состояния, да и действий, похоже, тоже четыре: «бросить монетку», «вернуть монетку», «дернуть за рычаг» и «выдать шарик». Но... потом мы проверяем количество шариков в состоянии «Шарик продан» и переходим либо в состояние «Нет шариков», либо в состояние «Нет монетки». Значит, всего пять переходов между состояниями.

Джуди: Проверка количества шариков также подразумевает, что мы должны где-то хранить текущее количество шариков. Каждый выданный шарик может оказаться последним; для такого случая и предусмотрен переход в состояние «Нет шариков».

Джо: И не забудьте, что пользователь может сделать что-нибудь бессмысленное: попытаться вернуть монетку, когда ее нет, или бросить сразу две монетки.

Фрэнк: Да, я об этом не подумал; придется учесть такую возможность.

Джо: Для каждого возможного действия необходимо проверить, в каком состоянии находится автомат, и выполнить соответствующую операцию. Давайте-ка запрограммируем эту диаграмму состояний...

Краткий курс конечных автоматов

Как преобразовать диаграмму состояний в программный код? Далее приводится краткое описание процесса реализации конечных автоматов:

1. Соберите все состояния:

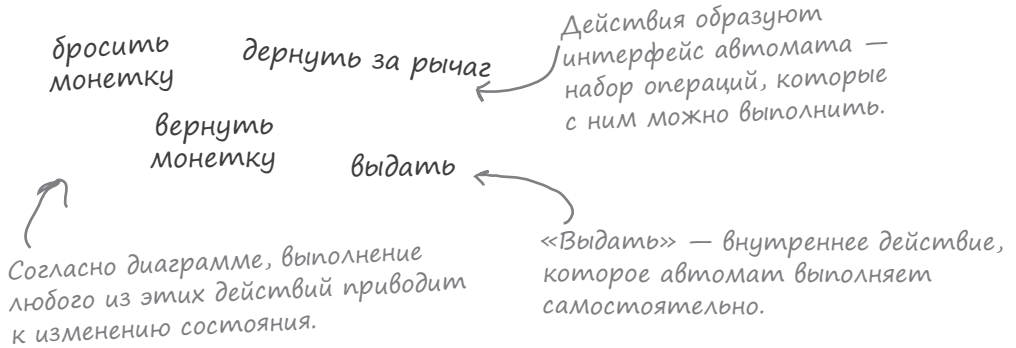
2. Создайте переменную экземпляра для хранения текущего состояния, определите значение для каждого возможного состояния:

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;

int state = SOLD_OUT;
```

Каждое состояние представлено целым числом...

...а это переменная экземпляра для хранения текущего состояния. Мы присваиваем ей исходное значение SOLD_OUT, потому что только что распакованный и включенный автомат пуст.

3. Собираем все действия, которые могут выполняться в системе:



4 Теперь мы создадим класс, который моделирует конечный автомат. Для каждого действия создается метод, который использует условные конструкции для выбора поведения, соответствующего каждому состоянию. Например, для действия «бросить монетку» пишется метод следующего вида:

```
public void insertQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("You can't insert another quarter");
    } else if (state == NO_QUARTER) {
        state = HAS_QUARTER;
        System.out.println("You inserted a quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't insert a quarter, the machine is sold out");
    } else if (state == SOLD) {
        System.out.println("Please wait, we're already giving you a gumball");
    }
}
```

Условная конструкция проверяет все возможные состояния...

...и выдает для каждого возможного состояния соответствующее поведение...

...с возможностью перехода в другое состояние, как показано на диаграмме.

После краткого обзора можно заняться реализацией автомата!

Программирование

Итак, пришло время реализовать логику автомата для продажи жевательной резинки. Мы уже выяснили, что нам понадобится переменная экземпляра для хранения текущего состояния. Что касается действий, необходимо реализовать действия «бросить монетку», «вернуть монетку», «дернуть за рычаг» и «выдать шарик»; также следует реализовать состояние «Нет шариков».

```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;

    public GumballMachine(int count) {
        this.count = count;
        if (count > 0) {
            state = NO_QUARTER;
        }
    }

    public void insertQuarter() {
        if (state == HAS_QUARTER) {
            System.out.println("You can't insert another quarter");
        } else if (state == NO_QUARTER) {
            state = HAS_QUARTER;
            System.out.println("You inserted a quarter");
        } else if (state == SOLD_OUT) {
            System.out.println("You can't insert a quarter, the machine is sold out");
        } else if (state == SOLD) {
            System.out.println("Please wait, we're already giving you a gumball");
        }
    }
```

Четыре состояния соответствуют состояниям на диаграмме.

Переменная экземпляра, в которой будет храниться текущее состояние. Инициализируется значением SOLD_OUT.

Во второй переменной хранится количество шариков в автомате.

Конструктор получает исходное количество шариков. Если оно отлично от нуля, то автомат переходит в состояние NO_QUARTER, ожидая, пока кто-нибудь бросит монетку; в противном случае автомат остается в состоянии SOLD_OUT.

Начинаем реализовывать действия в виде методов....

Когда в автомат бросают монетку, если....

...в автомате уже есть монетка, мы сообщаем об этом покупателю; в противном случае автомат принимает монетку и переходит в состояние HAS_QUARTER.

Если шарик был куплен, следует подождать завершения операции, прежде чем бросать другую монетку.

А если все шарики распроданы, автомат отклоняет монетку.

```
    public void ejectQuarter() {
        if (state == HAS_QUARTER) {
            System.out.println("Quarter returned");
            state = NO_QUARTER;
        } else if (state == NO_QUARTER) {
            System.out.println("You haven't inserted a quarter");
        } else if (state == SOLD) {
            System.out.println("Sorry, you already turned the crank");
        } else if (state == SOLD_OUT) {
            System.out.println("You can't eject, you haven't inserted a quarter yet");
        }
    }

    public void turnCrank() {
        if (state == SOLD) {
            System.out.println("Turning twice doesn't get you another gumball!");
        } else if (state == NO_QUARTER) {
            System.out.println("You turned but there's no quarter");
        } else if (state == SOLD_OUT) {
            System.out.println("You turned, but there are no gumballs");
        } else if (state == HAS_QUARTER) {
            System.out.println("You turned...");
            state = SOLD;
            dispense();
        }
    }

    public void dispense() {
        if (state == SOLD) {
            System.out.println("A gumball comes rolling out the slot");
            count = count - 1;
            if (count == 0) {
                System.out.println("Oops, out of gumballs!");
                state = SOLD_OUT;
            } else {
                state = NO_QUARTER;
            }
        } else if (state == NO_QUARTER) {
            System.out.println("You need to pay first");
        } else if (state == SOLD_OUT) {
            System.out.println("No gumball dispensed");
        } else if (state == HAS_QUARTER) {
            System.out.println("No gumball dispensed");
        }
    }

    // Другие методы: toString(),  refill() и т.д.
}
```

Если покупатель пытается вернуть монетку...

Если монетка есть, автомат возвращает ее и переходит в состояние NO_QUARTER.

Если монетки нет, то вернуть ее невозможно.

Шарик уже куплен, возврат невозможен!

Если шарики кончились, возврат невозможен — автомат не принимает монетки!

Покупатель пытается дернуть за рычаг...

Кто-то пытается обмануть автомат.

Сначала нужно бросить монетку.

Выдача невозможна — в автомате нет шариков.

Покупатель получает шарик. Переходим в состояние SOLD и вызываем метод dispense().

Вызывается для выдачи шарика.

Автомат в состоянии SOLD, выдать покупку!

Если шарик был последним, автомат переходит в состояние SOLD_OUT, а если нет — возвращается к состоянию «Нет монетки».

Все это не должно происходить, но если все же произойдет — автомат выдаст ошибку, а не шарик.

Внутреннее тестирование

Похоже, мы создали надежную архитектуру на базе хорошо продуманной методологии, не правда ли? Давайте проведем небольшое внутреннее тестирование, прежде чем передавать код заказчику для оснащения реальных торговых автоматов. Наша тестовая программа:

```
public class GumballMachineTestDrive {
    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.ejectQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.ejectQuarter();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
    }
}
```

Загружаем пять шариков.

Выводим состояние автомата.

Бросаем монетку...

Дергаем за рычаг; автомат должен выдать шарик.

Снова выводим состояние.

Бросаем монетку...

Требуем ее обратно.

Дергаем за рычаг; автомат не должен выдать шарик.

И снова выводим состояние автомата.

Бросаем монетку...

Дергаем за рычаг; автомат должен выдать шарик.

Бросаем монетку...

Дергаем за рычаг; автомат должен выдать шарик.

Требуем вернуть монетку, которую не бросали.

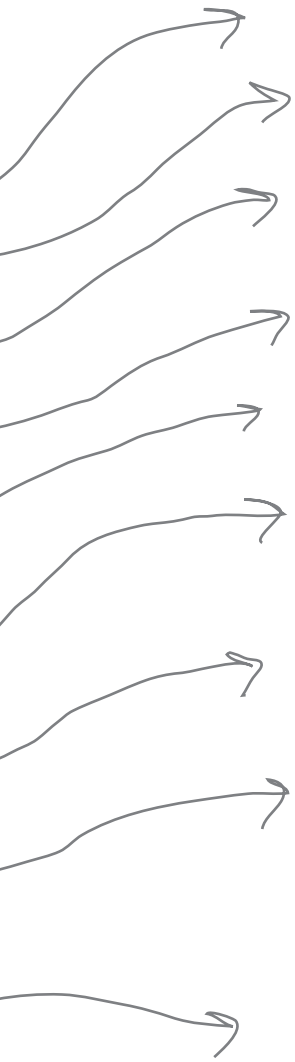
Снова выводим состояние автомата.

Бросаем ДВЕ монетки...

Дергаем за рычаг; должны получить шарик.

Даем возрастающую нагрузку... ☺

И в последний раз выводим состояние автомата.

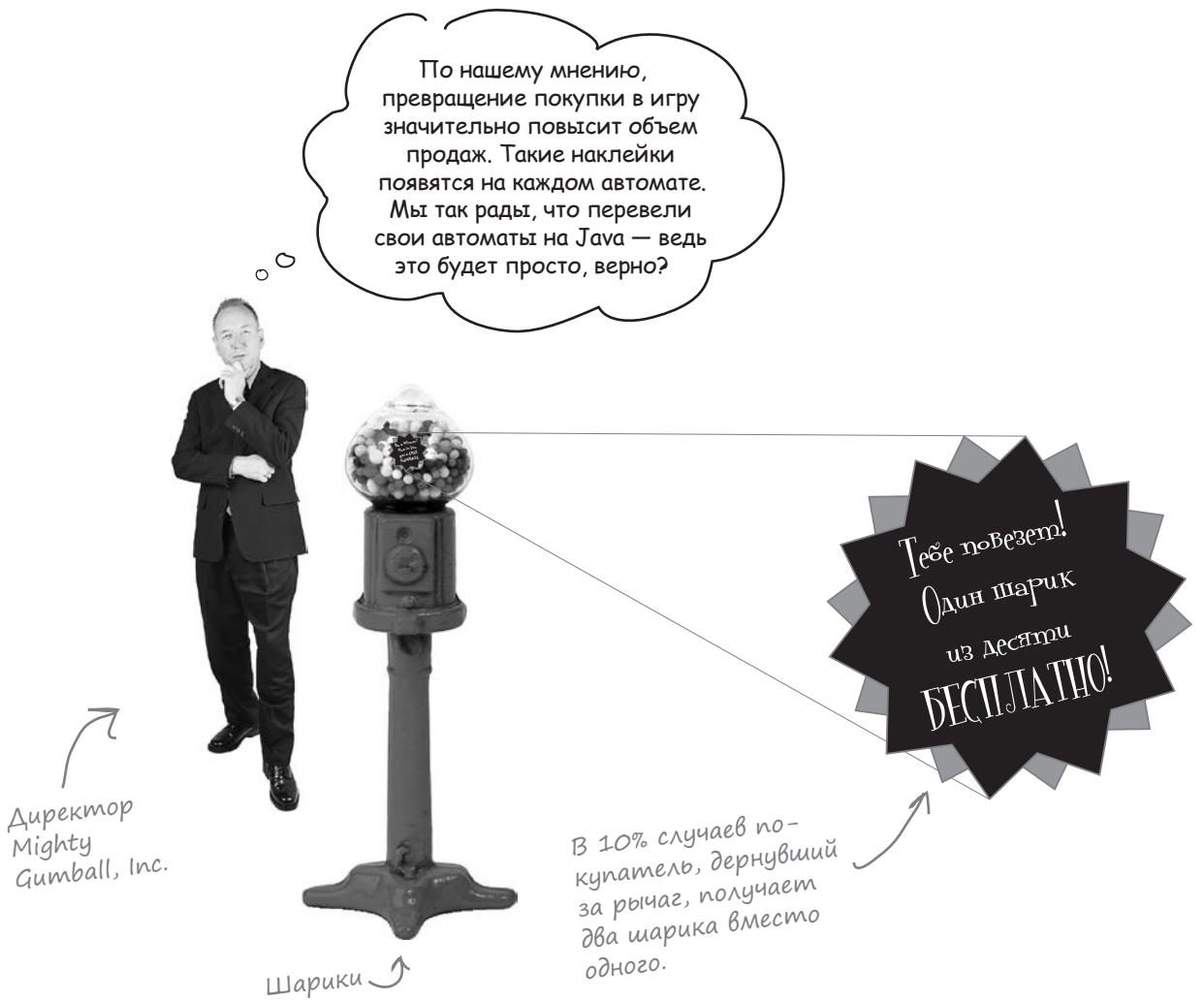


```
File  Edit  Window  Help  mightygumball.com

%java GumballMachineTestDrive

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
Quarter returned
You turned but there's no quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
You haven't inserted a quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 2 gumballs
Machine is waiting for quarter

You inserted a quarter
You can't insert another quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
Oops, out of gumballs!
You can't insert a quarter, the machine is sold out
You turned, but there are no gumballs

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 0 gumballs
Machine is sold out
```

Кто бы сомневался... запрос на изменение!

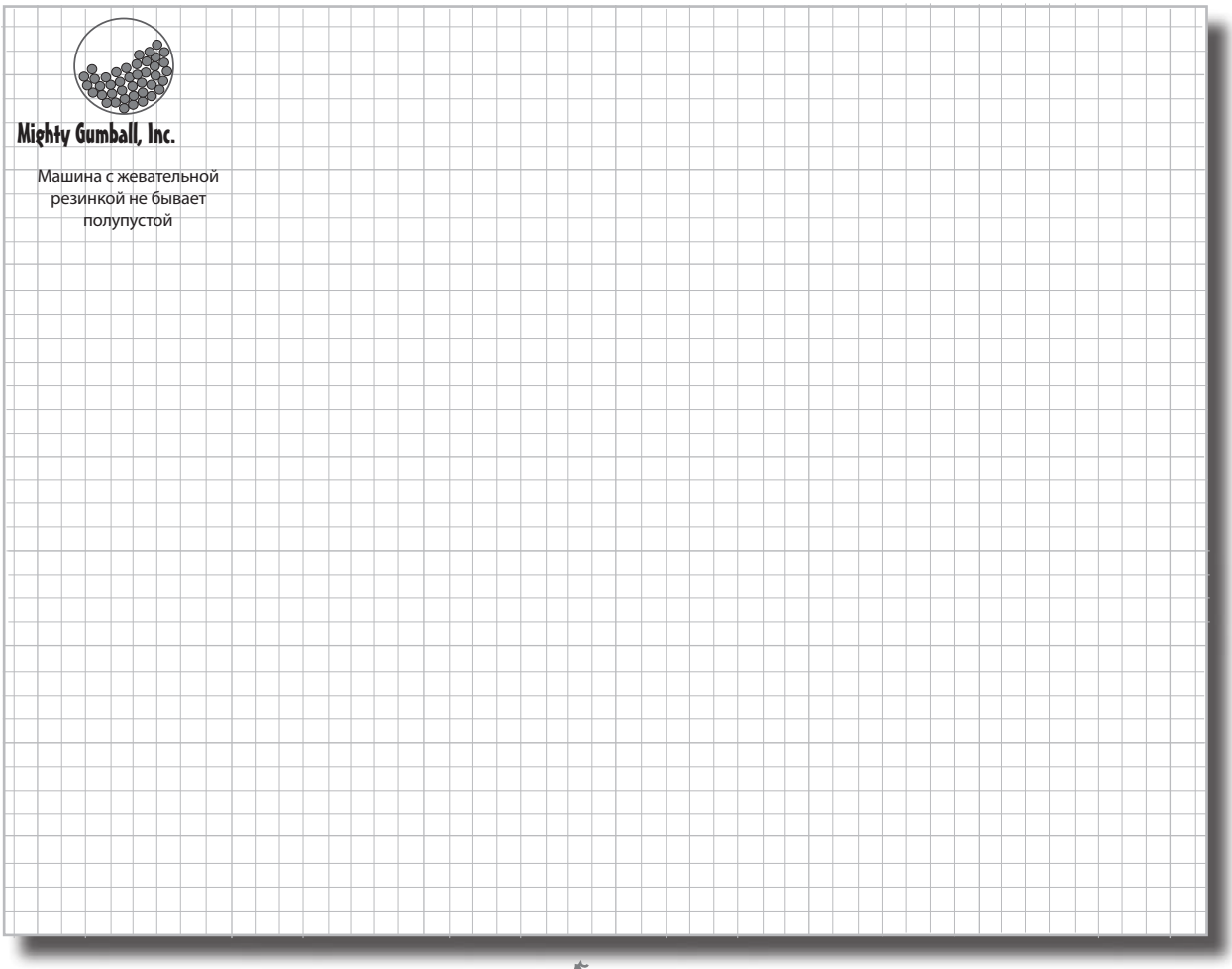
Фирма Mighty Gumball, Inc. установила ваш код на своих новейших автоматах, и специалисты по качеству испытывают его на прочность. Пока, по их мнению, все выглядит замечательно.

Настолько замечательно, что заказчик решил выйти на новый уровень...



Головоломка

Нарисуйте диаграмму состояний для автомата с поддержкой призовой игры «1 из 10». С 10%-й вероятностью при продаже автомат выдает два шарика вместо одного. Прежде чем двигаться дальше, сверьте свой ответ с нашим (приведен в конце главы).



Нарисуйте диаграмму состояний на бланке Mighty Gumball.

Печальное СОСТОЯНИЕ дел...

То, что программное обеспечение автомата написано на базе хорошо продуманной методологии, еще не значит, что оно будет легко расширяться. Стоит оглянуться на свой код и подумать, что необходимо сделать для его изменения...

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;
```

Для начала необходимо определить новое состояние WINNER. Само по себе это не так плохо...

```
public void insertQuarter() {
    // insert quarter code here
}

public void ejectQuarter() {
    // eject quarter code here
}

public void turnCrank() {
    // turn crank code here
}

public void dispense() {
    // dispense code here
}
```

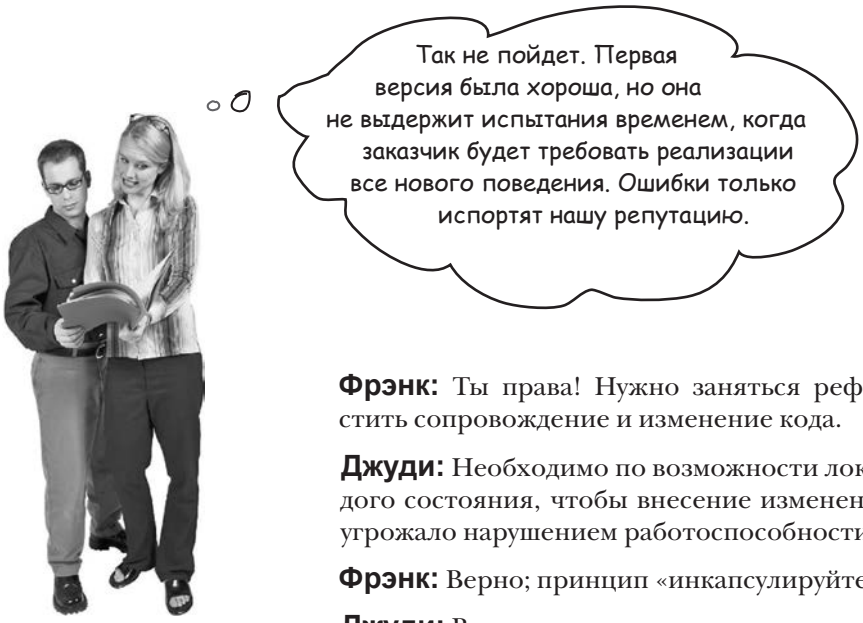
... но затем в каждый метод придется включить новое условие для обработки состояния WINNER; иначе говоря, придется изменить большой объем кода.

Особенно много хлопот будет с методом turnCrank(): придется добавить код для проверки выигрыша, а потом переходить либо в состояние WINNER, либо в состояние SOLD.

Возьми в руку карандаш

Какие из следующих утверждений относятся к нашей реализации? (Выберите все подходящие утверждения.)

- ❑ A. Код не соответствует принципу открытости/закрытости.
- ❑ B. Такой стиль программирования характерен для FORTRAN.
- ❑ C. Архитектуру трудно назвать объектно-ориентированной.
- ❑ D. Переходы между состояниями не очевидны: они «прячутся» в множестве условных конструкций.
- ❑ E. Переменные аспекты этой архитектуры не инкапсулированы.
- ❑ F. Дальнейшие изменения с большой вероятностью приведут к ошибкам в готовом коде.



Фрэнк: Ты права! Нужно заняться рефакторингом, чтобы упростить сопровождение и изменение кода.

Джуди: Необходимо по возможности локализовать поведение каждого состояния, чтобы внесение изменений в одном состоянии не угрожало нарушением работоспособности другого кода.

Фрэнк: Верно; принцип «инкапсулируйте то, что изменяется».

Джуди: Вот именно.

Фрэнк: А если выделить поведение каждого состояния в отдельный класс, то каждое состояние будет просто реализовывать свои действия.

Джуди: Точно. А автомат будет просто делегировать выполнение операций объекту, представляющему текущее состояние.

Фрэнк: Ага, «отдавайте предпочтение композиции перед наследованием»... Снова принципы в деле.

Джуди: Я не уверена на 100%, что это решение сработает, но по крайней мере это на что-то похоже.

Фрэнк: Упростит ли оно добавление новых состояний?

Джуди: Пожалуй... Код все равно придется менять, но изменения будут значительно менее масштабными, потому что для добавления нового состояния нам придется добавить новый класс. И, возможно, переделать пару-тройку переходов.

Фрэнк: Идея мне нравится. Беремся за дело!

Новая архитектура

Появился новый план: вместо сопровождения и модификации готового кода мы переработаем его, инкапсулируем объекты состояния в отдельных классах, а затем при выполнении действия делегируем операции объекту текущего состояния.

Такой подход соответствует основным принципам проектирования, так что полученная в конечном счете архитектура будет более простой в сопровождении. Основная последовательность действий:

1. **Определяем интерфейс State, содержащий метод для каждого возможного действия.**

2. **Реализуем класс State для каждого состояния автомата. Эти классы определяют поведение автомата, находящегося в соответствующем состоянии.**

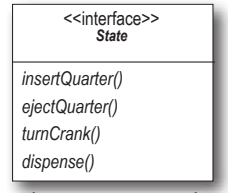
3. **Наконец, мы избавляемся от всего лишнего кода и делегируем выполнение операций классу состояния.**

Как вы вскоре убедитесь, мы не просто следуем принципам проектирования, а реализуем паттерн Состояние. Но формальное определение этого паттерна будет приведено после того, как мы переработаем наш код...

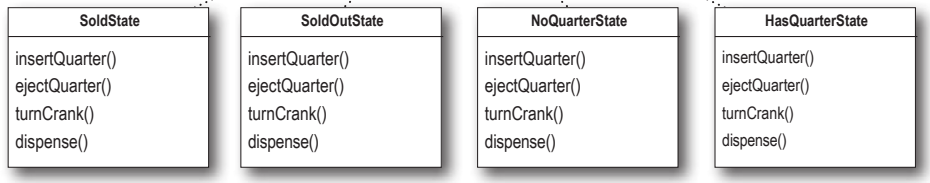
Определение интерфейса State и классов

Начнем с создания интерфейса State, реализуемого всеми классами состояний:

Интерфейс всех состояний. Методы непосредственно соответствуют действиям, которые могут выполняться с автоматом (и являются аналогами методов из предыдущей версии).



Затем каждое состояние архитектуры инкапсулируется в классе, реализующем интерфейс State.

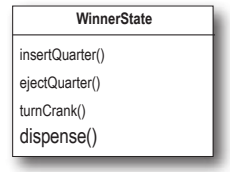


Чтобы понять, какие состояния нам понадобятся, обратимся к предыдущей версии кода...

```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;
```

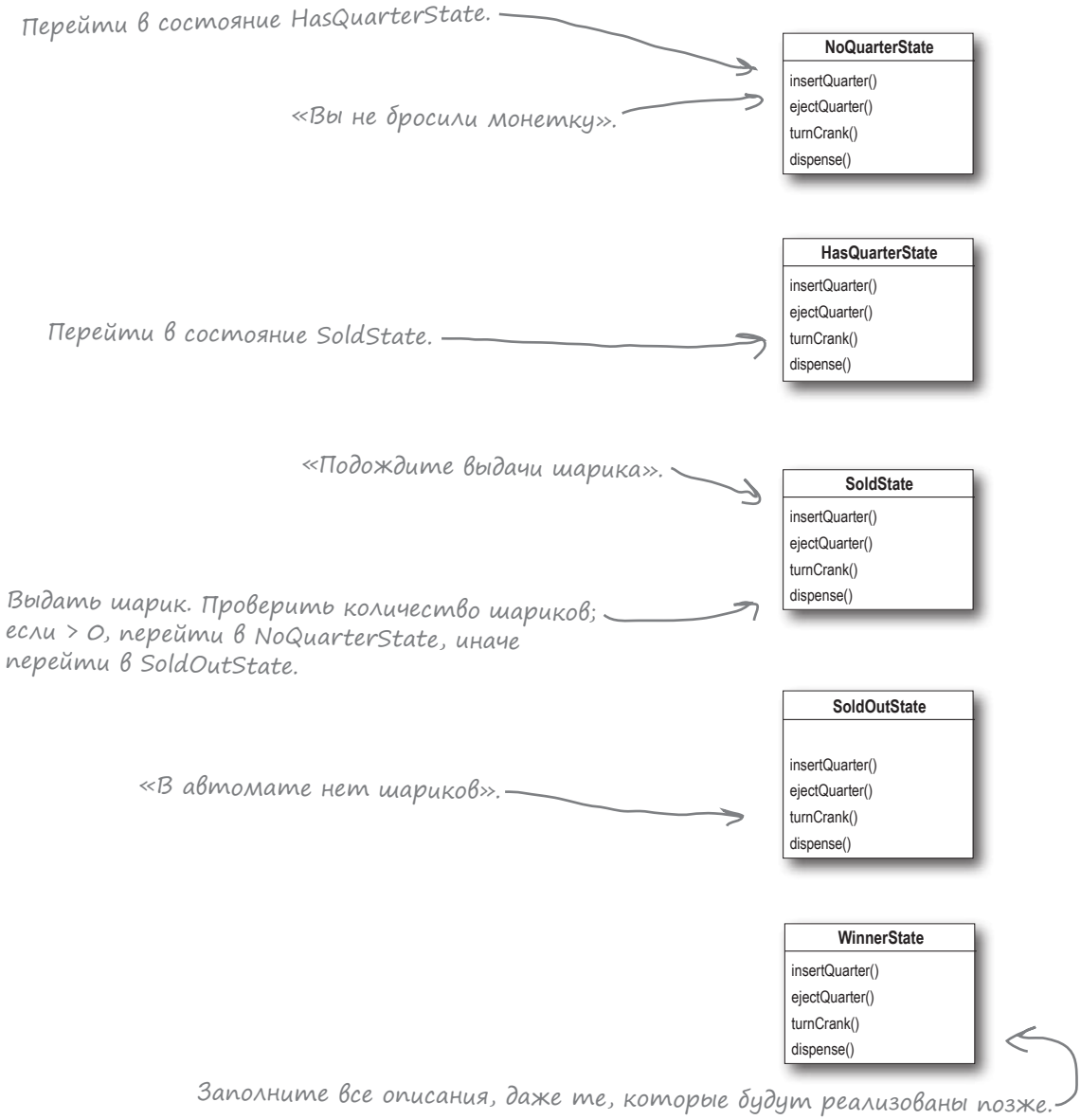
... для каждого состояния создается отдельный класс.

И не забудьте о новом состоянии, которое тоже должно реализовать интерфейс State. Мы вернемся к нему после переработки первой версии кода.



Возьми в руку карандаш

Реализация состояний начинается с определения поведения классов при выполнении каждого действия. Заполните следующую диаграмму описаниями поведения каждого класса; мы уже заполнили некоторые описания за вас.



Реализация классов состояний

Мы определились с желаемым поведением; осталось реализовать его в программном коде. Итоговый код принципиально не отличается от написанного ранее кода конечного автомата, но на этот раз он разбит на классы.

Начнем с состояния NoQuarterState:

Класс должен реализовать интерфейс State.

Конструктору передается ссылка на объект автомата, которая просто сохраняется в переменной экземпляра.

```
public class NoQuarterState implements State {
    GumballMachine gumballMachine;

    public NoQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }

    public void ejectQuarter() {
        System.out.println("You haven't inserted a quarter");
    }

    public void turnCrank() {
        System.out.println("You turned, but there's no quarter");
     }

    public void dispense() {
        System.out.println("You need to pay first");
    }
}
```

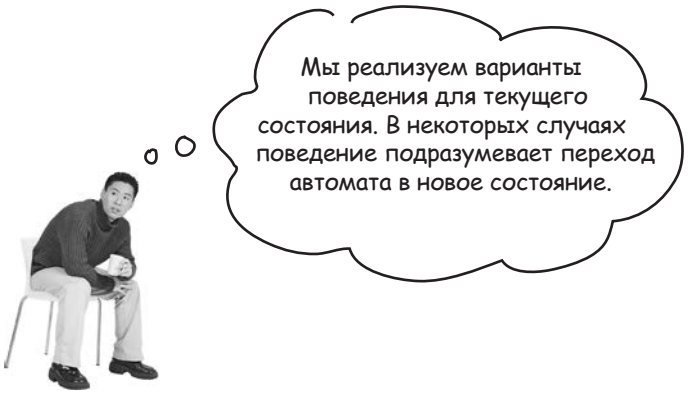
Если в автомат брошена монетка, вывести сообщение и перейти в состояние HasQuarterState.

Скоро вы поймете, что здесь происходит...

Чтобы вернуть монетку, нужно ее сначала бросить!

Нет монетки — нет шарика.

Шарик выдается только за монетку.



Переработка класса GumballMachine

Прежде чем завершать рассмотрение классов состояния, мы переработаем класс автомата GumballMachine. Это поможет лучше понять суть взаимодействия компонентов новой архитектуры. Начнем с переменных экземпляров, относящихся к реализации состояния, а затем перейдем от целочисленных кодов к объектам состояния:

```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;
```

Старый код

Код автомата переводится на использование новых классов вместо статических целочисленных кодов. Две версии кода в целом похожи, просто в одной используются целые числа, а в другой объекты...

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state = soldOutState;
    int count = 0;
```

Новый код

Все объекты State создаются и инициализируются в конструкторе.

Теперь содержит объект State вместо int.

А теперь полный код класса GumballMachine...

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state;
    int count = 0;

    public GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        } else {
        state = soldOutState;
        }
    }

    public void insertQuarter() {
        state.insertQuarter();
    }
    public void ejectQuarter() {
        state.ejectQuarter();
    }
    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }

    void setState(State state) {
        this.state = state;
    }

    void releaseBall() {
        System.out.println("A gumball comes rolling out the slot...");
        if (count != 0) {
            count = count - 1;
        }
    }

    // Другие методы, включая get-методы для всех состояний...
}
```

Все возможные состояния...

...и переменная экземпляра для хранения State.

В переменной count хранится количество шариков — изначально автомат пуст.

Конструктор получает исходное количество шариков и сохраняет его в переменной.

Он также создает экземпляры State для всех состояний.

Если количество шариков >0, устанавливается состояние NoQuarterState; в противном случае начинаем в состоянии SoldOutState.

Действия реализуются ОЧЕНЬ ПРОСТО: операция делегируется объекту текущего состояния.

Для метода dispense() в классе GumballMachine метод действия не нужен, потому что это внутреннее действие; пользователь не может напрямую потребовать, чтобы автомат выдал шарик. Однако метод dispense() для объекта State вызывается из метода turnCrank().

Этот метод позволяет другим объектам (в частности, нашим объектам State) перевести автомат в другое состояние.

Вспомогательный метод releaseBall() отпускает шарик и уменьшает значение переменной count.

Такие методы, как getNoQuarterState() для получения объекта состояния или getCount() для получения количества шариков.

Реализация других состояний

К этому моменту вы уже примерно представляете, как организовано взаимодействие автомата и состояний. Давайте реализуем классы состояний HasQuarterState и SoldState...

```
public class HasQuarterState implements State {
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }

    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        gumballMachine.setState(gumballMachine.getSoldState());
    }
    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

При создании экземпляра состояния ему передается ссылка на GumballMachine. Она используется для перевода автомата в другое состояние.

Некорректное действие для этого состояния.

Вернуть монетку и перейти в состояние NoQuarterState.

Другое некорректное действие для этого состояния.

Когда покупатель дергает за рычаг, автомат переводится в состояние SoldState (вызовом метода setState() с объектом SoldState). Для получения объекта SoldState используется метод getSoldState() (один из get-методов, определенных для всех состояний).

Теперь займемся классом SoldState...

Некорректные действия для этого состояния.

```
public class SoldState implements State {
    //Конструктор и переменные экземпляров

    public void insertQuarter() {
        System.out.println("Please wait, we're already giving you a gumball");
    }

    public void ejectQuarter() {
        System.out.println("Sorry, you already turned the crank");
    }

    public void turnCrank() {
        System.out.println("Turning twice doesn't get you another gumball!");
    }
```

Здесь начинается настоящая работа...

```
    public void dispense() {
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() > 0) {
            gumballMachine.setState(gumballMachine.getNoQuarterState());
        } else {
            System.out.println("Oops, out of gumballs!");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }
}
```

Находимся в состоянии SoldState, т. е. покупка оплачена. Сначала приказываем автомату выдать шарик.

Затем проверяем количество шариков, и в зависимости от результата переходим в состояние NoQuarterState или SoldOutState.

Присмотритесь к реализации GumballMachine. При некорректном повороте рычага (скажем, если покупатель не бросил монетку) автомат все равно может выдать шарик, хотя это и не обязательно. Как решить эту проблему?

Возьми в руку карандаш

Остался всего один нереализованный класс состояния: SoldOutState. Почему бы вам не реализовать его? Тщательно продумайте поведение автомата в каждой ситуации. Проверьте ответ, прежде чем двигаться дальше...

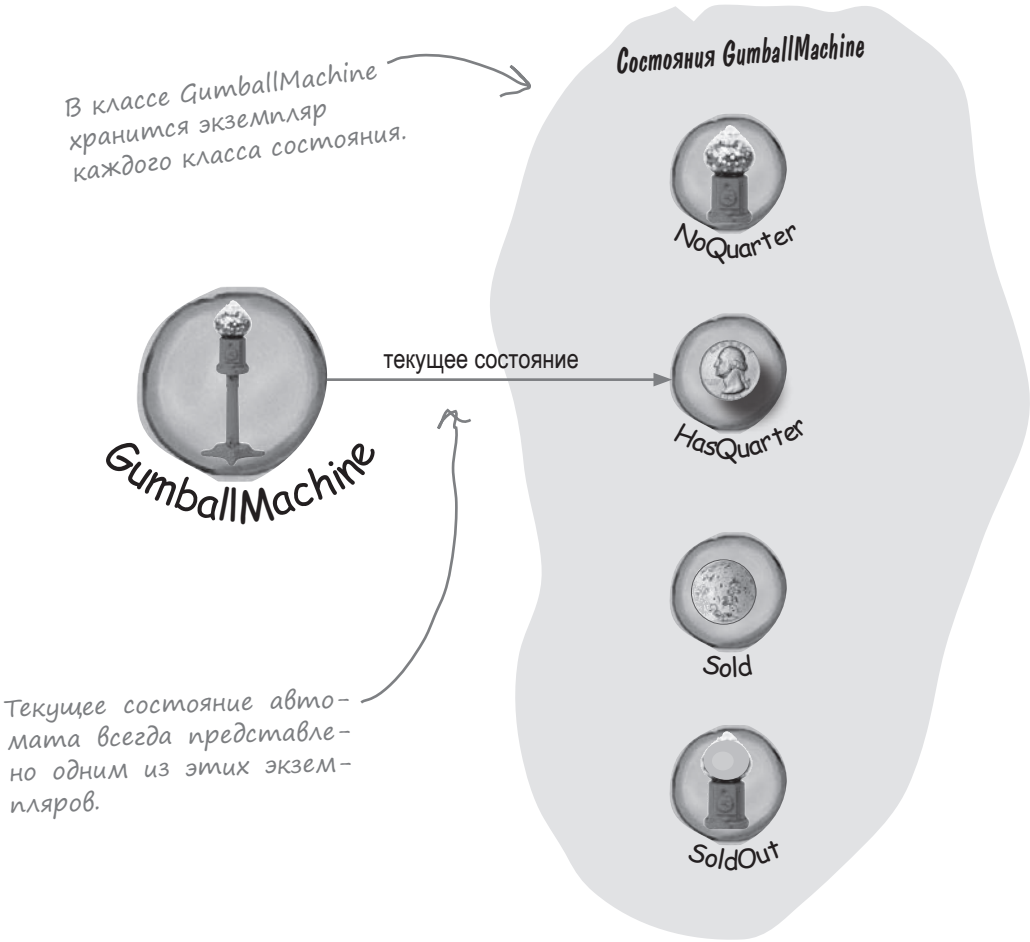
```
public class SoldOutState implements [          ] {
    GumballMachine gumballMachine;

    public SoldOutState(GumballMachine gumballMachine) {



    }

    public void insertQuarter() {



    }

    public void ejectQuarter() {



    }

    public void turnCrank() {



    }

    public void dispense() {



    }
}
```

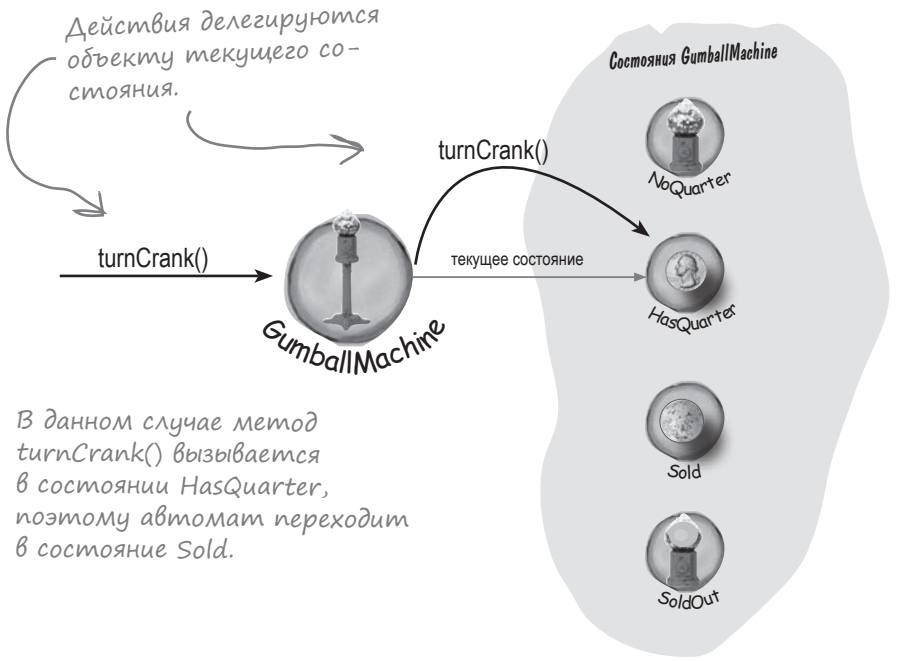
Что мы сделали к настоящему моменту...

Прежде всего мы создали реализацию торгового автомата, которая *на структурном уровне* отличается от первой версии, но при этом сохранила прежнюю *функциональность*. Преимущества структурного изменения реализации:

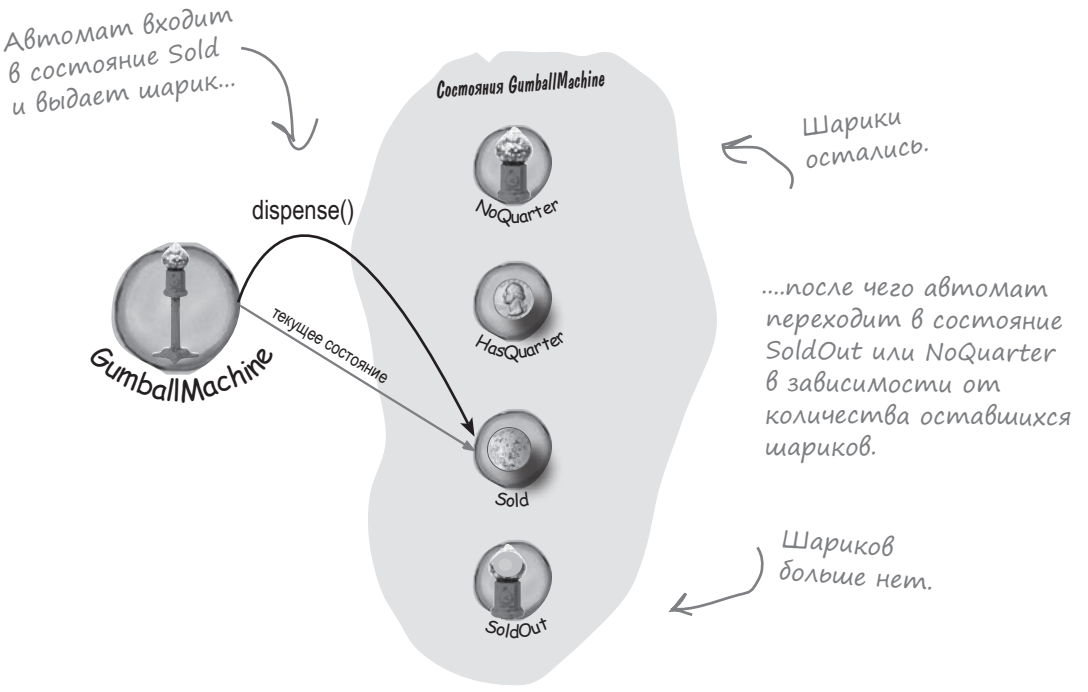
- Локализация поведения каждого состояния в отдельном классе.
- Исключение многочисленных конструкций `if`, усложнявших сопровождение кода.
- Каждое состояние закрыто для изменения, однако сам автомат открыт для расширения посредством добавления новых классов состояний (чем мы вскоре займемся).
- Создание кодовой базы и структуры классов, приближенной к диаграмме переходов автомата, а следовательно — более простой для чтения и понимания.

Чуть подробнее о функциональном аспекте того, что мы сделали:





ПЕРЕХОД В СОСТОЯНИЕ SOLD

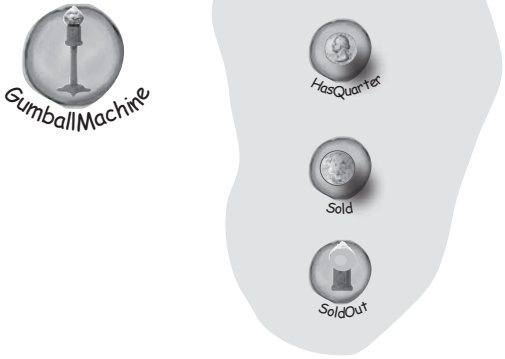


Возьми в руку карандаш

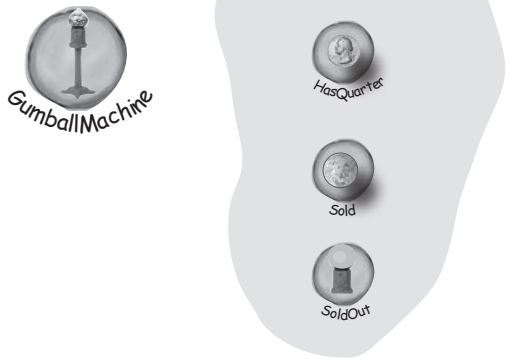
За сценой: Самостоятельное путешествие

Нарисуйте диаграмму работы автомата, начиная с состояния NoQuarter. Укажите на диаграмме действия и результаты работы автомата. В этом упражнении следует считать, что количество шариков в автомате достаточно велико.

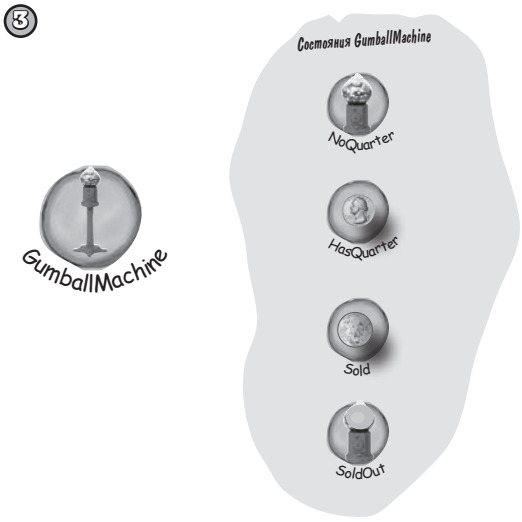
1



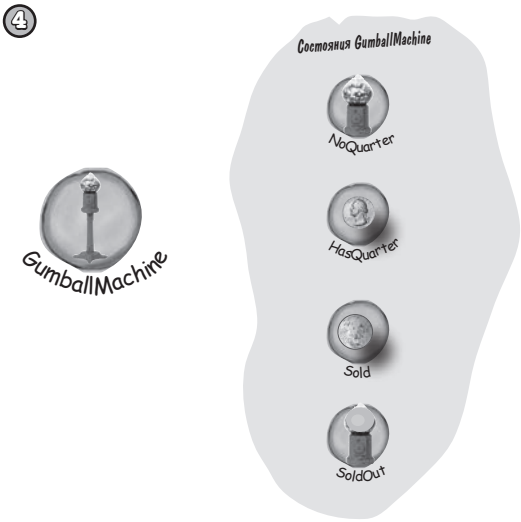
2



3



4



Определение паттерна Состояние

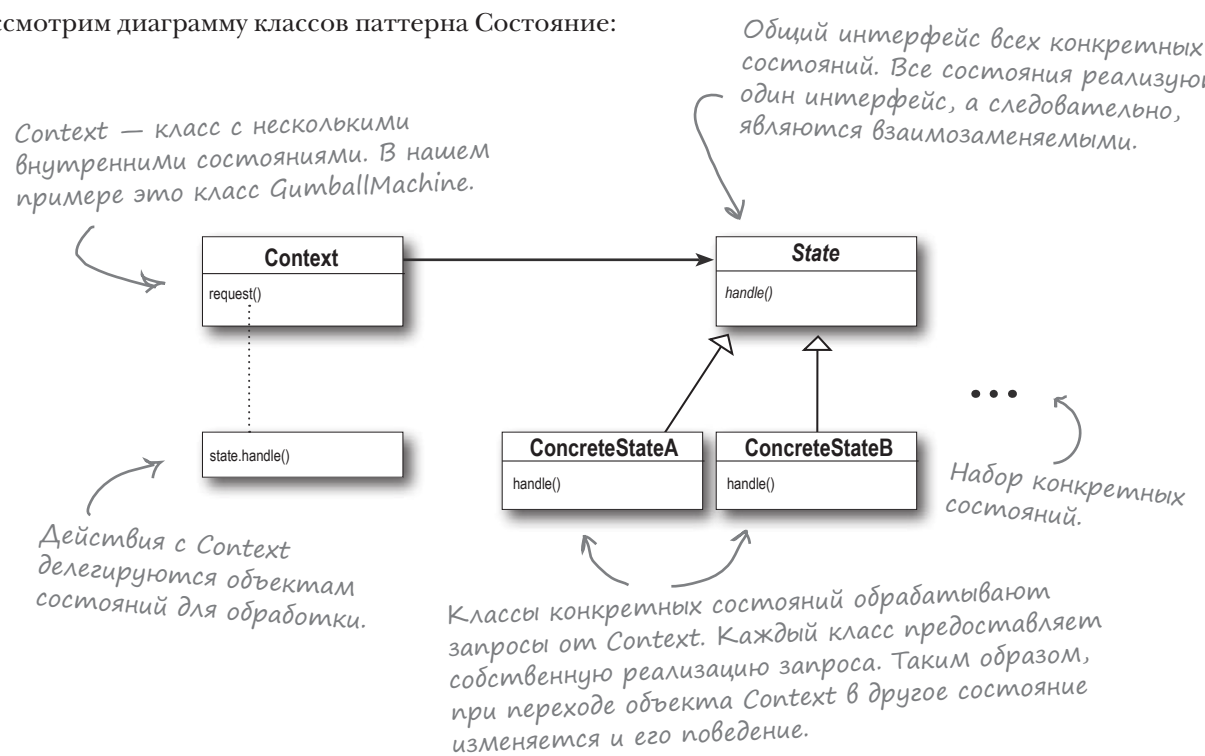
Да, мы успешно реализовали паттерн Состояние! А теперь давайте разберемся с теоретической стороной дела:

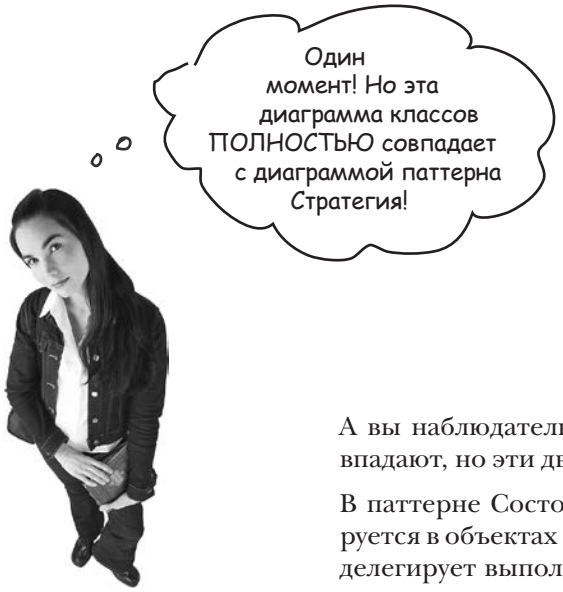
> **Паттерн Состояние** управляет изменением поведения объекта при изменении его внутреннего состояния. Внешне это выглядит так, словно объект меняет свой класс.

Первая часть описания выглядит вполне разумно, не так ли? Поскольку паттерн инкапсулирует состояние в отдельных классах и делегирует операции объекту, представляющему текущее состояние, поведение изменяется вместе с внутренним состоянием. Так, если автомат из нашего примера находится в состоянии NoQuarterState, и в него бросают монетку, его поведение (автомат принимает монетку) будет отличаться от поведения в состоянии HasQuarterState (автомат отвергает монетку).

Но что со второй частью определения? Что значит «словно объект меняет свой класс»? Представьте происходящее с точки зрения клиента: если используемый объект полностью изменяет свое поведение, для клиента это равносильно переходу на работу с объектом другого класса. Конечно, на практике изменение класса имитируется простым переключением на другой объект состояния, задействованный в композиции.

Рассмотрим диаграмму классов паттерна Состояние:





А вы наблюдательны! Да, диаграммы классов практически совпадают, но эти два паттерна различаются своей *целью*.

В паттерне Состояние набор вариантов поведения инкапсулируется в объектах состояния; в любой момент времени контекст делегирует выполнение действий одному из этих объектов. Со временем текущее состояние переключается на другие объекты в соответствии с изменениями внутреннего состояния контекста, так что поведение контекста также изменяется со временем. При этом клиент обычно знает об объектах состояния очень мало (или вообще ничего не знает).

В паттерне Стратегия клиент обычно определяет объект стратегии, связываемый с контекстом. Хотя паттерн обеспечивает необходимую гибкость для изменения объекта стратегии во время выполнения, часто имеется объект стратегии, наиболее подходящий для объекта контекста. Например, в главе 1 некоторые утки инициализировались «типичным» поведением при полете, а для других (резиновых уток и приманок) выбиралось поведение, при котором они оставались на земле.

В общем случае паттерн Стратегия может рассматриваться как гибкая альтернатива субклассированию: если поведение класса определяется наследованием, класс жестко привязывается к этому поведению, даже если позднее его потребуется изменить. Паттерн Стратегия позволяет изменить поведение посредством композиции с другим объектом.

Паттерн Состояние может рассматриваться как замена многочисленных условных конструкций в коде контекста: если поведение инкапсулировано в объектах состояния, для изменения поведения контекста достаточно выбрать другой объект состояния.

Часто Задаваемые Вопросы

В: В классе GumballMachine выбор следующего состояния определяется объектами конкретных состояний. Всегда ли это так?

О: Нет, не всегда. Также следующее состояние может выбираться классом контекста на основании диаграммы переходов.

В общем случае, если переходы между состояниями статичны, их рекомендуется размещать в контексте; если же переходы имеют динамическую природу, их лучше разместить в самом классе состояния (например, переход в состояние NoQuarter или SoldOut в нашем примере зависел от количества шариков, определяемого во время выполнения).

Недостаток определения переходов в классах состояний заключается в том, что оно создает зависимости между классами состояний. В своей реализации GumballMachine мы попытались свести к минимуму такие зависимости за счет использования get-методов вместо жесткого кодирования конкретных классов состояний.

Обратите внимание: принимаемое решение определяет, какие классы будут закрыты для изменений (класс контекста или классы состояний) в ходе эволюции системы.

В: Взаимодействует ли клиент с состояниями?

О: Нет. Состояния используются контекстом для представления его внутреннего состояния и поведения, поэтому все запросы к состояниям поступают от контекста. Клиент не может непосредственно изменить состояние контекста. Контекст обычно сам управляет своим состоянием, а попытки клиента изменить состояние контекста без участия последнего обычно нежелательны.

В: Если в приложении используется несколько экземпляров контекста, могут ли они совместно использовать объекты состояния?

О: Да, вполне. Более того, это обычная ситуация. Единственное требование заключается в том, что объекты состояния не могут обладать собственным внутренним состоянием; в противном случае для каждого контекста необходимо создать отдельный экземпляр.

Обычно в таких ситуациях каждое состояние присваивается статической переменной. Если состояние должно использовать методы или переменные контекста, то ссылка на контекст передается в каждом методе handler().

В: Похоже, использование паттерна Состояние всегда увеличивает количество классов в архитектуре. Посмотрите, сколько новых классов пришлось создать во второй версии GumballMachine!

О: Верно, инкапсуляция поведения в отдельных классах состояний всегда увеличивает количество классов в архитектуре. Это стандартная цена, которую приходится платить за гибкость. Если только вы не пишете «одноразовую» реализацию с очень коротким сроком жизни, увеличьте количество классов, и, скорее всего, вы не пожалеете об этом в долгосрочной перспективе. Причем учтите, что во многих случаях важно количество классов, с которыми имеет дело клиент, а лишние классы всегда можно скрыть от клиента (скажем, объявив их с пакетным уровнем видимости).

В: На диаграмме классов паттерна класс State обозначен как абстрактный. Но ведь в реализации нашего автомата используется интерфейс?

О: Да. Так как у нас нет общей функциональности для размещения в абстрактном классе, мы выбрали интерфейс. В своей реализации вы можете использовать абстрактный класс. В частности, это позволит вам позднее добавить методы в абстрактный класс без нарушения работоспособности конкретных реализаций состояния.

Реализация игры «1 из 10»

Однако работа еще не закончена — нам нужно реализовать игру. Впрочем, в архитектуре на базе паттерна Состояние это делается очень просто. Сначала в класс GumballMachine добавляется новое состояние:

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State winnerState;

    State state = soldOutState;
    int count = 0;

}
   // Методы
}
```

Добавляем переменную для состояния WinnerState и инициализируем ее в конструкторе.

Не забудьте добавить get-метод для состояния WinnerState.

Реализация класса WinnerState очень похожа на реализацию SoldState:

```
public class WinnerState implements State {

    // Переменные экземпляров и конструктор
    // Сообщение об ошибке для insertQuarter
    // Сообщение об ошибке для ejectQuarter
    // Сообщение об ошибке для turnCrank


    public void dispense() {
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() == 0) {
            gumballMachine.setState(gumballMachine.getSoldOutState());
        } else {
            gumballMachine.releaseBall();
            System.out.println("YOU'RE A WINNER! You got two gumballs for your quarter");
            if (gumballMachine.getCount() > 0) {
                gumballMachine.setState(gumballMachine.getNoQuarterState());
            } else {
                System.out.println("Oops, out of gumballs!");
                gumballMachine.setState(gumballMachine.getSoldOutState());
            }
        }
    }
}
```

Как в SoldState.

Выдаем два шарика, после чего переходим либо в состояние NoQuarterState, либо в состояние SoldOutState.

Если в автомате есть второй шарик, освобождаем его.

Завершение реализации игры

Осталось внести последние изменения: реализовать призовую игру и добавить переход в состояние WinnerState. Оба изменения вносятся в состояние HasQuarterState, так как именно в этом состоянии покупатель дергает за рычаг.

```
public class HasQuarterState implements State {
    Random randomWinner = new Random(System.currentTimeMillis());
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }

    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        int winner = randomWinner.nextInt(10);
        if ((winner == 0) && (gumballMachine.getCount() > 1)) {
            gumballMachine.setState(gumballMachine.getWinnerState());
        } else {
            gumballMachine.setState(gumballMachine.getSoldState());
        }
    }
    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

Добавляем генератор случайных чисел с 10%-й вероятностью выигрыша...

...проверяем, повезло ли покупателю.

Если покупателю повезло, и в автомате остался второй шарик, переходим в состояние WinnerState; в противном случае переходим в состояние SoldState (как делалось ранее).

Получилось на удивление просто! Мы включили в GumballMachine новое состояние, а затем реализовали его. В дальнейшем осталось лишь реализовать проверку выигрыша и переход в правильное состояние. Похоже, новая стратегия программирования себя оправдала...

Демо-версия для начальства

На демонстрацию нового кода зашел исполнительный директор фирмы-заказчика. Будем надеяться, что мы ничего не напутали с состояниями! Демо-версия должна быть короткой (всем известно, что руководство Mighty Gumball не любит подолгу задерживаться на одной теме), но при этом достаточно длинной, чтобы пользователь выиграл хотя бы один раз.

Тестовый код почти не изменился; мы только немного сократили его.

```
public class GumballMachineTestDrive {
    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);
    }
}
```

Снова начинаем с пяти шариков.

Чтобы проверить, как работает призовая игра, мы постоянно бросаем монетки и дергаем за рычаг. И каждый раз выводим состояние автомата...

Все программисты ждут результатов тестирования!!!

Что это, везение? В демо-версии мы выиграли не один, а целых два раза!

```
File Edit  Window  Help  Whenisagumballajawbreaker?
%java GumballMachineTestDrive
Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...
YOU'RE A WINNER! You get two gumballs for your quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 3 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot...
You inserted a quarter
You turned...
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...
YOU'RE A WINNER! You get two gumballs for your quarter
Oops, out of gumballs!

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 0 gumballs
Machine is sold out
%
```

Часто Задаваемые Вопросы

В: Зачем создавать отдельное состояние WinnerState? Почему бы не выдать два шарика в состоянии SoldState?

О: Отличный вопрос. Состояния SoldState и WinnerState почти полностью идентичны, только WinnerState выдает два шарика вместо одного. Конечно, код выдачи двух шариков можно разместить и в SoldState. Недостаток такого решения заключается в том, что класс начинает представлять сразу ДВА состояния. Таким образом, за устранение дублирования приходится платить ясностью кода. Также стоит вспомнить о принципе, представленном в предыдущей главе: «Один класс — одна обязанность». Включая обязанности WinnerState в SoldState, вы наделяете SoldState ДВУМЯ обязанностями. А что произойдет, если рекламная акция завершится? Или изменится приз? Таким образом, речь идет о компромиссном решении, принятом на архитектурном уровне.

Проверка разумности

Да, директору Mighty Gumball проверка разумности определенно не помешает, но сейчас мы говорим не об этом. Давайте припомним некоторые аспекты GumballMachine, которые не мешало бы доработать перед выдачей окончательной версии:

- Состояния SoldState и WinnerState содержат большой объем дублирующегося кода. Как избавиться от дублирования? Можно сделать State абстрактным классом и встроить в эти методы некое поведение по умолчанию; в конце концов, сообщения об ошибках не должны быть видны клиенту. Таким образом, обобщенное поведение обработки ошибок может наследоваться от абстрактного суперкласса.

Я все-таки торговый автомат, а не компьютер!

- Метод dispense() вызывается всегда, даже если покупатель дернул за рычаг, не бросив монетки. Хотя автомат выдает шарик только в правильном состоянии, эта проблема легко решается возвращением логического флага из метода turnCrank() или обработкой исключений. Как вы думаете, какое решение лучше?

- Вся информация о переходах между состояниями хранится в классах состояний. Какие проблемы могут из-за этого возникнуть? Не переместить ли эту логику в Gumball Machine? Какими достоинствами и недостатками будет обладать такое решение?

- Если вы планируете создавать большое количество объектов GumballMachine, возможно, экземпляры состояний стоит переместить в статические переменные для совместного использования. Какие изменения придется внести в GumballMachine и классы состояний?

Беседа у камина

Воссоединение паттернов Стратегия и Состояние

Стратегия

Здорово, брат. Слыхал, про меня написали в главе 1?

А я тут просто помогал Шаблонному Методу закончить его главу. Как дела, чем занимаешься?

Право, не знаю... Мне всегда кажется, что ты просто копируешь то, что делаю я, но описываешь другими словами. Только подумай: я помогаю объектам внедрять разные варианты поведения или алгоритмы посредством композиции и делегирования. По сути, ты делаешь то же самое.

Да ну? Это как? Что-то я тебя не пойму.

Да, это была *классная* работа... И ты наверняка видишь, что это более мощный механизм, чем наследование поведения?

Извини, придется объяснить подробнее.

Состояние

Да, мне уже рассказывали.

Да как обычно — помогаю классам проявлять разное поведение в зависимости от состояния.

Признаю, наши задачи определенно имеют немало общего, но мои намерения полностью отличаются от твоих. И мои клиенты учатся применять композицию и делегирование совершенно иным способом.

Если потратишь немного времени на размышления о чем-то, кроме себя самого — то поймешь. Подумай, как ты работаешь: есть класс, экземпляр которого ты создаешь; ты обычно передаешь ему объект стратегии, реализующий некоторое поведение. Помнишь, в главе 1 мы реализовали разные варианты кряканья? Настоящие утки крякали, а резиновые пищали.

Да, конечно. А теперь подумай, как работаю я; ничего похожего.

Стратегия

Что тут такого? Я тоже могу изменять поведение во время выполнения; для этого и нужна композиция!

Признаю, я не заставляю свои объекты иметь четко определенный набор переходов между состояниями. Обычно я предпочитаю управлять тем, какую стратегию используют мои объекты.

Да, да, мечтай, братец. Ты изображаешь из себя крутой паттерн вроде меня, но не забудь: меня описали в главе 1, а тебе нашлось место только в главе 10. Я хочу сказать — сколько людей дочитает книгу до этого места?

Узнаю своего брата. Мечтает, как всегда...

Состояние

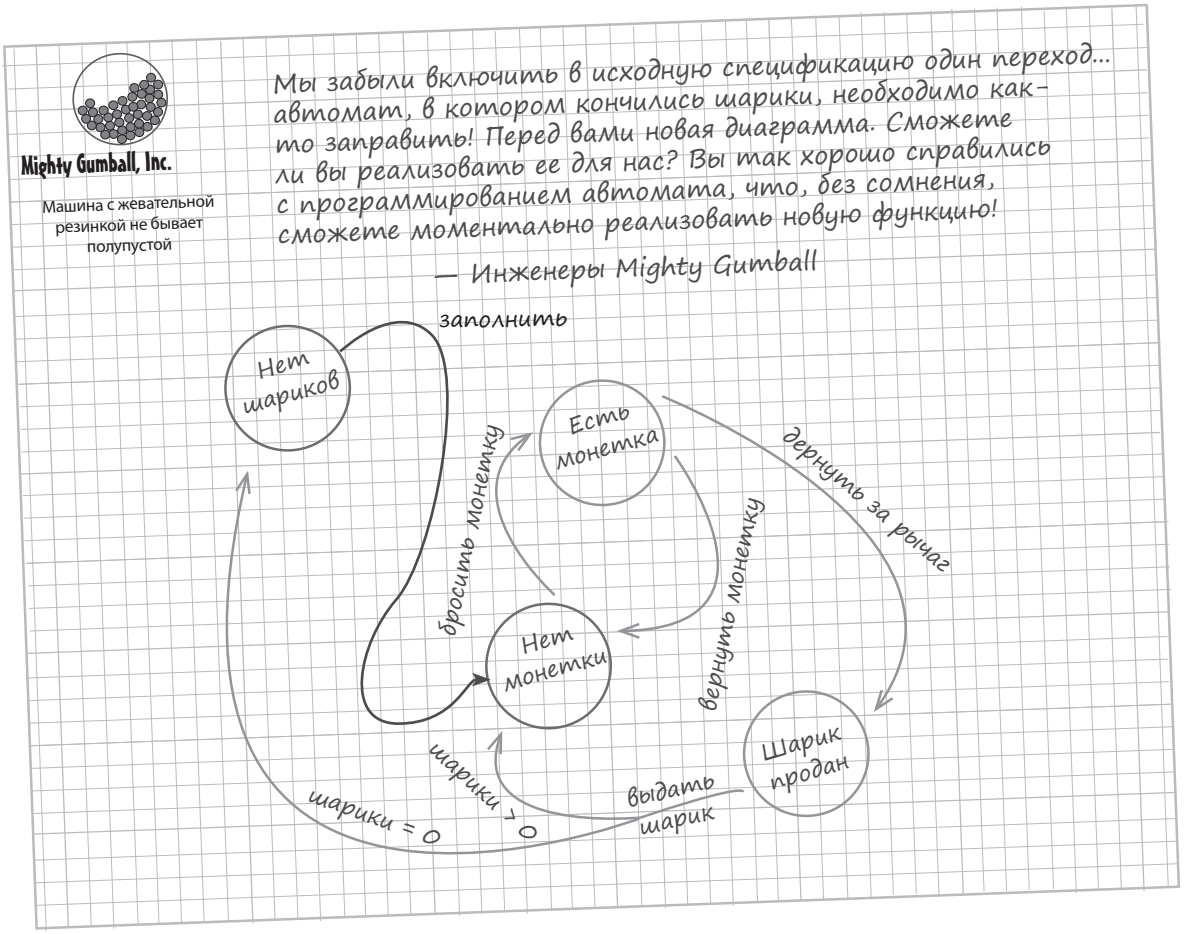
При создании объектов контекста я могу задать их исходное состояние, но со временем они могут сами изменять свое состояние.

Можешь, конечно, но моя работа построена на дискретных состояниях; мои объекты контекста изменяют состояние со временем в соответствии с четко определенными переходами. Иначе говоря, изменение поведения встроено в мою схему — я так работаю!

Послушай, мы уже договорились, что у нас много общего в структуре, но задачи, решаемые с нашей помощью, заметно различаются. В мире найдется работа для нас обоих.

Шутишь? Это крутая книга для крутых читателей. Конечно, они дочитают до главы 10!

Чуть не забыли!



Возьми в руку карандаш

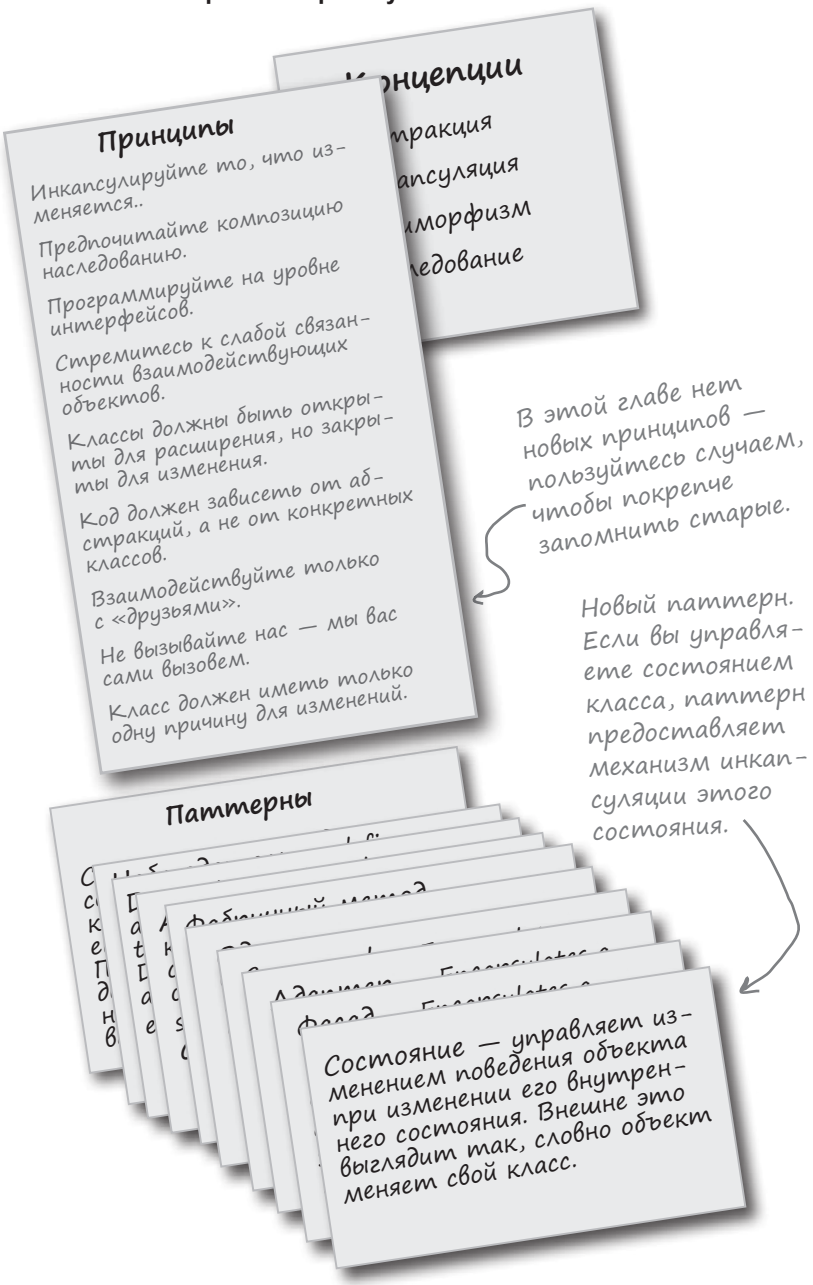
Напишите метод refill() для заправки автомата. Метод получает один аргумент (количество шариков, загружаемых в автомат), обновляет переменную-счетчик шариков и инициализирует текущее состояние автомата.

Соедините каждый паттерн с его описанием:

Паттерн	Описание
Состояние	Инкапсулирует взаимозаменяемые варианты поведения и выбирает один из них посредством делегирования
Стратегия	Субклассы выбирают реализацию шагов алгоритма
Шаблонный Метод	Инкапсулирует поведение, связанное с состоянием, с делегированием поведения объекту текущего состояния

Новые инструменты

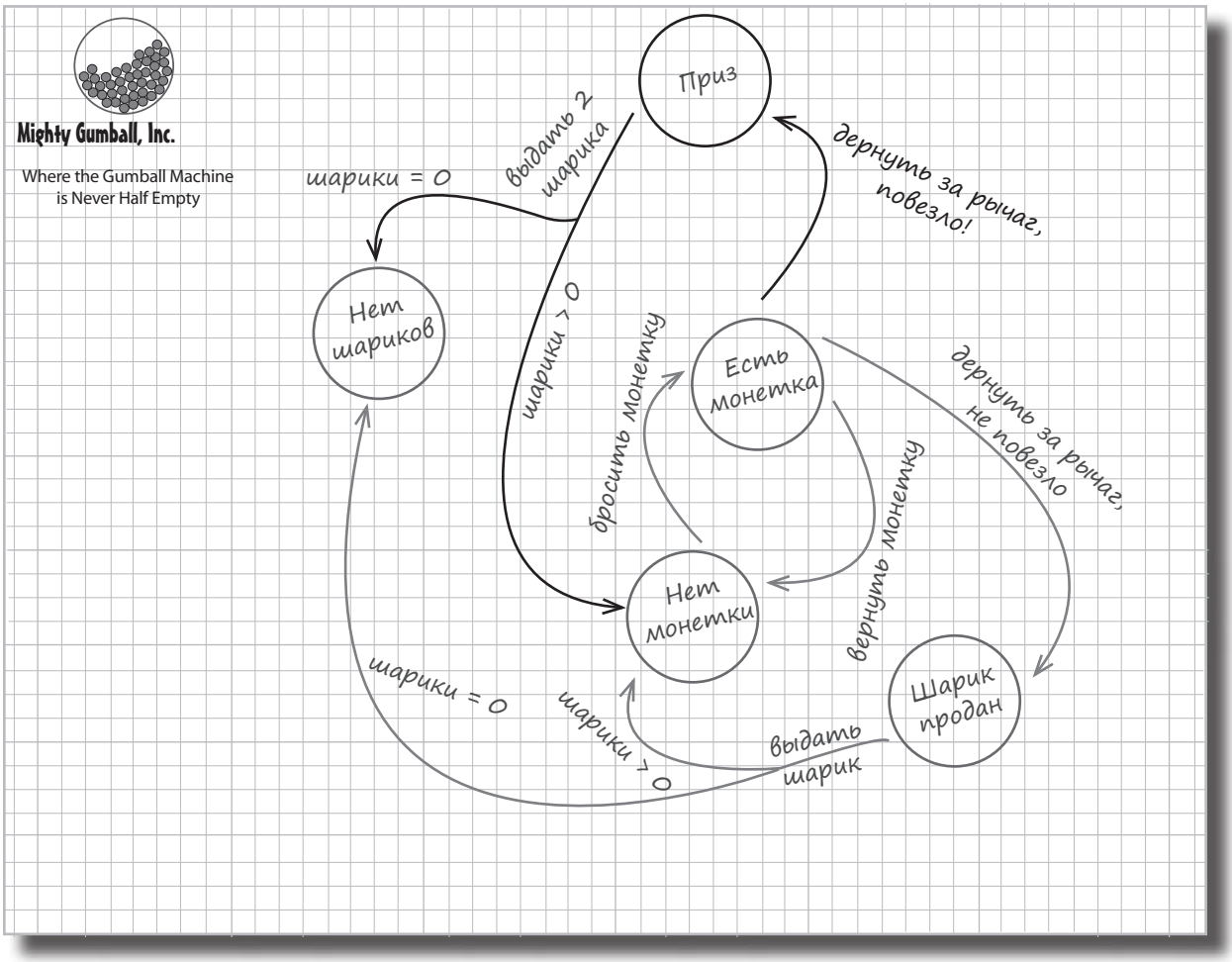
Подошла к концу очередная глава. Известных вам паттернов вполне достаточно для того, чтобы с легкостью пройти любое собеседование при приеме на работу!



КЛЮЧЕВЫЕ МОМЕНТЫ

- Паттерн Состояние позволяет объекту иметь много разных вариантов поведения в зависимости от его внутреннего состояния.
- В отличие от процедурных конечных автоматов, состояние в этом паттерне представляется полноценным классом.
- Поведение контекста реализуется делегированием выполняемых операций текущему объекту состояния, с которым он связан посредством композиции.
- Инкапсуляция состояния в классе локализует его возможные изменения.
- Паттерны Состояние и Стратегия имеют похожие диаграммы классов, но решают разные задачи.
- Паттерн Стратегия обычно определяет в классе контекста поведение алгоритма.
- Паттерн Состояние изменяет поведение контекста в соответствии с изменениями его состояния.
- Переходами между состояниями могут управлять как классы состояний, так и классы контекстов.
- Применение паттерна Состояние обычно увеличивает количество классов в архитектуре.
- Классы состояний могут совместно использоваться несколькими экземплярами контекстов.

Ответы к упражнениям



Возьми в руку карандаш
Решение

Какие из следующих утверждений относятся к нашей реализации? (Выберите все подходящие утверждения.)

- ☑ A. Код не соответствует принципу открытости/закрытости.
- ☑ B. Такой стиль программирования характерен для FORTRAN.
- ☑ C. Архитектуру трудно назвать объектно-ориентированной.
- ☑ C. Переходы между состояниями не очевидны; они «прячутся» в множестве условных конструкций.
- ☑ D. Переменные аспекты этой архитектуры не инкапсулированы.
- ☑ E. Дальнейшие изменения с большой вероятностью приведут к ошибкам в готовом коде.

Возьми в руку карандаш
Решение

Остался всего один нереализованный класс состояния: SoldOutState. Почему бы вам не реализовать его? Тщательно продумайте поведение автомата в каждой ситуации. Проверьте ответ, прежде чем двигаться дальше...

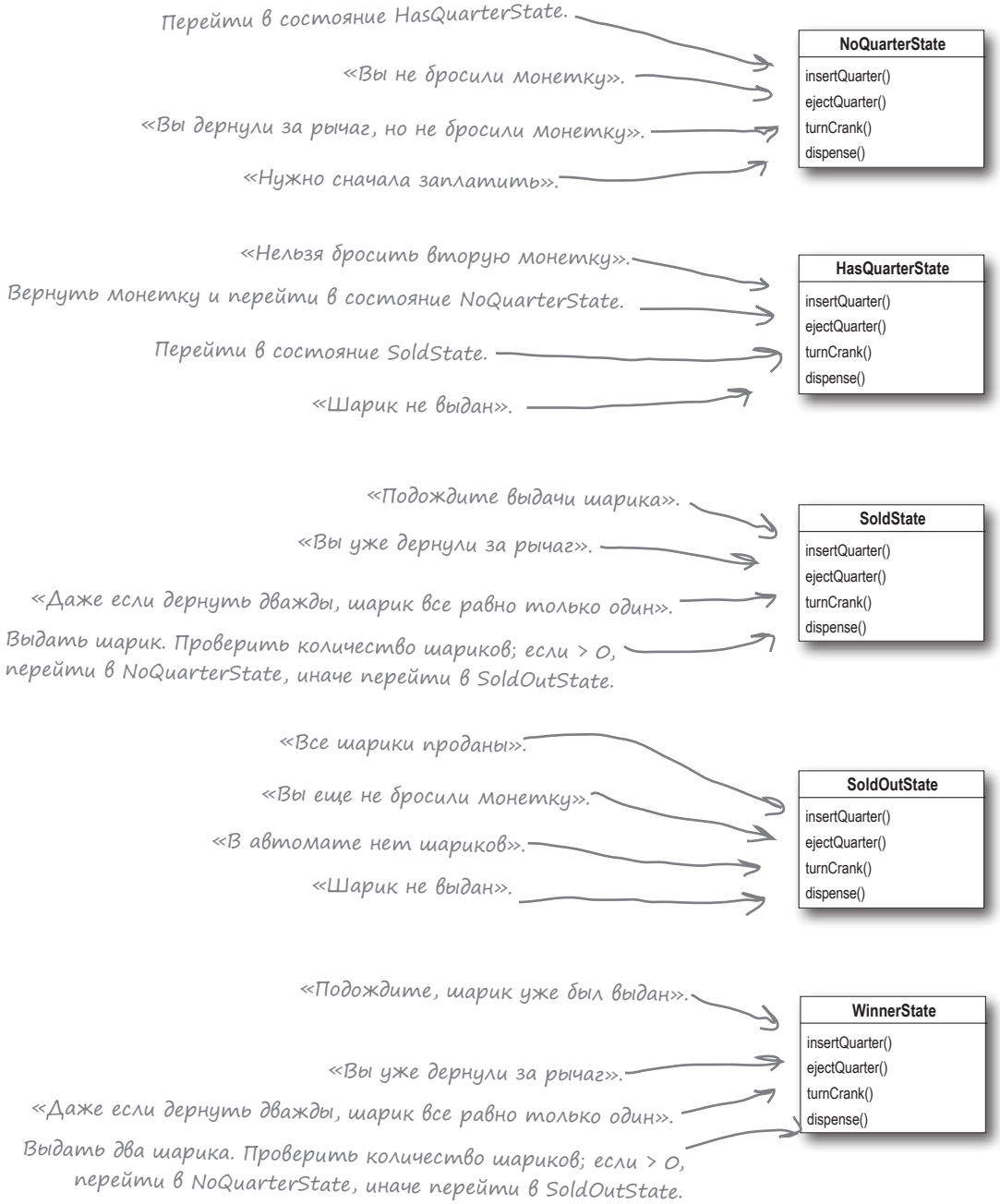
В состоянии SoldOutState никакие действия невозможны, пока кто-то не заправит автомат шариками.

```
public class SoldOutState implements State {
    GumballMachine gumballMachine;

    public SoldOutState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert a quarter, the machine is sold out");
    }

    public void ejectQuarter() {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }

    public void turnCrank() {
        System.out.println("You turned, but there are no gumballs");
    }

    public void dispense() {
        System.out.println("No gumball dispensed");
    }

    public String toString() {
        return "sold out";
    }
}
```

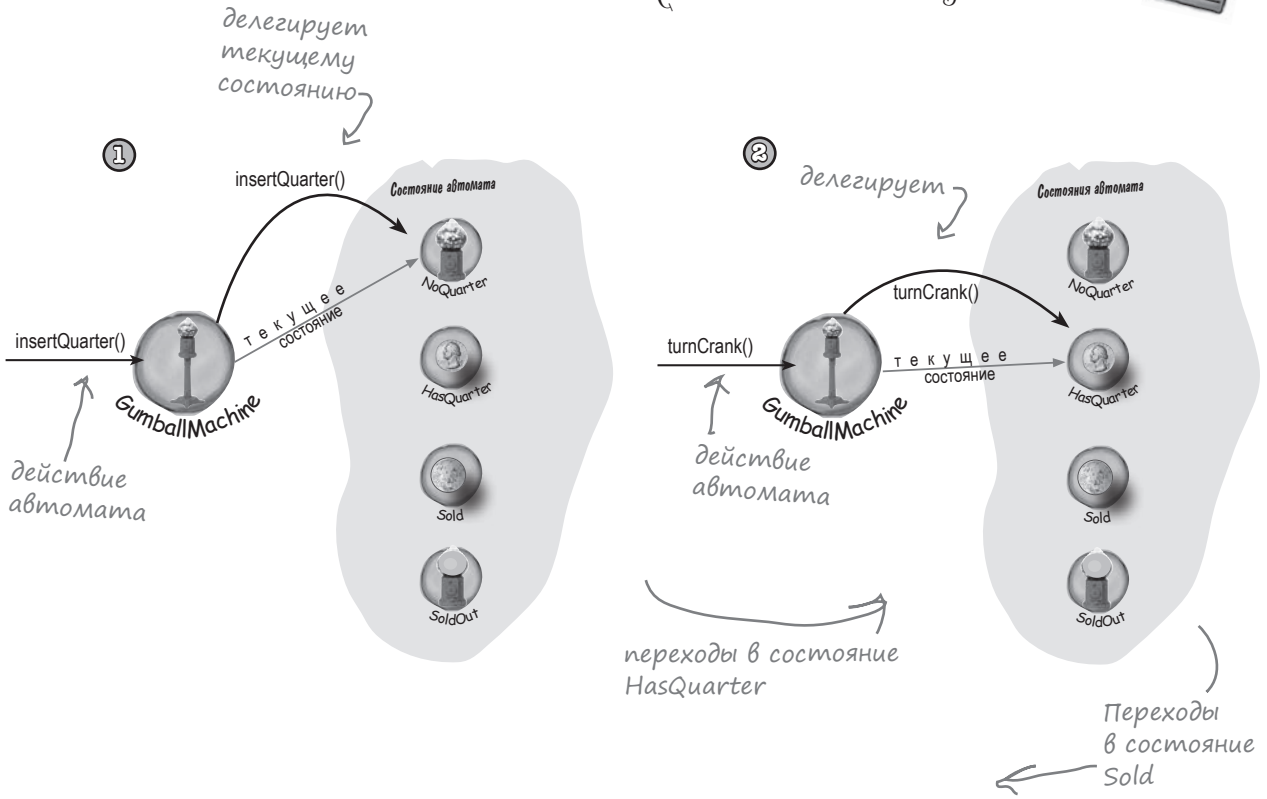
Возьми в руку карандаш

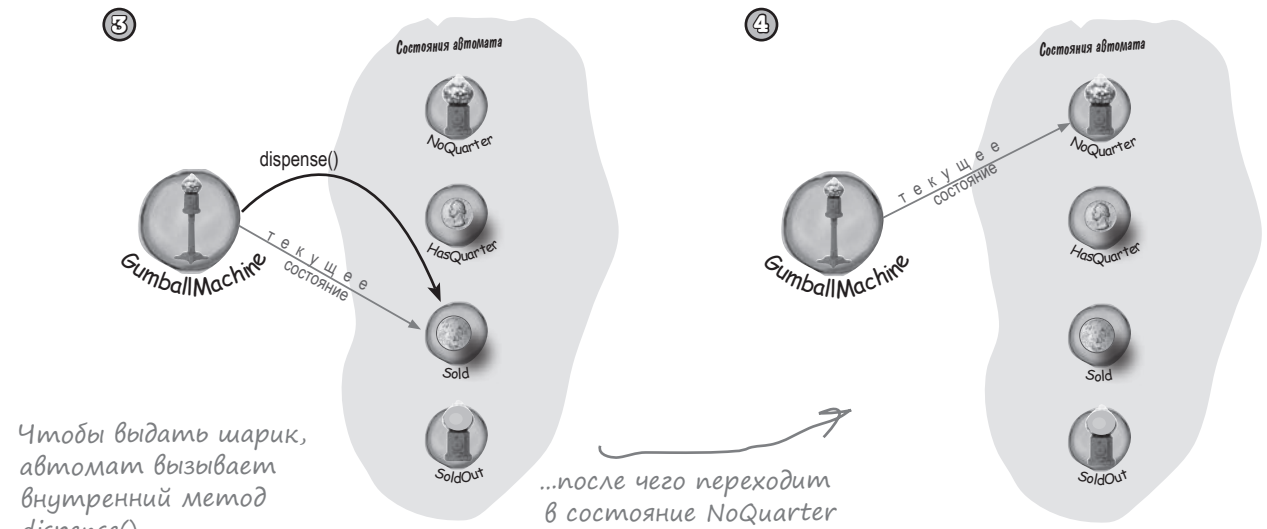
Решение

Реализация состояний начинается с определения поведения классов при выполнении каждого действия. Заполните следующую диаграмму описаниями поведения каждого из классов; мы уже заполнили некоторые описания за вас.



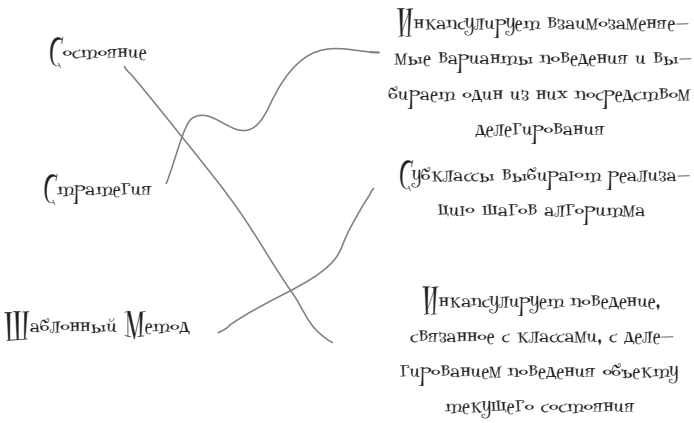
За сценой: Самостоятельное путешествие





Кто и что делает? Решение

Соедините каждый паттерн с его описанием:



Возьми в руку карандаш Решение

Для заполнения автомата мы добавим в интерфейс State метод refill(), который должен быть реализован каждым классом, реализующим State. В любом состоянии, кроме SoldOutState, метод не делает ничего. В состоянии SoldOutState refill() переходит в состояние NoQuarterState. Мы также добавим в GumballMachine метод refill(), который увеличивает счетчик шариков, а затем вызывает метод refill() текущего состояния.

Этот метод добавляется в SoldOutState.

```
public void refill() {
    gumballMachine.setState(gumballMachine.getNoQuarterState());
}
```

А этот метод добавляется в GumballMachine.

```
void refill(int count) {
    this.count += count;
    System.out.println("The gumball machine was just refilled; it's new count is: " + this.count);
    state.refill();
}
```

Управление доступом к объектам

Когда-нибудь разыгрывали сценку «хороший полицейский, плохой полицейский»? Вы — «хороший полицейский», вы общаетесь со всеми любезно и по-дружески, но не хотите, чтобы все обращались к вам за каждым пустяком. Поэтому вы обзаводитесь «плохим полицейским», который управляет доступом к вам. Именно этим и занимаются заместители: они управляют доступом. Как вы вскоре увидите, существует множество способов взаимодействия заместителей с обслуживаемыми объектами. Иногда заместители пересылают по Интернету целые вызовы методов, а иногда просто терпеливо стоят на месте, изображая временно отсутствующие объекты.

Еще не забыли директора Mighty Gumball?

Вроде бы задача не из сложных. Если помните, в коде автомата уже имеются методы для получения количества шариков (getCount()) и текущего состояния автомата (getState()).

Остается лишь построить отчет и отослать его директору. И вероятно, для каждого автомата стоит добавить поле с описанием его местонахождения, чтобы было понятно, к какому автомату относится отчет.

Можно браться за программирование. Моментальное исполнение задачи произведет впечатление на заказчика.

Программирование монитора

Начнем с добавления поля местонахождения в класс GumballMachine:

```
public class GumballMachine {
    // Другие переменные
    String location;

    public GumballMachine(String location, int count) {
        // Код конструктора
        this.location = location;
    }

    public String getLocation() {
        return location;
    }

    // Другие методы
}
```

Обычная переменная типа String.

Местонахождение передается конструктору и сохраняется в переменной экземпляра.

Также добавляем get-метод для получения строки местонахождения.

Новый класс GumballMonitor получает местонахождение автомата, количество оставшихся шариков и текущее состояние и выводит данные в виде небольшого аккуратного отчета:

```
public class GumballMonitor {
    GumballMachine machine;

    public GumballMonitor(GumballMachine machine) {
        this.machine = machine;
    }

    public void report() {
        System.out.println("Gumball Machine: " + machine.getLocation());
        System.out.println("Current inventory: " + machine.getCount() + " gumballs");
        System.out.println("Current state: " + machine.getState());
    }
}
```

Конструктор получает объект автомата и сохраняет его в переменной экземпляра machine.

Метод report() выводит отчет с информацией о местонахождении, количестве шариков и состоянии машины.

Тестирование монитора

Задача решена за считанные минуты. Заказчик будет поражен нашим мастерством разработки.

Теперь необходимо создать экземпляр GumballMonitor и задать автомат для наблюдения:

```
public class GumballMachineTestDrive {
    public static void main(String[] args) {
        int count = 0;

        if (args.length < 2) {
            System.out.println("GumballMachine <name> <inventory>");
            System.exit(1);
        }

        count = Integer.parseInt(args[1]);
        GumballMachine gumballMachine = new GumballMachine(args[0], count);

        GumballMonitor monitor = new GumballMonitor(gumballMachine);


        // rest of test code here

        monitor.report();
    }
}
```

Местонахождение и исходное количество шариков передаются в командной строке.

Передаем параметры конструктору...

Создание монитора с указанием автомата, для которого строится отчет.

Чтобы получить отчет для автомата, вызываем метод report().

```
File  Edit  Window  Help  FlyingFish
%java GumballMachineTestDrive Seattle 112

Gumball Machine: Seattle
Current Inventory: 112 gumballs
Current State: waiting for quarter
```

А вот и результат!

Классный отчет... Но, видимо, я недостаточно ясно объяснил задачу. Данные об автоматах должны передаваться ПО СЕТИ! Все коммуникации уже проложены. Не забывайте, что мы живем в эпоху Интернета!

Фрэнк: Удаленный кто?

Джо: *Удаленный заместитель*. Подумайте: код монитора уже написан, верно? Мы передаем объекту GumballMonitor ссылку на автомат, а он выдает нам отчет. Проблема в том, что монитор работает в одной JVM-машине с объектом автомата, а заказчик хочет сидеть за столом и получать данные об автоматах в *удаленном* режиме! А если мы оставим класс GumballMonitor в прежнем виде, но передадим ему заместителя *удаленного* объекта?

Фрэнк: Что-то я не до конца понял.

Джим: Я тоже.

Джо: Начнем с начала... Заместитель — суррогат *настоящего* объекта. В данном случае заместитель ведет себя, как объект GumballMachine, а «за кулисами» взаимодействует по сети с настоящим удаленным объектом GumballMachine.

Джим: То есть наш код остается без изменений, а монитору мы передаем ссылку на заместителя GumballMachine...

Фрэнк: И этот заместитель прикидывается настоящим объектом, а на самом деле он просто взаимодействует с ним по сети.

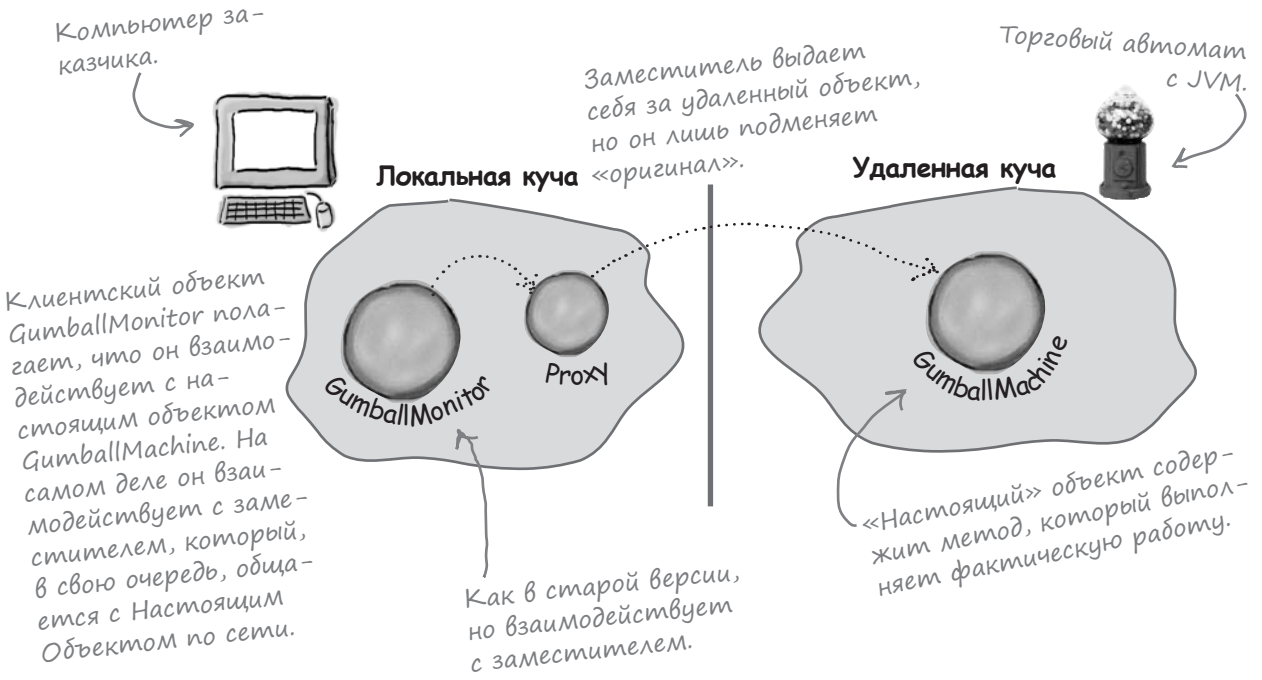
Джо: Да, примерно так.

Фрэнк: Однако сказать легче, чем сделать.

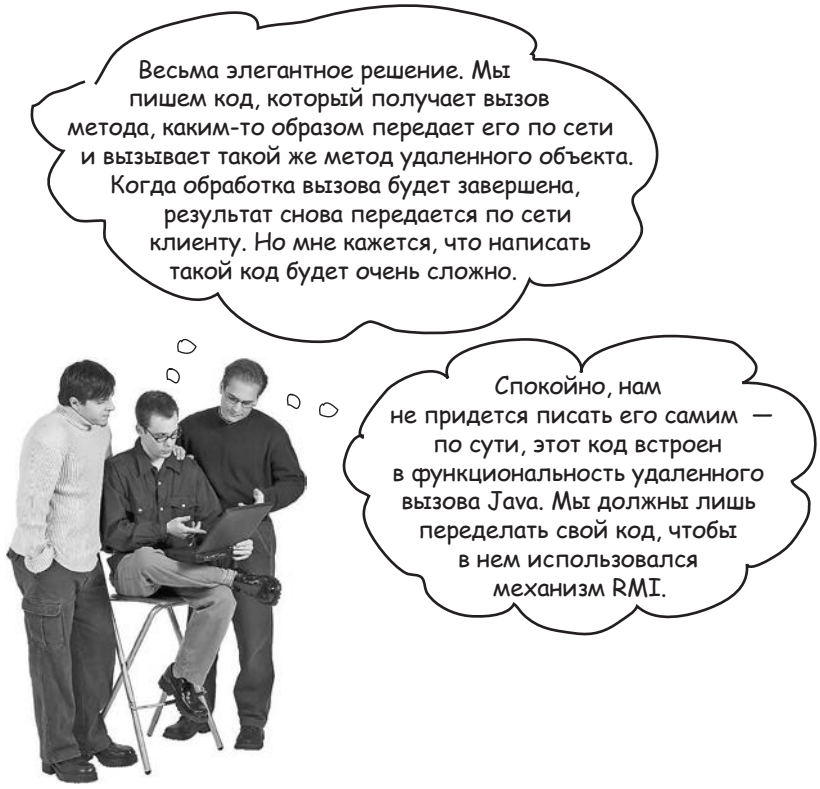
Джо: Возможно, но вряд ли у нас возникнут сложности. Нужно позаботиться о том, чтобы автомат работал как сетевая служба и принимал запросы по сети; также необходимо реализовать для монитора механизм получения ссылки на объект-заместитель. К счастью, в Java уже имеются отличные встроенные средства для решения подобных задач. Но сначала давайте еще немного поговорим об удаленных заместителях...

Роль «удаленного заместителя»

Удаленный заместитель действует как *локальный представитель удаленного объекта*. Что такое «удаленный объект»? Это объект, находящийся в куче другой виртуальной машины Java (или в более общем значении — удаленный объект, выполняемый в другом адресном пространстве). Что такое «локальный представитель»? Это объект, вызовы локальных методов которого перенаправляются удаленному объекту.



Клиентский объект работает так, словно он вызывает методы удаленного объекта. Но в действительности он вызывает методы объекта-заместителя, существующего в локальной куче, который берет на себя все низкоуровневые подробности сетевых взаимодействий.



МОЗГОВОЙ ШТУРМ

Прежде чем двигаться дальше, подумайте, как бы вы спроектировали систему поддержки вызова удаленных методов. Как упростить задачу для разработчика, чтобы ему пришлось писать по возможности небольшой объем кода? Как обеспечить органичную интеграцию удаленных вызовов?

МОЗГОВОЙ ШТУРМ 2

Должны ли удаленные вызовы быть полностью прозрачными для клиента? Какие проблемы могут возникнуть при таком подходе?

Включение удаленного заместителя в код мониторинга GumballMachine

На бумаге все смотрится хорошо, но как создать заместителя, который умеет вызывать методы объекта из другого экземпляра JVM?

Хмм... Ведь мы же не можем получить ссылку на объект из другой кучи, верно? Иначе говоря, нельзя использовать конструкции вида

```
Duck d = <объект из другой кучи>
```

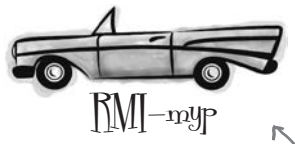
Объект, на который ссылается переменная d, должен находиться в пространстве той же кучи, что и код, выполняющий команду. Как подойти к решению этой задачи? На помощь приходит механизм Java RMI (Remote Method Invocation). Он позволяет получать доступ к объектам в удаленных JVM и вызывать их методы.

Если вы еще не сталкивались с RMI, небольшое введение поможет вам получить представление об этой теме до включения поддержки заместителей в код GumballMachine. Итак, вот что мы собираемся сделать:

1. **Сначала мы познакомимся с механизмом RMI в общих чертах. Даже если вы уже разбираетесь в этой теме, будет полезно освежить информацию в памяти.**

2. **Затем код GumballMachine будет преобразован в удаленную службу с набором методов, вызываемых по сети.**

3. **После этого мы создадим заместителя, который взаимодействует с удаленным объектом GumballMachine средствами RMI, и построим новый вариант системы сбора данных, чтобы заказчик мог наблюдать за любым количеством торговых автоматов.**



Если вы еще не работали с RMI, краткое описание на нескольких страницах введет вас в курс дела. А если работали — хотя бы бегло пролистайте их и вспомните основные моменты.

Удаленные вызовы методов

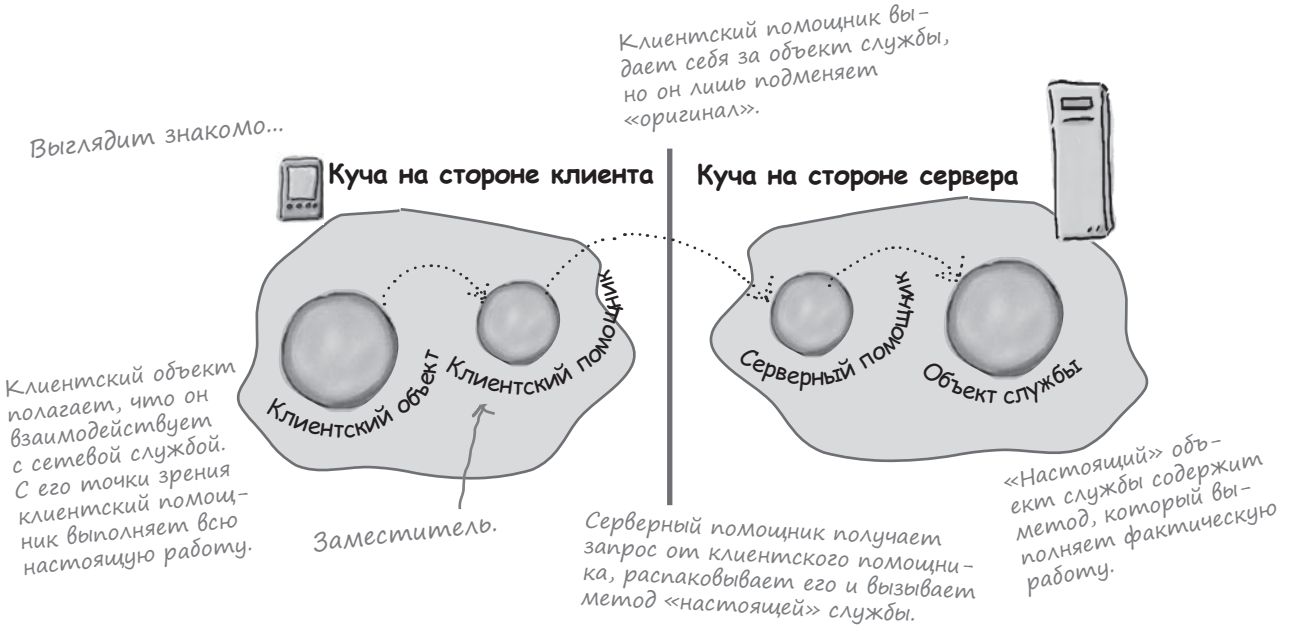
Допустим, вы проектируете систему, в которой обращенные к локальному объекту вызовы перенаправляются удаленному объекту. Как бы вы подошли к ее проектированию? Вероятно, будет создана пара вспомогательных объектов, которые будут выполнять обмен данными за вас. Эти вспомогательные объекты позволят клиенту действовать так, словно он вызывает метод не удаленного, а локального объекта (что, собственно, и происходит), а локальный объект перенаправляет запрос реальному поставщику сервиса.

Иначе говоря, клиентский объект считает, что он вызывает метод удаленной службы, потому что вспомогательный объект (далее для краткости — помощник) изображает объект сетевой службы. Изображает объект с методом, который требуется вызвать клиенту.

Однако клиентский помощник не является удаленной службой. Он предоставляет метод, но не содержит фактической логики, необходимой клиенту. Вместо этого помощник связывается с сервером, передает информацию о вызове (имя метода, аргументы и т. д.) и ждет ответа от сервера.

На стороне сервера другой помощник получает запрос от клиентского объекта (через сокет), распаковывает информацию о вызове и вызывает реальный метод реального объекта службы. Таким образом, для объекта службы вызов является локальным. Он поступает от помощника на стороне сервера, а не от удаленного клиента.

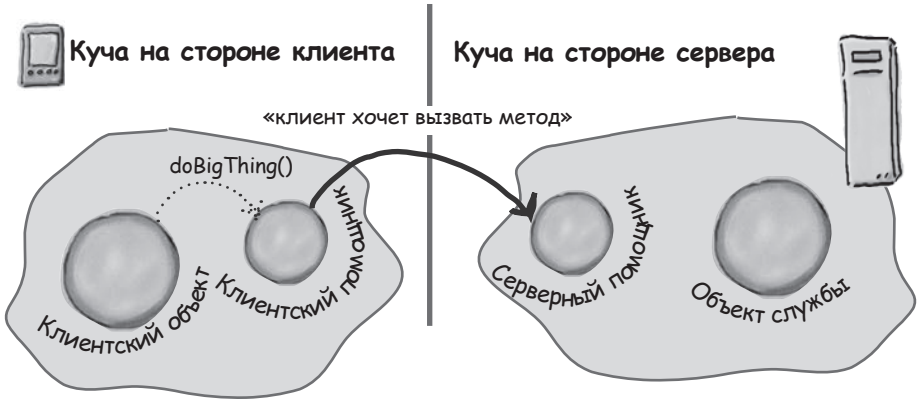
Серверный помощник получает возвращаемое значение от службы, упаковывает его и передает обратно (через выходной поток сокета) клиентскому помощнику. Последний распаковывает информацию и возвращает значение клиентскому объекту.



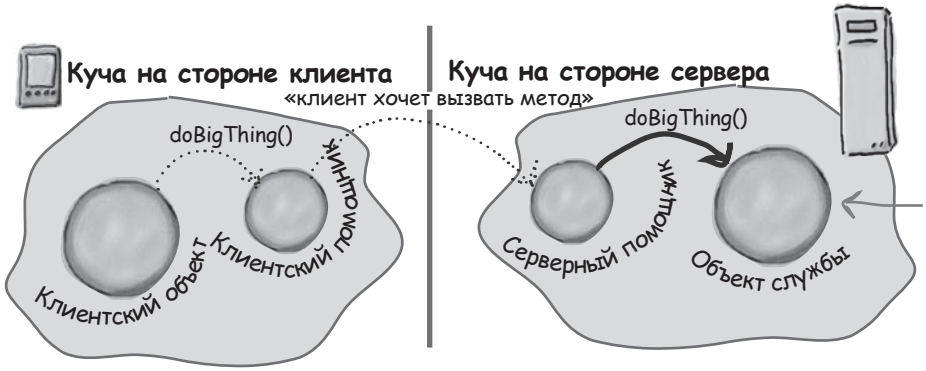
Как происходит вызов метода

① Клиентский объект вызывает метод doBigThing() клиентского помощника.

② Клиентский помощник упаковывает информацию о вызове (аргументы, имя метода и т. д.) и передает ее по сети серверному помощнику.



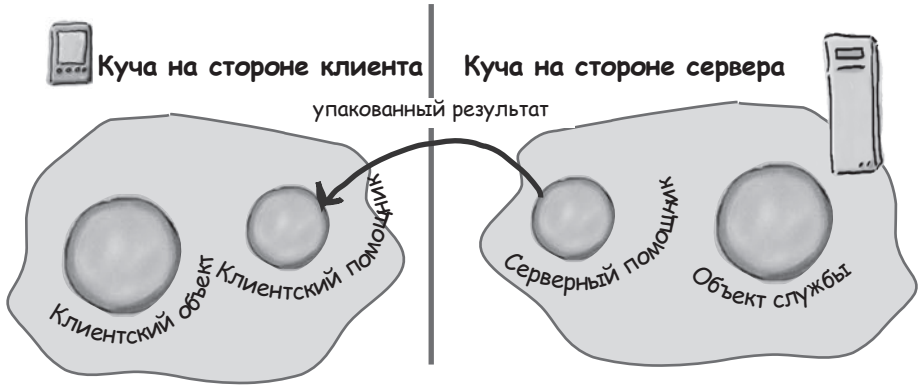
③ Серверный помощник распаковывает полученную информацию, определяет, какой метод (и какого объекта) следует вызвать, и вызывает настоящий метод настоящего объекта службы.



Напоминаем: это объект с ФАКТИЧЕСКОЙ логикой метода! Тот, который выполняет реальную работу!

4 Вызванный метод объекта службы возвращает результат серверному помощнику.

5 Серверный помощник упаковывает информацию, полученную в результате вызова, и возвращает ее по сети вспомогательному объекту на стороне клиента.



6 Клиентский помощник распаковывает данные и возвращает их клиентскому объекту. Все происходящее полностью прозрачно с точки зрения клиентского объекта.

Java RMI: общая картина

Итак, вы примерно представляете, как вызываются удаленные методы; теперь нужно разобраться, как использовать RMI для вызова удаленных методов.

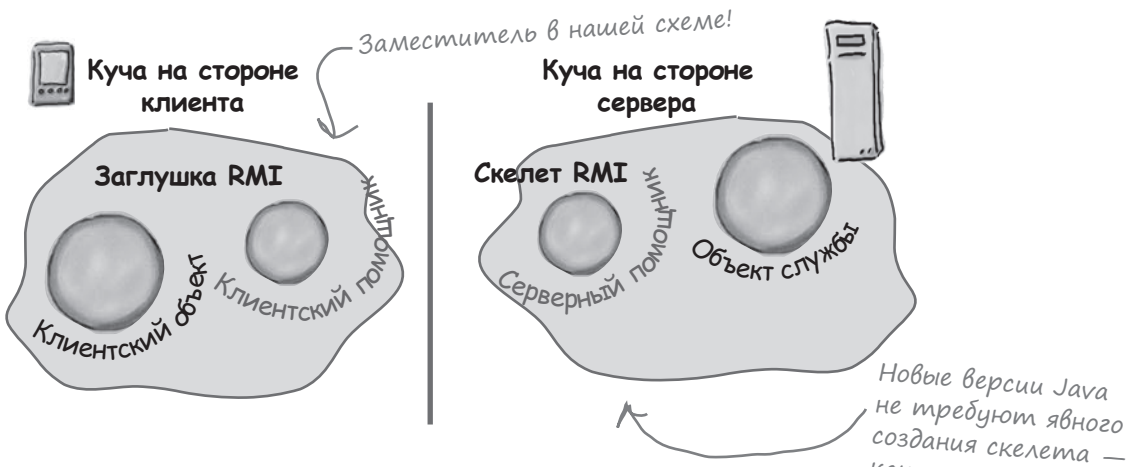
RMI генерирует помощников на стороне клиента и сервера — вплоть до создания клиентского помощника с таким же набором методов, как у удаленной службы. При этом вам не приходится писать код передачи данных по сети или ввода данных самостоятельно. Вы просто вызываете удаленные методы точно так же, как вызываете обычные методы объектов, выполняемых в локальной JVM клиента.

RMI также предоставляет всю инфраструктуру, обеспечивающую функционирование этого механизма (включая сервис поиска и обращения к удаленным объектам).

Между вызовами RMI и обычными (локальными) вызовами существует одно важное различие. Хотя с точки зрения клиента вызов метода ничем не отличается от локального, клиентский помощник передает вызов метода по сети. В передаче данных задействована передача данных по сети и операции ввода/вывода. А что мы о них знаем?

Они небезопасны! Они могут завершиться неудачей! Они могут генерировать исключения. В результате клиент должен быть готов к неожиданностям. Через пару страниц вы увидите, в чем именно заключается подготовка.

В терминологии RMI клиентский помощник называется «заглушкой» (stub), а серверный помощник — «скелетом» (skeleton).



А теперь давайте рассмотрим все действия, которые необходимы для превращения объекта в службу, принимающую удаленные вызовы (а также для выполнения удаленных вызовов на стороне клиента).

Пристегните ремни. Впереди нас ждут выбоины и крутые повороты, но ничего невозможного.

Создание удаленной службы

Далее приводится **краткий обзор** пяти шагов построения удаленной службы – другими словами, шагов, необходимых для того, чтобы взять обычный объект и наделить его способностью обслуживать вызовы удаленных клиентов. Позднее мы сделаем это с классом GumballMachine, а пока рассмотрим общую последовательность действий (подробные объяснения будут приведены позднее).

Шаг 1

Создание **интерфейса удаленного доступа**. Интерфейс удаленного доступа определяет методы, вызываемые клиентом в удаленном режиме. Именно он будет указываться клиентом в качестве типа вашей службы. Реализуется как заглушкой, так и самой службой!

MyService.java

Интерфейс определяет удаленные методы, которые будут вызываться клиентами.

Шаг 2

Создание **реализации интерфейса удаленного доступа**. Речь идет о классе, выполняющем Настоящую Работу. Он определяет фактическую реализацию методов, определенных в интерфейсе удаленного доступа, и это его методы вызываются клиентом.

MyServiceImpl.java

«Настоящая» служба; класс с методами, выполняющими фактическую работу. Реализует интерфейс удаленного доступа.

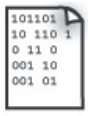
Шаг 3

Запуск **rmiregistry**. Клиент обращается к реестру RMI (*rmiregistry*) для получения заместителя (клиентской заглушки/объекта-помощника).

Запускается в отдельном терминале.

Шаг 4

Запуск **удаленной службы**. Наконец, необходимо запустить объект удаленной службы. Класс реализации службы создает экземпляр службы и регистрирует его в реестре RMI. Зарегистрированная служба становится доступной для клиентов.



Stub

Skeleton

Заглушка и Заготовка динамически генерируются за вас.

Шаг 1: создание интерфейса удаленного доступа

① Расширение java.rmi.Remote

Интерфейс Remote не содержит методов. Тем не менее он имеет особый смысл для RMI, поэтому это правило необходимо выполнить. Обратите внимание на термин «расширение» — один интерфейс может *расширять* другой интерфейс.

```
public interface MyRemote extends Remote {
```

Означает, что интерфейс будет использоваться для поддержки удаленных вызовов.

② Объявление возможного исключения RemoteException

Интерфейс удаленного доступа используется клиентом в качестве типа службы. Иначе говоря, клиент вызывает методы чего-то, реализующего интерфейс удаленного доступа. Конечно, этим «чем-то» является заглушка, а при выполнении передачи данных по сети и операций ввода-вывода возможны разные неприятности. Клиент должен подтвердить этот риск, обрабатывая или объявляя исключения удаленного доступа. Если методы интерфейса объявляют исключения, то эти исключения должны обрабатываться или объявляться всем кодом, вызывающим методы для ссылок интерфейсного типа.

```
import java.rmi.*;

public interface MyRemote extends Remote {
    public String sayHello() throws RemoteException;
}
```

Интерфейс удаленного доступа в java.rmi

Вызовы удаленных методов считаются «рискованными». Объявление RemoteException в каждом методе обращает внимание клиента на то, что вызов может не сработать.

③ Аргументы и возвращаемые значения должны быть примитивами или Serializable

Аргументы и возвращаемые значения удаленного метода должны быть либо примитивными типами, либо Serializable. И это логично: аргументы удаленного метода необходимо упаковать и передать по сети, а это делается посредством сериализации. То же происходит и с возвращаемыми значениями. При использовании примитивов, String и большинства типов API (включая массивы и коллекции) все будет нормально. Если вы передаете пользовательские типы, проследите за тем, чтобы ваши классы реализовали интерфейс Serializable.

```
public String sayHello() throws RemoteException;
```

Возвращаемое значение будет передаваться по каналу связи от сервера к клиенту, поэтому оно должно поддерживать сериализацию.

Шаг 2: создание реализации интерфейса удаленного доступа

1 Реализация интерфейса удаленного доступа

Служба должна реализовать интерфейс удаленного доступа — интерфейс, методы которого будут вызываться клиентом.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public String sayHello() {
       return "Server says, 'Hey'";
    }
    // more code in class
}
```

Компилятор проследит за тем, чтобы вы реализовали все методы интерфейса. В данном случае метод только один.

2 Расширение UnicastRemoteObject

Для работы в качестве удаленной службы ваш объект должен обладать некоторой стандартной функциональностью. Для включения этой функциональности проще всего расширить класс UnicastRemoteObject (из пакета java.rmi.server), чтобы этот (супер)класс выполнил всю работу за вас.

```
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    private static final long serialVersionUID = 1L;
```

UnicastRemoteObject реализует интерфейс Serializable, поэтому понадобится поле serialVersionUID.

3 Определение конструктора без аргументов, объявляющего RemoteException

У нового суперкласса UnicastRemoteObject имеется одна проблема — его конструктор может инициировать исключение RemoteException. Единственное решение заключается в объявлении конструктора вашей удаленной реализации, чтобы у вас появилось место для объявления RemoteException. Вспомните, что при создании экземпляра класса всегда вызывается конструктор его суперкласса. И если конструктор суперкласса инициирует исключение, у вас нет другой возможности, кроме объявления исключения в своем конструкторе.

```
public MyRemoteImpl() throws RemoteException { }
```

Размещать код в конструкторе не нужно. Это всего лишь способ объявить, что конструктор суперкласса инициирует исключение.

4 Регистрация службы в реестре RMI

Чтобы созданная служба стала доступной для удаленных клиентов, следует создать ее экземпляр и поместить его в реестр RMI (который должен работать в системе; в противном случае выполнение этой строки кода завершится неудачей). При регистрации объекта реализации система RMI помещает в реестр *заглушку*, потому что клиент взаимодействует именно с ней. Регистрация службы осуществляется статическим методом rebind() класса java.rmi.Naming.

```
try {
    MyRemote service = new MyRemoteImpl();
    Naming.rebind("RemoteHello", service);
} catch(Exception ex) {...}
```

Присвойте своей службе имя (чтобы клиенты могли искать ее в реестре) и зарегистрируйте ее в реестре RMI. Система RMI помещает в реестр информацию о заглушке.

Шаг 3: запуск rmiregistry

1 Откройте терминал и запустите rmiregistry.

Не забудьте выполнить команду из каталога, имеющего доступ к вашим классам. Проще всего сделать это из каталога classes.

Шаг 4: запуск службы

1 Откройте другой терминал и запустите службу

Запуск может осуществляться из метода main() в классе реализации или из отдельного стартового класса. В этом простом примере код запуска размещается в классе реализации – в методе main, который создает экземпляр объекта и регистрирует его в реестре RMI.

Будьте осторожны!

До появления Java 5 разработчикам приходилось генерировать статические заглушки и заготовки командой rmic. Теперь это можно не делать — более того, так делать не следует, потому что статические заглушки и заготовки считаются устаревшими.

Вместо этого заглушки и заготовки генерируются динамически. Это происходит автоматически, когда мы субклассируем UnicastRemoteObject (по аналогии с тем, как это делалось в классе MyRemoteImpl).

Часто Задаваемые Вопросы

В: А почему вы показываете заглушки и заготовки на диаграммах для кода RMI? Я думал, что они уже давно остались в прошлом.

О: Вы правы. Для заготовок исполнительная среда RMI может передавать клиентские вызовы удаленной службе напрямую, используя отражение, а заглушки генерируются динамически с использованием динамических посредников (о которых вы узнаете чуть позднее в этой главе). Заглушка удаленного объекта представляет собой экземпляр java.lang.reflect.Proxy, который автоматически генерируется для обработки всех подробностей доставки локальных вызовов клиента к удаленному объекту. Но мы решили показать как заглушку, так и заготовку, потому что на концептуальном уровне полезно понимать, что «за кулисами» есть нечто, обеспечивающее взаимодействия между клиентской заглушкой и удаленной службой.

Полный код серверной части

Интерфейс удаленного доступа:

RemoteException и интерфейс Remote находятся в пакете java.rmi.

Интерфейс ДОЛЖЕН расширять java.rmi.Remote

Все удаленные методы должны объявлять RemoteException.

```
import java.rmi.*;

public interface MyRemote extends Remote {

    public String sayHello() throws RemoteException;
}
```

Удаленная служба (реализация):

UnicastRemoteObject находится в пакете java.rmi.server.

Расширение UnicastRemoteObject — простейший способ создания объекта с поддержкой удаленного доступа.

Конечно, все методы интерфейса должны быть реализованы, но объявлять RemoteException НЕ ОБЯЗАТЕЛЬНО.

Вы ОБЯЗАНЫ реализовать интерфейс удаленного доступа!

Конструктор суперкласса (UnicastRemoteObject) объявляет исключение, поэтому вы должны определить конструктор как признак вызова небезопасного кода (конструктора суперкласса).

Создайте удаленный объект и зарегистрируйте его в реестре RMI статическим вызовом Naming.rebind(). Регистрируемое имя будет использоваться клиентами для поиска объекта в реестре.

```
import java.rmi.*;
import java.rmi.server.*;

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
   private static final long serialVersionUID = 1L;

   public String sayHello() {
       return "Server says, 'Hey'";
   }

   public MyRemoteImpl() throws RemoteException { }

   public static void main (String[] args) {

       try {
           MyRemote service = new MyRemoteImpl();
           Naming.rebind("RemoteHello", service);
       } catch(Exception ex) {
           ex.printStackTrace();
       }
   }
}
```

Как клиент получает объект заглушки?

Клиент должен получить объект заглушки (заместителя), поскольку именно его методы будут вызываться клиентом. За этой информацией он обращается к реестру RMI. Клиент указывает имя и запрашивает заглушку, зарегистрированную с этим именем.

Далее мы рассмотрим пример кода поиска и получения объекта заглушки.

А вот как работает эта схема...

Код под увеличительным стеклом

Клиент всегда указывает в качестве типа службы тип интерфейса удаленного доступа. Ему вообще не обязательно знать фактическое имя класса удаленной службы.

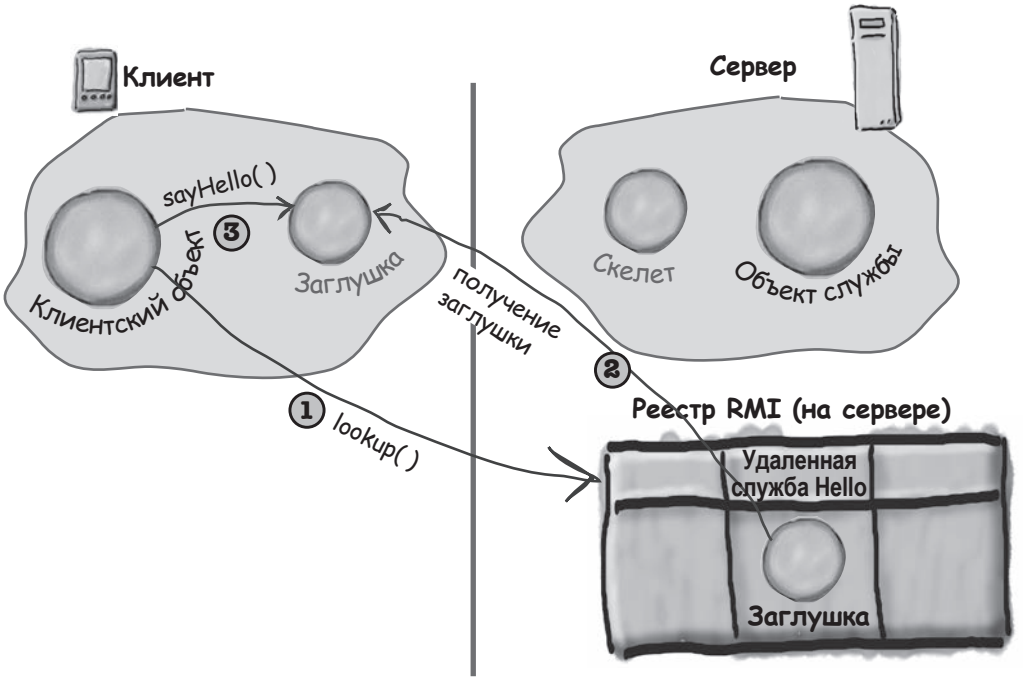
lookup() — статический метод класса Naming.

Имя, под которым была зарегистрирована служба.

```
MyRemote service =
    (MyRemote) Naming.lookup("rmi://127.0.0.1/RemoteHello");
```

Необходимо преобразование к типу интерфейса, поскольку по запросу возвращается тип Object.

Имя или IP-адрес хоста, на котором работает служба.



Как работает эта схема...

1 Клиент обращается с запросом к реестру RMI

```
Naming.lookup("rmi://127.0.0.1/RemoteHello");
```

2 Реестр RMI возвращает объект заглушки

(как возвращаемое значение метода lookup), а RMI автоматически десериализует его.

3 Клиент вызывает метод заглушки, словно метод НАСТОЯЩЕЙ службы.

Полный клиентский код

```
import java.rmi.*;

public class MyRemoteClient {
   public static void main (String[] args) {
        new MyRemoteClient().go();
   }

  public void go() {

     try {
        MyRemote service = (MyRemote) Naming.lookup("rmi://127.0.0.1/RemoteHello");

        String s = service.sayHello();

        System.out.println(s);
      } catch(Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

Класс Naming (для поиска в реестре RMI) находится в пакете java.rmi.

Возвращается из реестра в виде типа Object, не забудьте выполнить преобразование.

IP=адрес или имя хоста.

...и имя, использованное для идентификации службы.

Выглядит как обычный вызов метода! (кроме подтверждения возможности RemoteException).

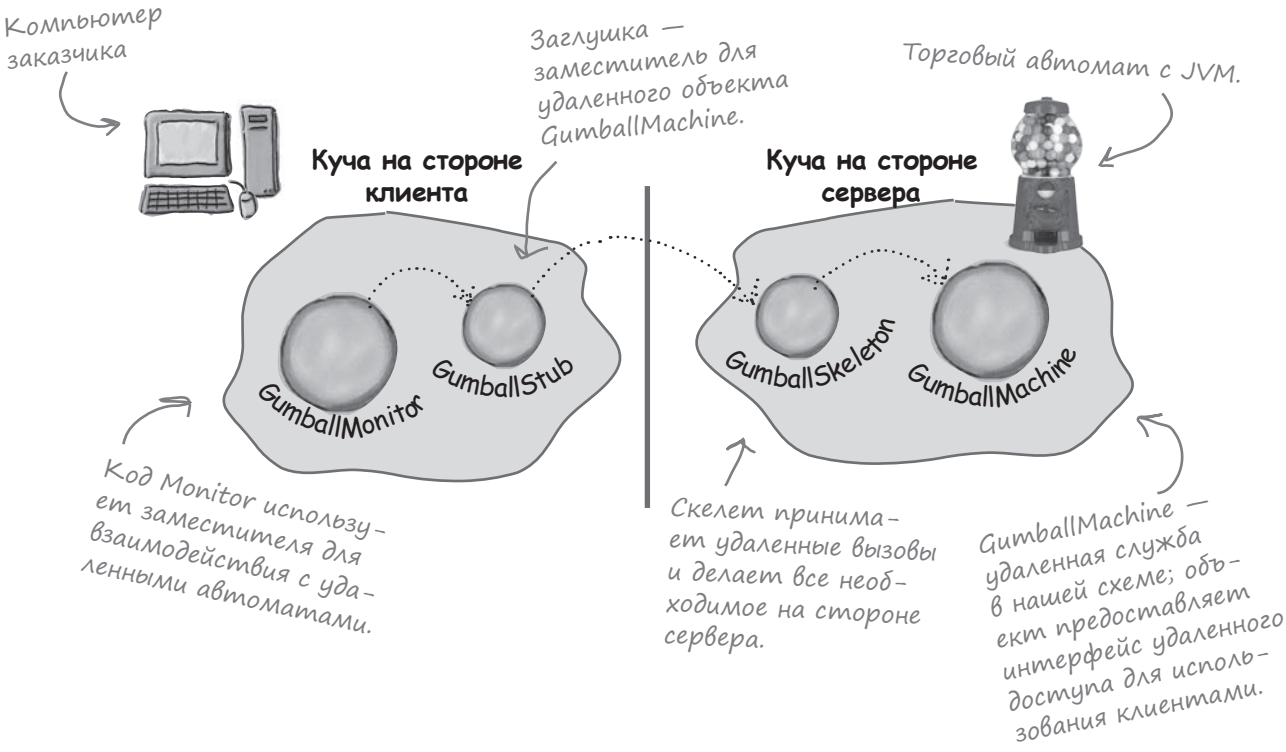
Типичные ошибки, которые совершаются программистами при работе с RMI:

1) Программа rmiregistry не запускается перед запуском удаленной службы (в момент регистрации службы вызовом Naming.rebind() программа rmiregistry должна работать в системе!).

2) Типы аргументов и возвращаемых значений не сериализуются (вы не узнаете об этом до момента выполнения; такие ошибки компилятором не обнаруживаются).

Возвращаемся к удаленному заместителю GumballMachine

Итак, после знакомства с основами RMI мы располагаем всем необходимым для реализации удаленного заместителя GumballMachine. Давайте посмотрим, как удаленный сбор данных о торговых автоматах вписывается в уже известную нам инфраструктуру:



Преобразование GumballMachine в удаленную службу

Преобразование кода начинается с включения обслуживания удаленных запросов от клиентов в классе GumballMachine. Иначе говоря, класс GumballMachine превращается в сетевую службу. Для этого необходимо следующее:

1) Создать интерфейс удаленного доступа для GumballMachine. Интерфейс предоставляет набор методов, вызываемых в удаленном режиме.

2) Убедиться в том, что все возвращаемые типы интерфейса сериализуемы.

3) Реализовать интерфейс в конкретном классе.

Начнем с интерфейса удаленного доступа:

*Не забудьте импортировать java.rmi.**

Интерфейс удаленного доступа.

```
import java.rmi.*;

public interface GumballMachineRemote extends Remote {
    public int getCount() throws RemoteException;
    public String getLocation() throws RemoteException;
    public State getState() throws RemoteException;
}
```

Все возвращаемые типы должны быть примитивными или Serializable...

Методы, которые будут поддерживаться службой. Каждый метод может инициировать RemoteException.

Один из возвращаемых типов не поддерживает Serializable: это класс State. Исправляем...

Serializable находится в пакете java.io.

```
import java.io.*;

public interface State extends Serializable {
    public void insertQuarter();
    public void ejectQuarter();
    public void turnCrank();
    public void dispense();
}
```

Просто расширяем интерфейс Serializable interface (который не содержит методов) — и теперь объект State во всех субклассах может передаваться по сети.

Осталось решить еще одну проблему с сериализацией. Как вы, вероятно, помните, каждый объект State хранит ссылку на объект GumballMachine для вызова методов и изменения его состояния. Мы не хотим, чтобы вместе с объектом State сериализовался и передавался весь объект GumballMachine. Проблема решается просто:

```
public class NoQuarterState implements State {
    private static final long serialVersionUID = 2L;
    transient GumballMachine gumballMachine;
    // all other methods here
}
```

В каждой реализации State переменная экземпляра GumballMachine помечается ключевым словом transient. Оно сообщает JVM, что это поле не сериализуется.

Класс GumballMachine уже реализован, но мы должны обеспечить его работу в режиме службы и обработку запросов, поступающих по сети. Для этого класс GumballMachine должет сделать все необходимое для реализации интерфейса GumballMachineRemote.

Как было показано ранее, это достаточно просто, нужно лишь добавить пару полей...

Необходимо импортировать пакеты rmi.

Класс GumballMachine будет субклассировать UnicastRemoteObject; это позволяет ему работать в режиме удаленной службы.

GumballMachine также должен реализовать интерфейс удаленного доступа...

```
import java.rmi.*;
import java.rmi.server.*;

public class GumballMachine
        extends UnicastRemoteObject implements GumballMachineRemote
{
    private static final long serialVersionUID = 2L;
    // Переменные

    public GumballMachine(String location, int numberGumballs) throws RemoteException {
        // Код
    }

    public int getCount() {
        return count;
    }

    public State getState() {
        return state;
    }

    public String getLocation() {
        return location;
    }

    // Другие методы
}
```

...и конструктор должен объявить исключение, потому что оно объявлено в суперклассе.

А здесь ничего не изменяется!

Регистрация в реестре RMI...

Код службы готов. Теперь нужно запустить его, чтобы он мог принимать запросы. Служба регистрируется в реестре RMI, чтобы клиенты могли получить доступ к ней.

В тестовую программу включается небольшой фрагмент кода:

```
public class GumballMachineTestDrive {

    public static void main(String[] args) {
        GumballMachineRemote gumballMachine = null;
        int count;
        if (args.length < 2) {
            System.out.println("GumballMachine <name> <inventory>");
            System.exit(1);
        }

        try {
            count = Integer.parseInt(args[1]);

            gumballMachine = new GumballMachine(args[0], count);
            Naming.rebind("//" + args[0] + "/gumballmachine", gumballMachine);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Создание экземпляра заключено в блок try/catch, потому что конструктор может инициировать исключения.

Также добавляем вызов Naming.rebind, который публикует заглушку GumballMachine под именем gumballmachine.

Остается выполнить пару команд...

Сначала выполняется эта команда.

Запуск службы реестра RMI.

Замените именем своего компьютера.

```
File Edit Window Help Huh?
% rmiregistry

File Edit Window Help Huh?
% java GumballMachineTestDrive seattle.mightygumball.com 100
```

Затем выполняется эта команда.

Служба GumballMachine запущена и зарегистрирована в реестре RMI.

А теперь клиент GumballMonitor...

Еще не забыли класс GumballMonitor? Мы хотели использовать его для работы по сети, но по возможности обойтись без переписывания кода. Именно это мы сейчас и сделаем, хотя несколько изменений внести все же придется.

Необходимо импортировать пакет RMI, потому что ниже используется класс RemoteException...

```
import java.rmi.*;

public class GumballMonitor {
    GumballMachineRemote machine;

    public GumballMonitor(GumballMachineRemote machine) {
        this.machine = machine;
    }

    public void report() {
        try {
            System.out.println("Gumball Machine: " + machine.getLocation());
            System.out.println("Current inventory: " + machine.getCount() + " gumballs");
            System.out.println("Current state: " + machine.getState());
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

В дальнейшем мы будем работать с реализацией интерфейса удаленного доступа, а не с конкретным классом GumballMachine.

Также необходимо перехватывать все исключения, возможные при вызове методов, которые фактически выполняются по сети.

Тестовая программа для монитора

Теперь у нас есть все необходимые компоненты. Осталось написать тестовый код, чтобы заказчик мог получить данные от нескольких торговых автоматов:

```
import java.rmi.*;

public class GumballMonitorTestDrive {

    public static void main(String[] args) {
        String[] location = {"rmi://santafe.mightygumball.com/gumballmachine",
                             "rmi://boulder.mightygumball.com/gumballmachine",
                             "rmi://seattle.mightygumball.com/gumballmachine"};

        GumballMonitor[] monitor = new GumballMonitor[location.length];

        for (int i=0;i < location.length; i++) {
            try {
                GumballMachineRemote machine =
                        (GumballMachineRemote) Naming.lookup(location[i]);
                monitor[i] = new GumballMonitor(machine);
                System.out.println(monitor[i]);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        for(int i=0; i < monitor.length; i++) {
            monitor[i].report();
        }
    }
}
```

Тестовая программа, которую будет запускать заказчик!

Местонахождения всех автоматов, которые мы собираемся отслеживать.

Создаем массив местонахождений, по одному для каждого автомата.

Также создаем массив мониторов.

Теперь нужно получить заместителя для каждого автомата.

Перебираем автоматы и для каждого выводим отчет.

Код под увеличительным стеклом

Возвращает заместителя для удаленного объекта GumballMachine (или инициирует исключение, если найти объект не удается).

Напоминаем: Naming.lookup() — статический метод из пакета RMI, который ищет службу в реестре RMI по заданному местонахождению и имени.

```
try {
    GumballMachineRemote machine =
                (GumballMachineRemote) Naming.lookup(location[i]);

    monitor[i] = new GumballMonitor(machine);

} catch (Exception e) {
    e.printStackTrace();
}
```

Получив заместителя для удаленного автомата, мы создаем новый объект GumballMonitor и передаем ему отслеживаемый автомат.

Новая демонстрация для заказчика...

Что ж, соберем все вместе и построим новую демонстрацию. Для начала следует позаботиться о том, чтобы новый код работал на нескольких автоматах:

На каждом компьютере запустите rmiregistry в фоновом режиме или в отдельном терминальном окне...

...затем запустите тестовую программу с указанием местонахождения и исходного количества шариков.

```
File Edit Window Help Huh?
% rmiregistry &
% java GumballMachineTestDrive santafe.mightygumball.com 100
```

```
File Edit Window Help Huh?
% rmiregistry &
% java GumballMachineTestDrive boulder.mightygumball.com 100
```

```
File Edit Window Help Huh?
% rmiregistry &
% java GumballMachineTestDrive seattle.mightygumball.com 250
```

популярный автомат!

А теперь вручим монитор заказчику. Будем надеяться, что на этот раз результат его устроит:

```
File Edit Window Help GumballsAndBeyond
% java GumballMonitorTestDrive
Gumball Machine: santafe.mightygumball.com
Current inventory: 99 gumballs
Current state: waiting for quarter

Gumball Machine: boulder.mightygumball.com
Current inventory: 44 gumballs
Current state: waiting for turn of crank

Gumball Machine: seattle.mightygumball.com
Current inventory: 187 gumballs
Current state: waiting for quarter
%
```

Монитор перебирает удаленные автоматы и вызывает их методы getLocation(), getCount() и getState().

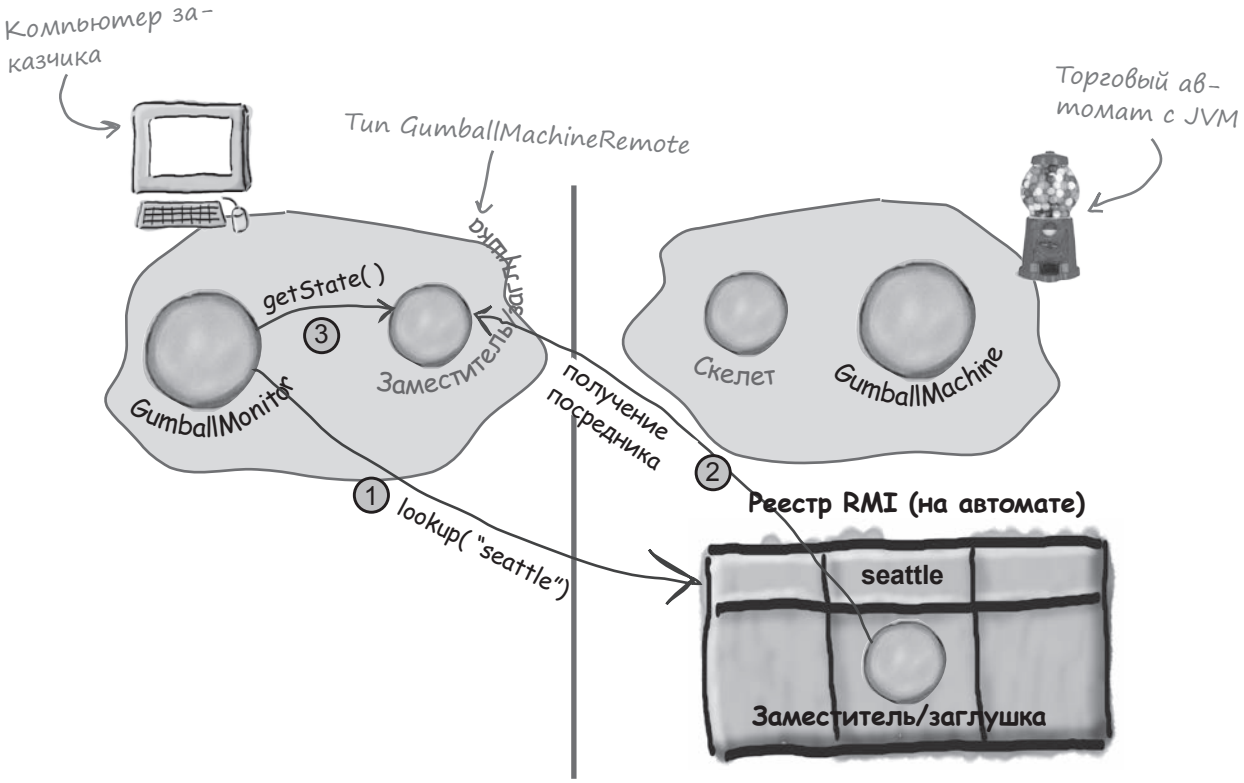
Потрясающе; это произведет революцию в моем бизнесе и сметет конкурентов!

В результате вызова методов заместителя вызываются удаленные методы, возвращающие String, целое число и объект State. Так как мы используем заместителя, GumballMonitor не знает, что вызовы выполняются в удаленном режиме — впрочем, для него это несущественно (разве что ему приходится побеспокоиться о возможных исключениях удаленного доступа).

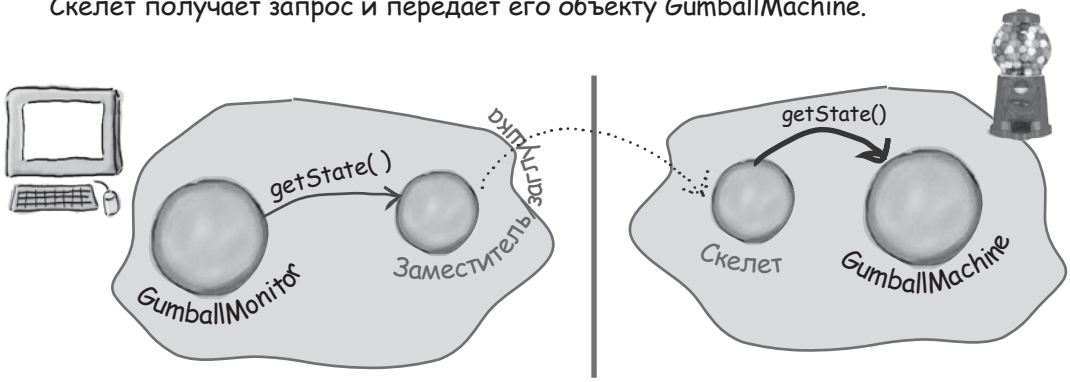
Все отлично работает! Но я хочу быть уверен в том, что я правильно понимаю, **как** это происходит...

За сценой

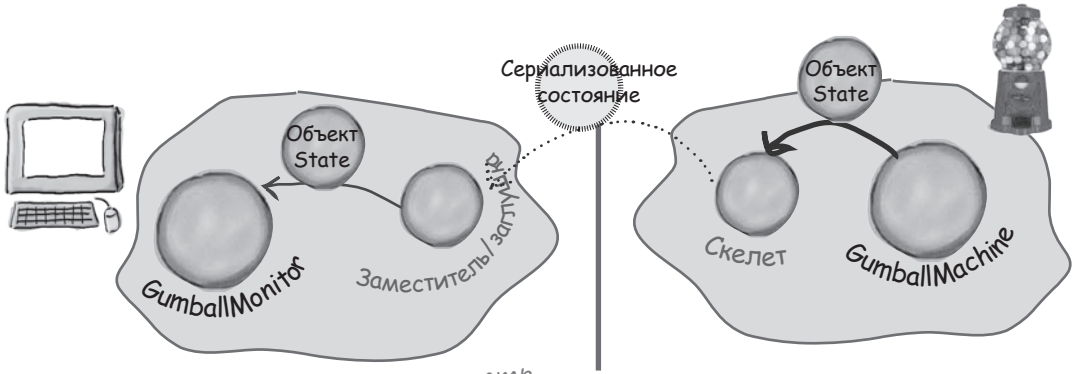
1 Заказчик запускает монитор, который сначала получает заместителей для удаленных автоматов, а затем вызывает для каждого из них getState() (вместе с getCount() и getLocation()).



❷ Вызов метода getState() заместителя передается удаленной службе. Скелет получает запрос и передает его объекту GumballMachine.



❸ Скелет получает от GumballMachine объект State, сериализует его и передает по сети заместителю. Заместитель десериализует объект и возвращает его монитору в виде объекта.



Монитор не изменился, если не считать того, что он теперь знает о возможных исключениях удаленного доступа. Кроме того, он использует интерфейс GumballMachineRemote вместо конкретной реализации.

Класс GumballMachine тоже реализует другой интерфейс и может инициировать исключение удаленного доступа из конструктора, но в остальном код не изменился.

Также добавился небольшой фрагмент кода для регистрации и поиска заглушек по реестру RMI. Но если вы пишете код, который должен работать по Интернету, механизм поиска понадобится в любом случае.

Определение паттерна Заместитель

Позади осталась изрядная часть этой главы; как видите, объяснение схемы с удаленным заместителем требует довольно много места. Но несмотря на это, определение и диаграмма классов паттерна Заместитель относительно просты. У паттерна существует несколько разновидностей, мы поговорим о них позже. А пока давайте более подробно разберем общий паттерн.

Определение паттерна Заместитель:

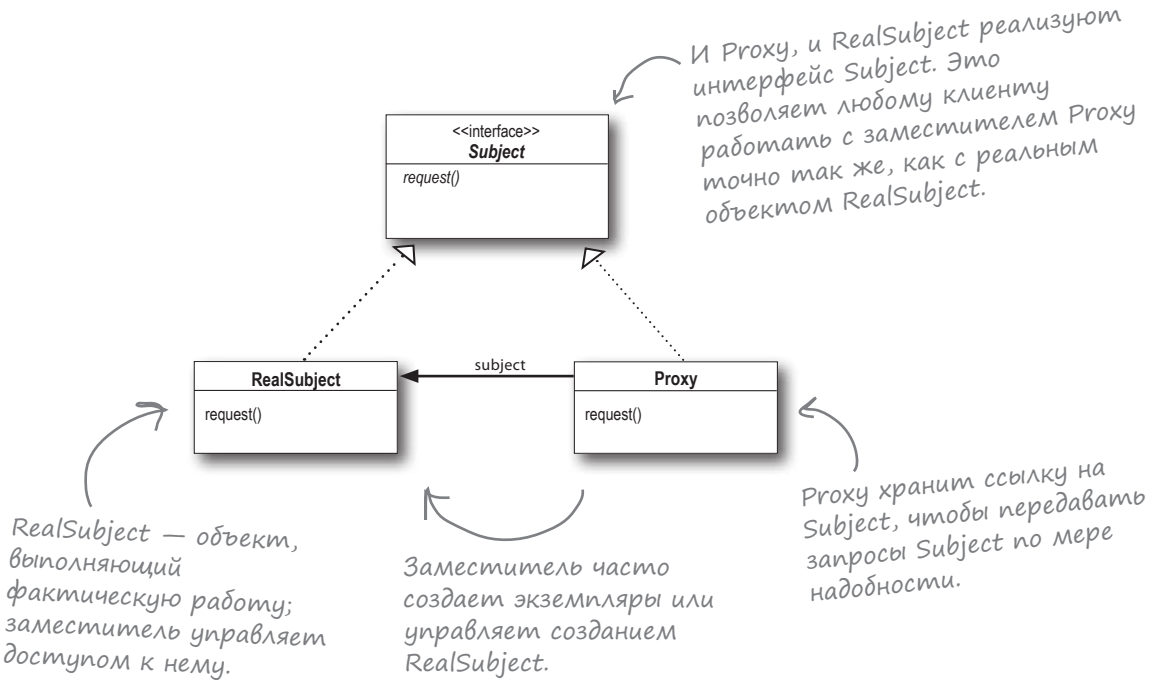
> **Паттерн Заместитель** предоставляет суррогатный объект, управляющий доступом к другому объекту.

Используйте паттерн Заместитель для создания объекта, управляющего доступом к другому объекту — удаленному, защищенному требующему слишком больших затрат при созданиии и т. д.

Мы рассмотрели пример того, как паттерн Заместитель предоставляет суррогатный объект для другого объекта. Но как насчет заместителя, управляющего доступом? Звучит немного странно... В нашем примере с торговыми автоматами заместитель должен обеспечить доступ к удаленному объекту, потому что клиент (монитор) не умеет взаимодействовать с удаленным объектом. Таким образом, удаленный заместитель берет на себя управление доступом, чтобы избавить клиента от низкоуровневых подробностей сетевых взаимодействий. Как упоминалось ранее, существует несколько разновидностей паттерна Заместитель, и все они обычно тем или иным образом связаны с тем, как именно организуется «управление доступом». Мы подробно рассмотрим эту тему позднее, а пока упомянем лишь несколько вариантов управления доступом в заместителях:

- Удаленный заместитель управляет доступом к удаленному объекту.
- Виртуальный заместитель управляет доступом к ресурсу, создание которого требует больших затрат ресурсов.
- Защитный заместитель контролирует доступ к ресурсу в соответствии с системой привилегий.

Теперь, когда вы представляете общую суть паттерна, перейдем к диаграмме классов...



Основные блоки этой диаграммы:

Subject — интерфейс для взаимодействия с RealSubject и Proxy. Реализация общего интерфейса позволяет использовать Proxy вместо RealSubject во всех операциях.

RealSubject — объект, выполняющий фактическую работу. Proxy представляет этот объект и управляет доступом к нему.

Заместитель Proxy хранит ссылку на RealSubject. В некоторых случаях Proxy отвечает за создание и уничтожение RealSubject. Клиенты взаимодействуют с RealSubject через Proxy. Так как Proxy и RealSubject реализуют общий интерфейс (Subject), Proxy может использоваться повсюду вместо RealSubject. Proxy также управляет доступом к RealSubject; это может быть полезно, если объект RealSubject работает на удаленном компьютере, если создание его экземпляров связано с большими затратами, или если доступ к нему должен быть тем или иным образом защищен.

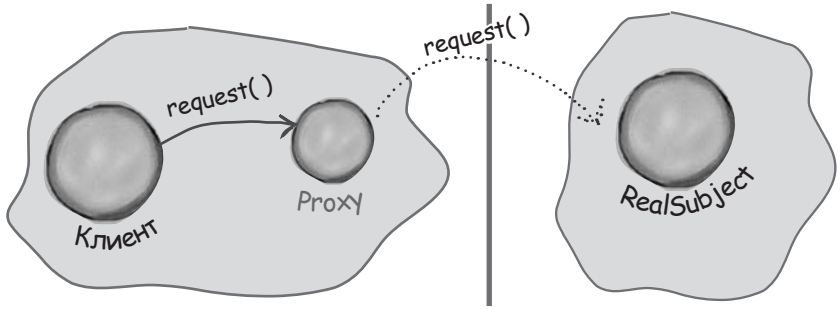
Итак, теперь вы имеете некоторое представление об этом паттерне, и мы можем рассмотреть другие варианты его использования, помимо уже известного нам удаленного заместителя...

Знакомьтесь: Виртуальный Заместитель

К настоящему моменту мы рассмотрели определение паттерна Заместитель и одно из его конкретных воплощений: *Удаленный Заместитель*. Переходим к другой разновидности: *Виртуальный Заместитель*. Паттерн Заместитель существует во многих формах, но все они строятся на приблизительно похожей архитектуре опосредованного доступа. Почему паттерн имеет столько разновидностей? Потому что он может применяться во множестве разных ситуаций. Давайте рассмотрим Виртуального Заместителя и сравним его с Удаленным Заместителем:

Удаленный Заместитель

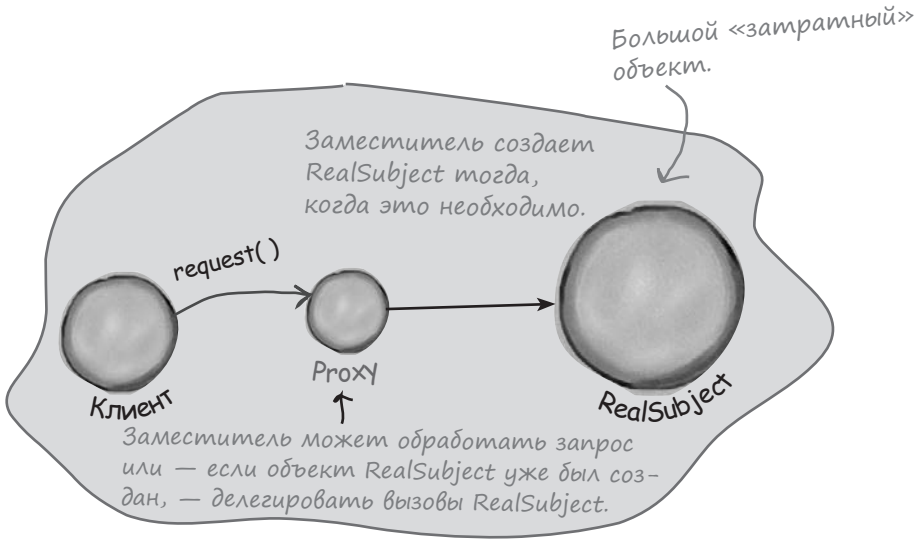
В этой разновидности заместитель выполняет функции локального представителя объекта, находящегося в другой виртуальной машине Java. Вызов метода заместителя передается по сети, метод вызывается на удаленном компьютере, результат возвращается заместителю, а затем и клиенту.



Хорошо знакомая диаграмма...

Виртуальный Заместитель

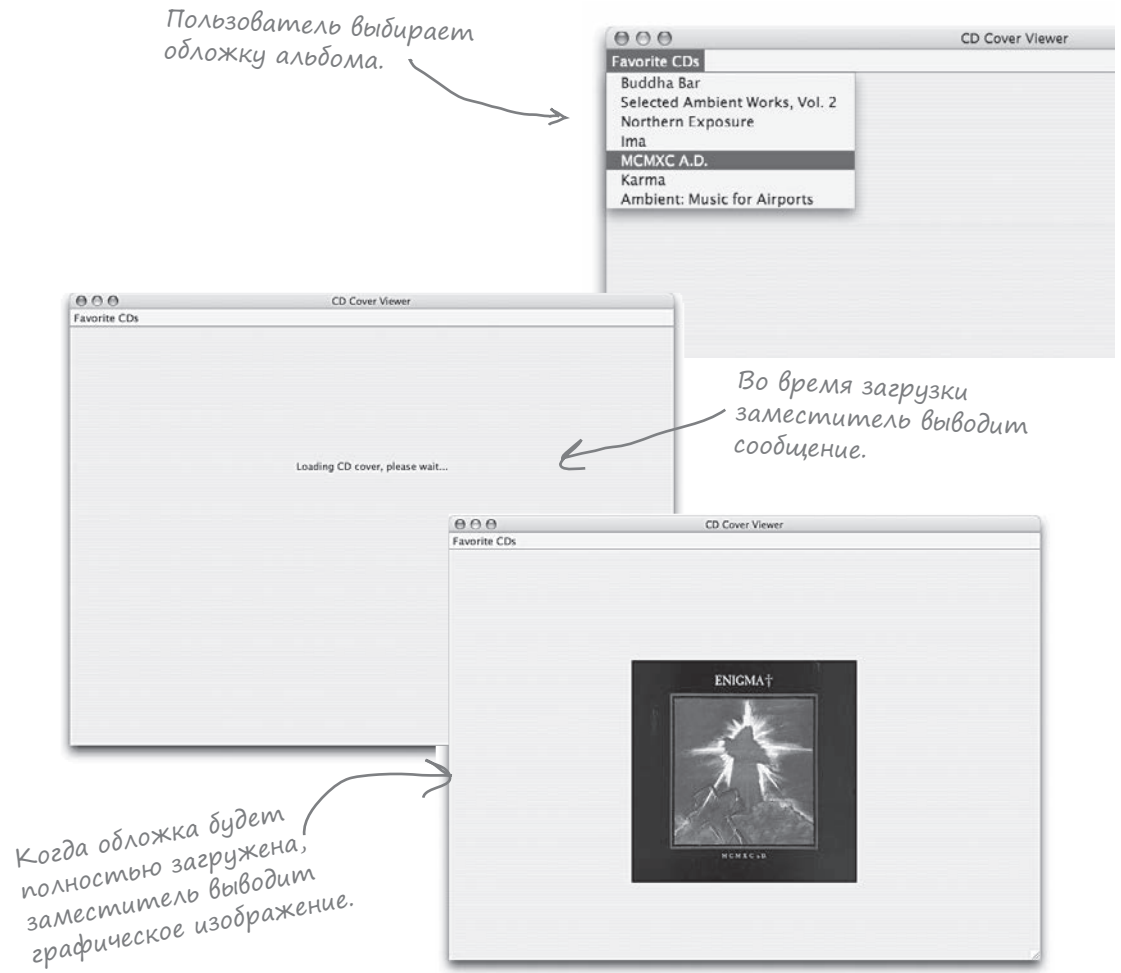
Виртуальный Заместитель представляет объект, создание которого сопряжено с большими затратами. Создание объекта часто откладывается до момента его непосредственного использования; Виртуальный Заместитель также выполняет функции суррогатного представителя объекта до и во время его создания. В дальнейшем заместитель делегирует запросы непосредственно RealSubject.



Отображение обложек компакт-дисков

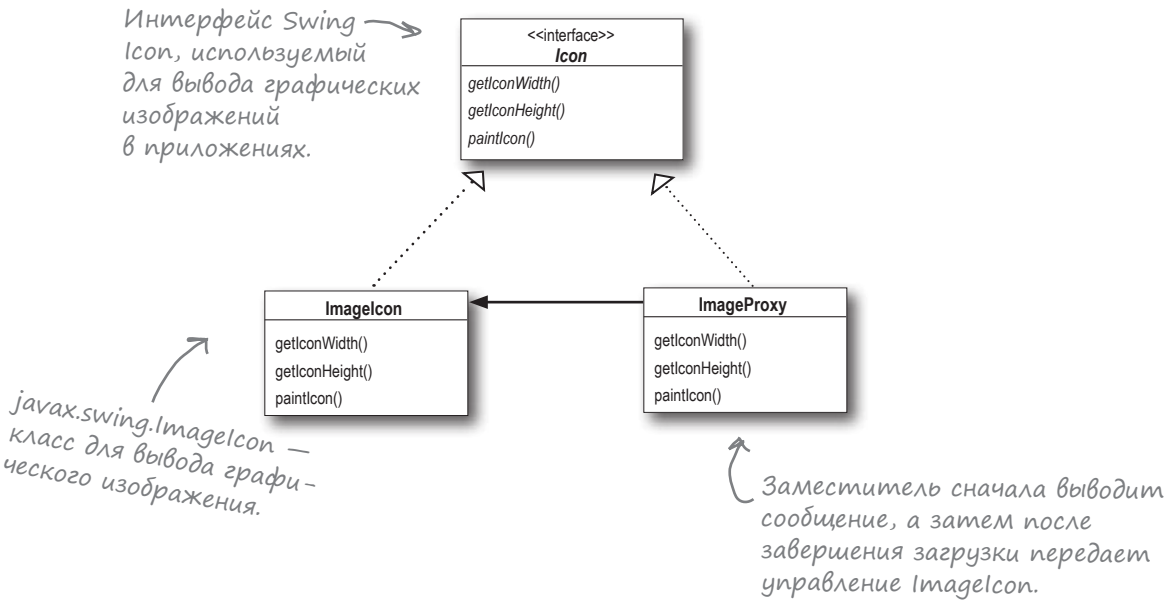
Допустим, вы пишете приложение для вывода обложек ваших любимых компакт-дисков. Диск выбирается в меню, а обложка загружается из Интернета (например, с сайта Amazon.com). При программировании с использованием Swing можно создать объект Icon и указать, что изображение должно загружаться по сети. Однако в зависимости от загруженности и пропускной способности канала загрузка может занять некоторое время; желательно, чтобы приложение отображало какую-то информацию в ходе ожидания. Кроме того, загрузка не должна парализовать работу всего приложения. После завершения загрузки сообщение исчезает, а на экране появляется обложка.

Для решения задачи проще всего воспользоваться виртуальным заместителем. Виртуальный заместитель заменяет объект Icon, управляя процессом фоновой загрузки, и до момента ее завершения выводит временное сообщение. Когда изображение будет загружено, заместитель передает управление Icon.



Проектирование виртуального заместителя

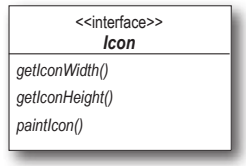
Прежде чем писать код программы просмотра обложек, рассмотрим диаграмму классов. Как видите, она мало отличается от диаграммы классов Удаленного Заместителя, но в данном случае заместитель используется для замещения объекта, создание которого занимает относительно много времени (так как данные передаются по сети).



Класс ImageProxy будет работать так:

1. **ImageProxy создает ImageIcon и начинает загрузку данных с сетевого URL-адреса.**
2. **Во время передачи графических данных ImageProxy выводит сообщение «Loading CD cover, please wait...».**
3. **Когда изображение будет полностью загружено, ImageProxy делегирует ImageIcon все вызовы методов, включая paintIcon(), getWidth() и getHeight().**
4. **Когда пользователь запросит новое изображение, мы создаем нового заместителя, и весь процесс повторяется заново.**

Класс ImageProxy



Класс ImageProxy реализует интерфейс Icon.

```
class ImageProxy implements Icon {
    volatile ImageIcon imageIcon;
    final URL imageURL;
    Thread retrievalThread;
    boolean retrieving = false;

    public ImageProxy(URL url) { imageURL = url; }

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    synchronized void setImageIcon(ImageIcon imageIcon) {
        this.imageIcon = imageIcon;
    }
    public void paintIcon(final Component c, Graphics g, int x, int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading CD cover, please wait...", x+300, y+190);
            if (!retrieving) {
                retrieving = true;
                retrievalThread = new Thread(new Runnable() {
                    public void run() {
                        try {
                            setImageIcon(new ImageIcon(imageURL, «CD Cover»));
                            c.repaint();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                });
                retrievalThread.start();
            }
        }
    }
}
```

В переменной imageIcon хранится НАСТОЯЩИЙ объект Icon, который должен отображаться после загрузки.

Конструктору передается URL-адрес изображения — того, которое должно отображаться после загрузки!

До завершения загрузки возвращаются значения длины и ширины по умолчанию; затем управление передается imageIcon.

imageIcon используется разными потоками, поэтому кроме объявления переменной с модификатором volatile (для защиты операций чтения) мы используем синхронизированный метод записи (для защиты операций записи).

Здесь происходит самое интересное. Изображение прорисовывается на экране (вызов делегируется imageIcon). Но если объект ImageIcon еще не создан, мы создаем его. Происходящее более подробно рассматривается на следующей странице...

Код под увеличительным стеклом

Метод вызывается тогда, когда требуется перерисовать изображение на экране.

```
public void paintIcon(final Component c, Graphics  g, int x,  int y) {
     if (imageIcon != null) {

         imageIcon.paintIcon(c, g, x, y);

     } else {

         g.drawString("Loading CD cover, please wait...", x+300, y+190);
         if (!retrieving) {

             retrieving = true;
             retrievalThread = new Thread(new Runnable() {
                 public void run() {
                     try {
                         setImageIcon(new ImageIcon(imageURL, «CD Cover»));
                         c.repaint();
                     } catch (Exception e) {
                         e.printStackTrace();
                     }
                 }
             });

             retrievalThread.start();
         }
     }
}
```

Если объект уже существует, то требование о перерисовке передается ему.

В противном случае выводится сообщение о загрузке.

Здесь загружается НАСТОЯЩЕЕ изображение. Следует учесть, что загрузка осуществляется синхронно: конструктор ImageIcon не возвращает управление до завершения загрузки. Соответственно, операции обновления экрана и вывода сообщения приходится осуществлять асинхронно. Подробнее на следующей странице...

Код под микроскопом

Если загрузка изображения еще не началась...

...можно начать загрузку (на всякий случай поясним, что paint вызывается только одним программным потоком, поэтому наша схема является потоково-безопасной).

Чтобы загрузка не парализовала пользовательский интерфейс программы, она будет выполняться в отдельном потоке.

```
if (!retrieving) {
    retrieving = true;

    retrievalThread = new Thread(new Runnable() {
        public void run() {
            try {
                setImageIcon(new ImageIcon(imageURL, «CD Cover»));
                c.repaint();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
    retrievalThread.start();
}
```

В отдельном потоке создается экземпляр объекта Icon. Его конструктор не возвращает управление до завершения загрузки данных.

После завершения загрузки оповещаем Swing о необходимости перерисовки.

Таким образом, при следующей перерисовке экрана после создания экземпляра ImageIcon метод paintIcon выведет изображение, а не текстовое сообщение.

Головоломка

Похоже, класс ImageProxy обладает двумя состояниями, причем нужное состояние выбирается при помощи условной конструкции. Можете ли вы предложить другой паттерн для упрощения этого кода? Как бы вы переработали ImageProxy в контексте этого паттерна?

```
class ImageProxy implements Icon {
    // Переменные экземпляров и конструктор

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics  g, int x,  int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Loading CD cover, please wait...", x+300, y+190);
            // ...продолжение...
        }
    }
}
```

Два состояния

Два состояния

Два состояния

Тестирование программы просмотра обложек

Готово к употреблению

Пора заняться тестированием виртуального заместителя. Новый класс ImageProxyTestDrive создает окно с панелью, настраивает меню и создает заместителя. Мы не будем описывать код во всех подробностях, но вы всегда можете загрузить полную версию программы и рассмотреть ее самостоятельно — или же обратиться к полному листингу виртуального заместителя в конце главы.

А здесь приводится небольшой фрагмент тестовой программы:

```
public class ImageProxyTestDrive {
    ImageComponent imageComponent;
    public static void main (String[] args) throws Exception {
        ImageProxyTestDrive testDrive = new ImageProxyTestDrive();
    }

    public ImageProxyTestDrive() throws Exception{

        // Создание панели и меню

        Icon icon = new ImageProxy(initialURL);
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
    }
}
```

Создаем заместителя и связываем его с исходным URL-адресом. Каждый раз, когда в меню выбирается новый диск, программа создает нового заместителя.

Затем заместитель упаковывается в компонент для добавления к объектам панели.

Добавляем заместителя к объектам панели, на которой должно выводиться изображение.

А теперь запустим тестовую программу:

```
File Edit Window Help JustSomeOfTheCDsThatGotUsThroughThisBook
% java ImageProxyTestDrive
```

Примерный вид окна ImageProxyTestDrive.

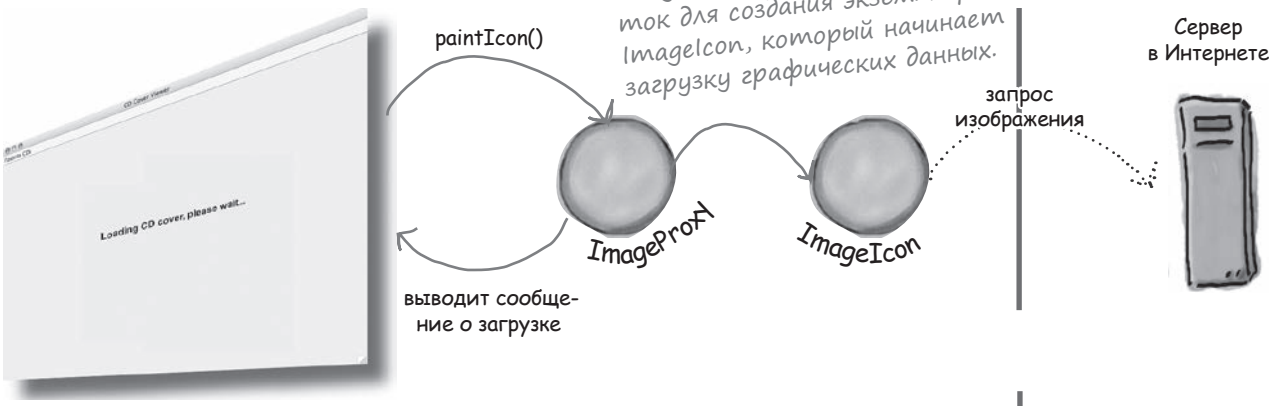
Попробуйте самостоятельно...

1. **Выбрать в меню разные обложки; проследите за тем, как заместитель выводит сообщение о загрузке до получения всех данных.**
2. **Изменить размеры окна во время вывода сообщения. Обратите внимание на то, что данные загружаются без блокировки окна Swing.**
3. **Включить в ImageProxyTestDrive свои любимые диски.**

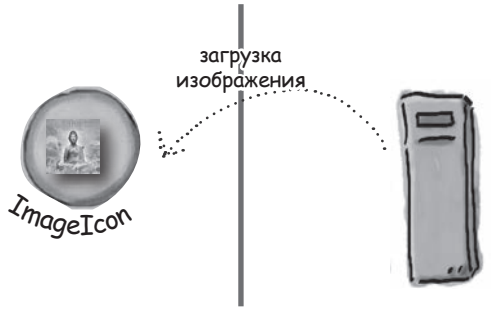
Что мы сделали?

За сценой 

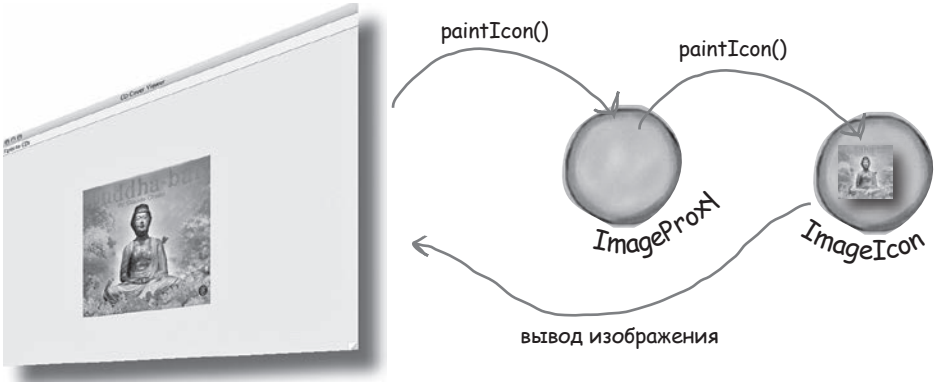
1. Мы создали класс заместителя ImageProxy. При вызове метода paintIcon() класс ImageProxy создает программный поток для загрузки изображения и создания ImageIcon.



2. После получения всех данных создание экземпляра ImageIcon завершается.



3. После создания ImageIcon при следующем вызове paintIcon() заместитель передает управление ImageIcon.



Часто Задаваемые Вопросы

В: Удаленный Заместитель и Виртуальный Заместитель имеют мало общего; вы уверены, что это ОДИН паттерн?

О: На практике встречается множество разновидностей паттерна Заместитель; их отличительной чертой является перехват методов реального объекта, вызываемых клиентом. Введение промежуточного уровня позволяет решать многие задачи: перенаправлять запросы удаленным объектам, предоставлять суррогаты для высокозатратных объектов на время их создания и т. д. Впрочем, это лишь начало; существует множество вариаций на тему паттерна Заместитель. Некоторые из них будут описаны в конце главы.

В: Мне кажется, что класс ImageProxy, скорее, использует паттерн Декоратор: один объект фактически упаковывается в другой, а вызовы методов делегируются ImageIcon. Может, я чего-то не замечаю?

О: Паттерны Заместитель и Декоратор иногда похожи друг на друга, но их цели принципиально различаются: Декоратор добавляет в класс новое поведение, а Заместитель управляет доступом к нему. Конечно, вы можете спросить: «Разве вывод сообщения не является расширением поведения?» Да, в каком-то смысле это так; но важнее здесь то, что ImageProxy управляет доступом к ImageIcon. Каким образом? Заместитель отделяет клиента от ImageIcon. Если бы не это разделение, то обновление всего интерфейса происходило бы только после полной загрузки изображения. Заместитель управляет доступом к ImageIcon и до завершения создания объекта предоставляет временную замену экранного представления. Когда объект будет готов, заместитель открывает доступ к нему.

В: Как заставить клиентов использовать заместителя вместо реального объекта?

О: Хороший вопрос. Один из стандартных приемов — создание фабрики, которая создает и возвращает реальный объект. В этом случае реальный объект можно упаковать в заместителя перед возвращением. Клиент даже не подозревает, что вместо реального объекта он использует заместителя.

В: В примере ImageProxy мы всегда создаем новый объект ImageIcon, даже если изображение уже было загружено ранее. Нельзя ли реализовать механизм кэширования данных, полученных в результате прошлых запросов?

О: Вы говорите о специализированной форме виртуальных заместителей, так называемом кэширующем заместителе. Кэширующий заместитель поддерживает кэш ранее созданных объектов и при поступлении запроса по возможности возвращает кэшированный объект.

Мы рассмотрим эту и некоторые другие разновидности паттерна Заместитель в конце главы.

В: Между паттернами Декоратор и Заместитель существует несомненное сходство, но как насчет паттерна Адаптер?

О: Паттерны Заместитель и Адаптер перехватывают запросы, предназначенные для других объектов. Однако при этом Адаптер изменяет интерфейс адаптируемых объектов, а Заместитель реализует тот же интерфейс.

Существует еще одно сходство, связанное с формой Защитного Заместителя. Защитный Заместитель может разрешить или запретить клиенту доступ к некоторым методам объекта в зависимости от роли клиента.

Таким образом, Защитный Заместитель может ограничивать интерфейс с точки зрения клиента; в этом отношении он напоминает некоторые адаптеры. Эта форма паттерна будет рассмотрена далее.

Беседа у камина

Заместитель и Декоратор переходят на личности

Заместитель

Привет, Декоратор. Полагаю, ты здесь потому, что люди нас иногда путают?

Я краду *твои* идеи? Брось. Я управляю доступом к объектам, а ты их просто декорируешь. Моя работа настолько важнее твоей, что это даже не смешно.

Будь по-твоему... Но я все равно не понимаю, почему ты думаешь, что я краду твои идеи. Я занимаюсь исключительно представлением объектов, а не их декорированием.

Мне кажется, ты чего-то не понимаешь. Я заменяю свои объекты, а не расширяю их поведение. Клиент использует меня как суррогат настоящего объекта, потому что я предотвращаю несанкционированный доступ, скрываю тот факт, что объект работает на удаленном компьютере. Мои цели не имеют с твоими ничего общего!

Декоратор

Да, нас действительно путают — в основном из-за того, что ты прикидываешься самостоятельным паттерном, а на самом деле ты всего лишь переодетый Декоратор. Нехорошо красть чужие идеи.

«Просто» декорирую? Ты думаешь, что это так легко и несерьезно? Так я скажу тебе, приятель, я расширяю *поведение*. Самое важное в объектах — то, что они *делают*!

Можешь называть это «представлением», какая разница... Сам подумай: твой Виртуальный Заместитель — обычный механизм добавления поведения на время загрузки большого затратного объекта, а Удаленный Заместитель — избавление клиентов от низкоуровневого взаимодействия с удаленными объектами. Все завязано на поведении, как я и сказал.

Называй, как хочешь. Я реализую тот же интерфейс, что и упакованные во мне объекты. Ты тоже.

Заместитель

Давай-ка разберемся. Упакованные объекты? Иногда мы неформально говорим, что заместитель упаковывает свой объект, но это неточное выражение.

Возьмем удаленного заместителя... Где тут упаковка? Я представляю объект, находящийся на другом компьютере, и управляю доступом к нему! А ты на такое способен?

Хорошо, возьмем виртуального заместителя... хотя бы из примера с обложками. Когда клиент впервые использует меня как заместителя, представляемый объект еще не существует!

Не думал, что декораторы настолько тупы! Ну конечно, я иногда создаю объекты — откуда они иначе возьмутся у виртуального заместителя! Хорошо, ты только что указал на важное различие между нами: декоратор только наводит внешний блеск, но ничего не создает.

Да, и после этого разговора я убежден, что ты — упрощенный заместитель!

Заместитель почти никогда не используется для многоуровневой упаковки. Если что-то приходится упаковывать на десять уровней, лучше пересмотреть архитектуру приложения.

Декоратор

Да ну? Почему же?

Допустим, но удаленный заместитель — особый случай. Еще примеры найдутся? Что-то сомневаюсь.

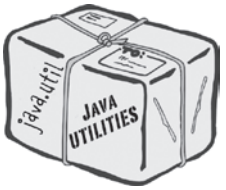
Ага, а теперь ты скажешь, что создание объектов — это тоже твоя заслуга.

Ах вот как?

Упрощенный заместитель? Хотел бы я видеть, как ты рекурсивно упакуешь объект на десяток уровней и при этом останешься в здравом уме.

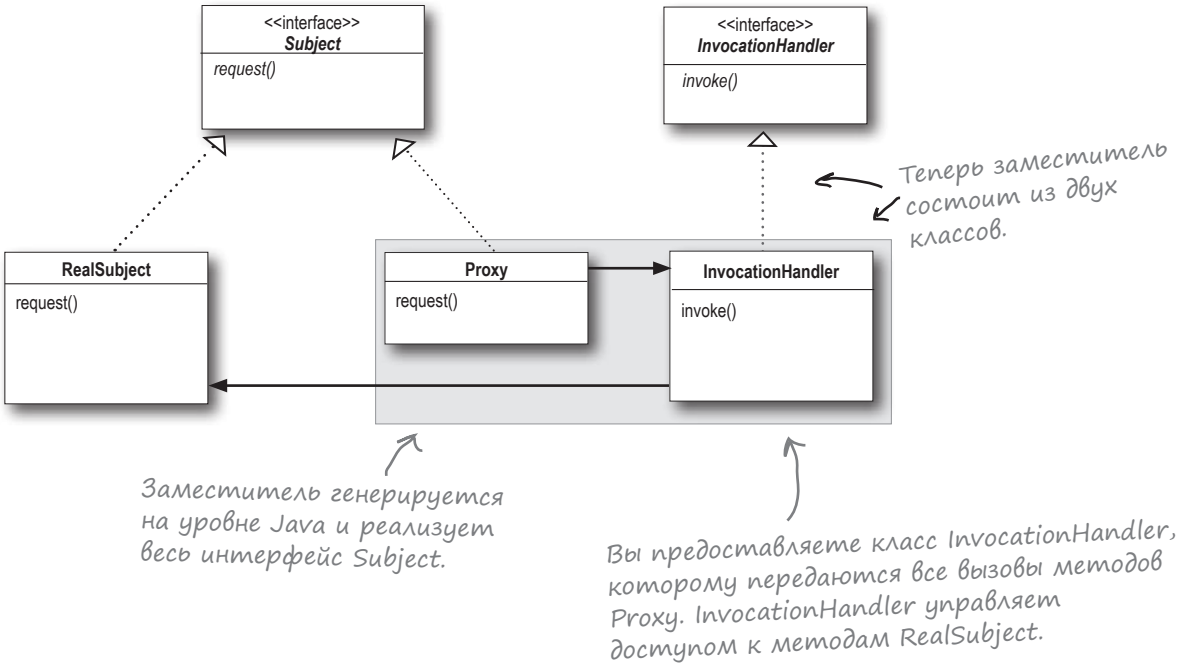
Да, это на тебя похоже: прикидываться, будто ты что-то значишь, хотя на самом деле ты только изображаешь другой объект. Могу тебя только пожалеть.

Создание защитного заместителя средствами Java API



Поддержка заместителей в языке Java находится в пакете java.lang.reflect. При использовании этого пакета Java позволяет *динамически* создать класс заместителя, который реализует один или несколько интерфейсов и перенаправляет вызовы методов заданному вами классу. Так как класс заместителя строится во время выполнения, такие заместители называются *динамическими*.

Мы воспользуемся механизмом динамических заместителей Java для создания нашей собственной реализации (защитного заместителя). Но сначала мы рассмотрим диаграмму классов, объясняющую суть динамических заместителей. Как это часто бывает в реальном мире, она слегка отличается от классического определения паттерна:



Так как Java строит класс Proxy *за вас*, вы должны как-то указать классу Proxy, что он должен делать. Этот код невозможно разместить в классе Proxy, как это делалось ранее, потому что мы не определяем реализацию этого класса. Если код не может находиться в классе Proxy, то где его место? В классе InvocationHandler. Задачей этого класса является обработка всех вызовов методов заместителя. Считайте, что Proxy после получения вызова метода обращается к InvocationHandler за выполнением фактической работы.

Давайте последовательно рассмотрим процесс использования динамического посредника...

Служба знакомств в Объективиле

В каждом городе должна быть своя служба знакомств, верно? Вам поручено реализовать такую службу в Объективиле. Стремясь творчески подойти к делу, вы предусмотрели систему оценок, которые участники могут ставить друг другу, — во-первых, у них будет больше стимулов для просмотра списка потенциальных кандидатов, а во-вторых, так интереснее.

Центральное место в вашей службе занимает компонент Person, предназначенный для чтения и записи данных кандидата:

Интерфейс; вскоре доберемся и до реализации...

Получение информации о кандидате: имя, пол, увлечения и оценка (1–10).

```
public interface PersonBean {

    String getName();
    String getGender();
    String getInterests();
    int getHotOrNotRating();

    void setName(String name);
    void setGender(String gender);
    void setInterests(String interests);
    void setHotOrNotRating(int rating);

}
```

Методы записи тех же данных.

Метод setHotOrNotRating() получает целое число и включает его в накапливаемую среднюю оценку кандидата.

Обратимся к реализации....

Реализация PersonBean

Класс PersonBeanImpl реализует интерфейс PersonBean.

```
public class PersonBeanImpl implements PersonBean {
    String name;
    String gender;
    String interests;
    int rating;
    int ratingCount = 0;

    public String getName() {
        return name;
    }

    public String getGender() {
        return gender;
    }

    public String getInterests() {
        return interests;
    }

    public int getHotOrNotRating() {
        if (ratingCount == 0) return 0;
        return (rating/ratingCount);
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public void setInterests(String interests) {
        this.interests = interests;
    }

    public void setHotOrNotRating(int rating) {
        this.rating += rating;
        ratingCount++;
    }
}
```

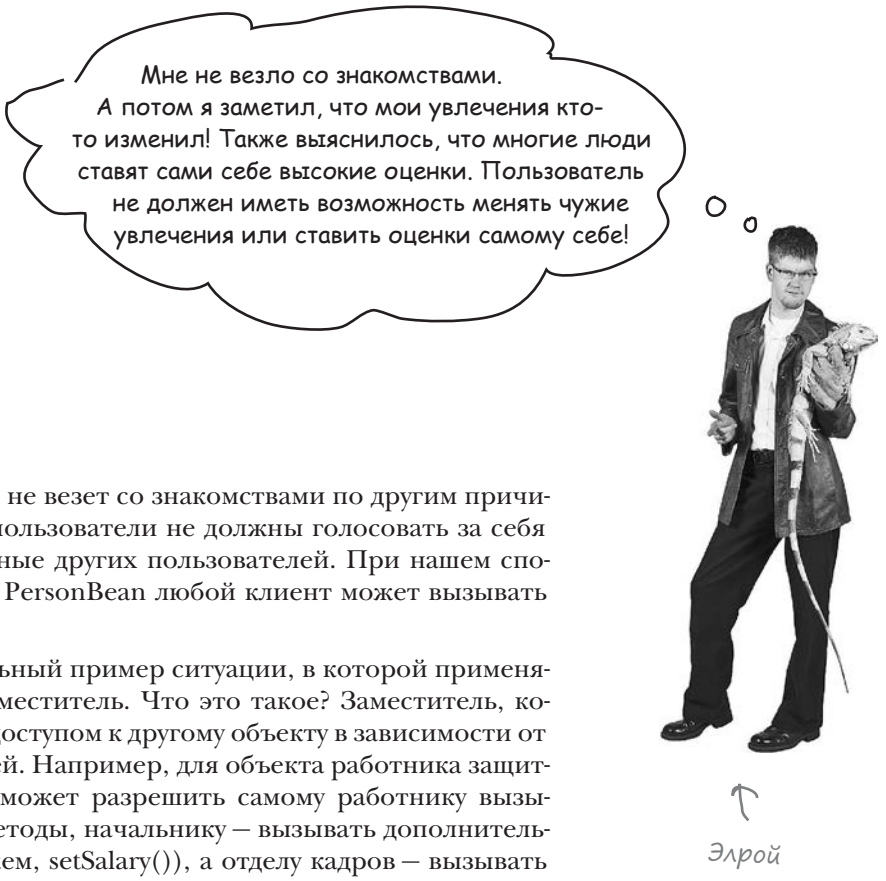
Переменные экземпляров.

Get-методы возвращают значения соответствующих переменных экземпляров...

...кроме метода getHotOrNotRating(), который вычисляет среднюю оценку делением суммы на количество оценок ratingCount.

Set-методы задают значения соответствующих переменных.

Наконец, метод setHotOrNotRating() увеличивает значение ratingCount и прибавляет оценку к накапливаемой сумме.



Возможно, Элрою не везет со знакомствами по другим причинам, но он прав: пользователи не должны голосовать за себя или изменять данные других пользователей. При нашем способе определения PersonBean любой клиент может вызывать любые методы.

Перед нами идеальный пример ситуации, в которой применяется защитный заместитель. Что это такое? Заместитель, который управляет доступом к другому объекту в зависимости от прав пользователей. Например, для объекта работника защитный заместитель может разрешить самому работнику вызывать некоторые методы, начальнику — вызывать дополнительные методы (скажем, setSalary()), а отделу кадров — вызывать любые методы объекта.

В нашей службе знакомств клиент должен иметь возможность задать свою личную информацию, но введенная информация не должна изменяться другими клиентами. С оценками ситуация прямо противоположная: они должны задаваться другими пользователями, но не самим клиентом. Кроме того, интерфейс PersonBean содержит ряд get-методов; так как эти методы не возвращают приватной информации, они могут вызываться любым пользователем.

Пятиминутная драма: защита клиентов

«Эпоха доткомов» осталась в прошлом. Чтобы найти лучшую, более высокооплачиваемую работу, было достаточно перейти через дорогу. У программистов даже появились свои агенты...

У меня есть предложение, можно пригласить ее к телефону?

Джейн Доткомм

Субъект

Агент

Она занята... Эээ... на важной встрече. А что вы хотели?

Как и защитный заместитель, агент управляет доступом к своему клиенту, пропуская только определенные звонки...

Думаю, мы можем предложить ей на 15% больше, чем на ее текущей работе.

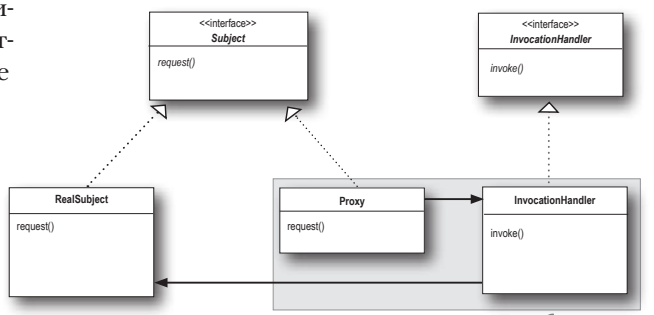
Не тратьте наше время попусту! Ни единого шанса! Звоните, когда появятся более выгодные предложения.

Общая картина: динамический заместитель для PersonBean

Нужно решить две проблемы: во-первых, пользователи не должны изменять свои оценки, а во-вторых, для них должна быть недоступна персональная информация других пользователей. Мы создадим двух заместителей: для обращения к «своему» объекту и для обращения к объекту PersonBean другого пользователя. Заместители будут определять, какие запросы возможны в каждой из этих ситуаций.

Для создания заместителей мы воспользуемся механизмом динамических заместителей Java API, который был представлен несколько страниц назад. Java сгенерирует заместителей автоматически; нам остается лишь предоставить обработчики, которые определяют действия при вызове методов заместителей.

Вспомните диаграмму, приведенную несколько страниц назад...



Понадобятся два таких объекта.

Сам заместитель создается во время выполнения.

Шаг 1:

Создание двух реализаций InvocationHandler. InvocationHandler реализует поведение заместителя. Как вскоре будет показано, Java создает класс и объект заместителя — нам остается лишь предоставить обработчик, который знает, что делать при вызове метода.

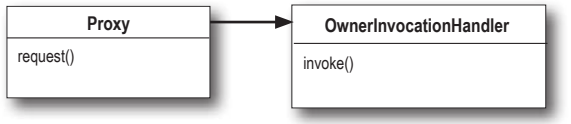
Шаг 2:

Напишите код создания динамического заместителя. В программу включается небольшой фрагмент кода, который генерирует класс заместителя и создает его экземпляр. Вскоре мы рассмотрим этот код более подробно.

Шаг 3:

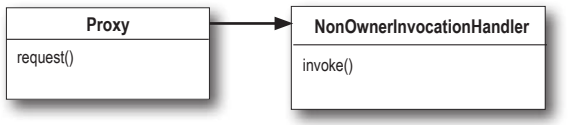
Каждый объект PersonBean упаковывается в соответствующего заместителя. Любой объект PersonBean описывает либо текущего пользователя (в этом случае мы будем называть его «владельцем»), либо другого пользователя службы знакомств.

В любом случае для PersonBean создается соответствующий заместитель.



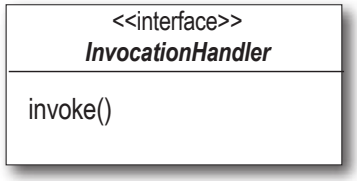
Пользователь просматривает свой объект.

Пользователь просматривает чужой объект.

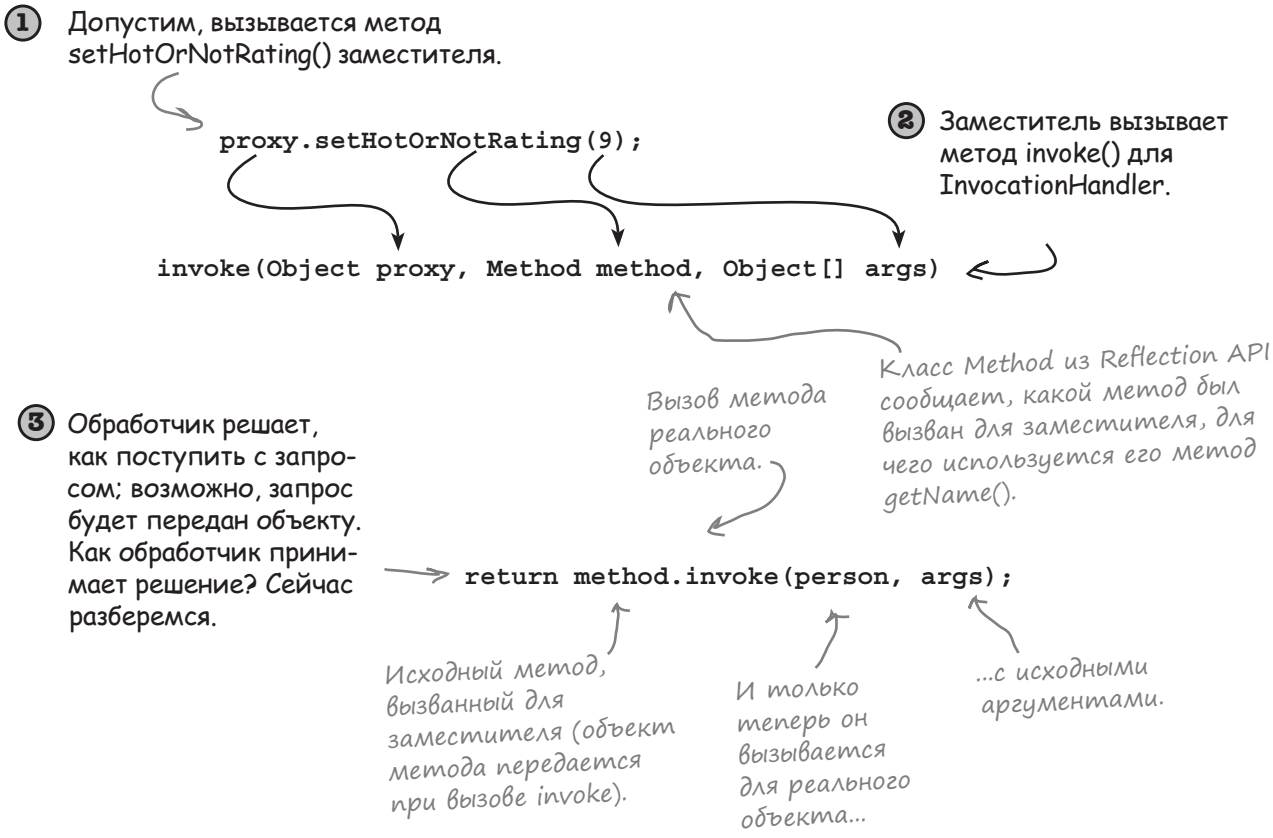


Шаг 1: создание реализаций Invocation Handler

Итак, мы должны написать два обработчика: для владельца и для других пользователей. Но что собой представляет обработчик? Когда клиент вызывает метод заместителя, последний перенаправляет этот вызов обработчику — но *не* вызывая соответствующий метод обработчика. Что же тогда он вызывает? Посмотрите на интерфейс InvocationHandler:



Он содержит единственный метод invoke(), и какой бы метод заместителя ни был вызван, это всегда приводит к вызову метода invoke() обработчика. Вот как работает эта схема:



Создание реализаций InvocationHandler (продолжение)...

Как же при вызове invoke() заместителя определить, что делать с этим вызовом? Обычно решение принимается на основании анализа имени метода, и, возможно, аргументов. Следующая реализация обработчика OwnerInvocationHandler показывает, как это делается:

```
import java.lang.reflect.*;

public class OwnerInvocationHandler implements InvocationHandler {
    PersonBean person;

    public OwnerInvocationHandler(PersonBean person) {
        this.person = person;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws IllegalAccessException {

        try {
            if (method.getName().startsWith("get")) {
                return method.invoke(person, args);
            } else if (method.getName().equals("setHotOrNotRating")) {
                throw new IllegalAccessException();
            } else if (method.getName().startsWith("set")) {
                return method.invoke(person, args);
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

Необходимо импортировать пакет java.lang.reflect, в котором определяется интерфейс InvocationHandler.

Все обработчики реализуют интерфейс InvocationHandler.

Передаем реальный объект в конструкторе и сохраняем ссылку на него.

При каждом вызове метода заместителя вызывается метод invoke.

Если вызван get-метод, вызов передается реальному объекту.

Вызов метода setHotOrNotRating() блокируется выдачей исключения IllegalAccessException.

Выполняется при выдаче исключения реальным объектом.

Вызов любых других set-методов владельцу разрешен, передаем реальному объекту.

При вызове любого другого метода мы просто возвращаем null, чтобы избежать лишнего риска.

Обработчик данных других пользователей NonOwnerInvocationHandler работает точно так же, как OwnerInvocationHandler, если не считать того, что он *разрешает* вызовы setHotOrNotRating() и *запрещает* вызовы всех остальных set-методов. Напишите код обработчика самостоятельно:

Шаг 2: создание класса и экземпляра заместителя

Остается лишь динамически сгенерировать класс заместителя и создать экземпляр объекта. Начнем с написания метода, который получает PersonBean и создает для него заместителя, предназначенного для владельца. Таким образом, мы создаем заместителя, который передает вызовы методов OwnerInvocationHandler.

Метод получает реальный объект (данные конкретного человека) и возвращает для него заместителя. Так как заместитель обладает тем же интерфейсом, что и реальный объект, мы возвращаем PersonBean.

Код, генерирующий заместителя. Мы последовательно разберем каждую из выполняемых операций.

Для создания заместителя используется статический метод newProxyInstance класса Proxy...

```
PersonBean getOwnerProxy(PersonBean person) {

    return (PersonBean) Proxy.newProxyInstance(
            person.getClass().getClassLoader(),
            person.getClass().getInterfaces(),
            new OwnerInvocationHandler(person));
}
```

Передаем ему загрузчик класса для нашего реального объекта...

...и набор интерфейсов, который должен реализовать заместитель...

Конструктору обработчика передается реальный объект (как было показано пару страниц назад, именно так обработчик получает доступ к реальному объекту).

...и обработчик вызова — в нашей ситуации OwnerInvocationHandler.

Возьми в руку карандаш

Код создания динамических заместителей нетривиален, но и сложным его никак не назовешь. Почему бы вам не написать собственную реализацию getNonOwnerProxy(), возвращающую заместителей для NonOwnerInvocationHandler?

Удастся ли вам написать общий метод getProxy(), который получает обработчик и объект PersonBean и возвращает соответствующего заместителя?

Тестирование службы знакомств

Тестовый запуск наглядно показывает, как доступ к set-методам ограничивается в зависимости от используемого заместителя:

```
public class MatchMakingTestDrive {
    // Переменные экземпляров

    public static void main(String[] args) {
        MatchMakingTestDrive test = new MatchMakingTestDrive();
        test.drive();
    }

    public MatchMakingTestDrive() {
        initializeDatabase();
    }
    public void drive() {
        PersonBean joe = getPersonFromDatabase("Joe Javabean");
        PersonBean ownerProxy = getOwnerProxy(joe);
        System.out.println("Name is " + ownerProxy.getName());
        ownerProxy.setInterests("bowling, Go");
        System.out.println("Interests set from owner proxy");
        try {
            ownerProxy.setHotOrNotRating(10);
        } catch (Exception e) {
            System.out.println("Can't set rating from owner proxy");
        }
        System.out.println("Rating is " + ownerProxy.getHotOrNotRating());

        PersonBean nonOwnerProxy = getNonOwnerProxy(joe);
        System.out.println("Name is " + nonOwnerProxy.getName());
        try {
            nonOwnerProxy.setInterests("bowling, Go");
        } catch (Exception e) {
            System.out.println("Can't set interests from non owner proxy");
        }
        nonOwnerProxy.setHotOrNotRating(3);
        System.out.println("Rating set from non owner proxy");
        System.out.println("Rating is " + nonOwnerProxy.getHotOrNotRating());
    }

    // Методы getOwnerProxy, getNonOwnerProxy и т.д.
}
```

Main просто создает тестовый сценарий и запускает тестирование вызовом drive().

Конструктор инициализирует базу данных кандидатов службы знакомств.

Чтение записи из БД...

...и создание заместителя для владельца.

Вызываем get-метод...

...затем set-метод...

... а затем пытаемся изменить оценку.

Не должно работать!

Теперь создаем заместителя для другого пользователя.

...вызываем get-метод...

...set-метод...

Не должно работать!

Теперь пытаемся задать оценку.

Должно работать!

Запуск кода...

```
File Edit Window Help Born2BDynamic
% java MatchMakingTestDrive
Name is Joe Javabean
Interests set from owner proxy
Can't set rating from owner proxy
Rating is 7

Name is Joe Javabean
Can't set interests from non owner proxy
Rating set from non owner proxy
Rating is 5
%
```

Заместитель владельца разрешает чтение и запись (кроме оценки).

Заместитель другого пользователя разрешает только чтение, а также запись только для оценки.

Новая оценка вычисляется как среднее арифметическое предыдущей оценки 7 и значения, заданного другим пользователем (3).

Часто Задаваемые Вопросы

В: Почему динамические заместители так называются? Потому, что я создаю экземпляр заместителя и связываю его с обработчиком во время выполнения?

О: Нет, потому, что класс заместителя создается во время выполнения. До выполнения вашего кода класс заместителя не существует; он строится по требованию на основании переданных ему интерфейсов.

В: Для заместителя InvocationHandler выглядит довольно странно — он не реализует ни один метод класса, который он представляет.

О: InvocationHandler не является заместителем — это класс, к которому обращается заместитель за обработкой вызовов методов. Сам заместитель строится динамически вызовом метода Proxy.newProxyInstance().

В: Как отличить класс динамического заместителя от других классов?

О: Класс Proxy содержит статический метод isProxyClass(). Вызов этого метода для класса вернет true, если класс является классом динамического заместителя. В остальном класс ведет себя точно так же, как любой другой класс, реализующий конкретный набор интерфейсов.

В: Существуют ли ограничения для типов интерфейсов, передаваемых newProxyInstance()?

О: Да, существуют. Во-первых, newProxyInstance() всегда передается массив интерфейсов — разрешены только интерфейсы, но не классы. Главное ограничение заключается в том, что все интерфейсы с уровнем доступа, отличным от public, должны принадлежать одному пакету. Кроме того, запрещены конфликты имен в интерфейсах (то есть интерфейсы с методами, имеющими одинаковую сигнатуру). Существуют и другие второстепенные ограничения; информацию о них можно найти в javadoc.

Кто и что делает?

Соедините каждый паттерн с его описанием:

Паттерн	Описание
Декоратор	Упаковывает другой объект и предоставляет другой интерфейс для работы с ним
Фасад	Упаковывает другой объект и предоставляет дополнительное поведение
Заместитель	Упаковывает другой объект для управления доступом к нему
Адаптер	Упаковывает группу объектов для упрощения их интерфейса

Разновидности заместителей

Добро пожаловать в зоопарк городка Объективиль!

Вы уже знакомы с удаленными, виртуальными и защитными заместителями, но в разных уголках нашего мира встречается немало других форм и разновидностей этого паттерна. В специальном уголке нашего зоопарка представлена коллекция типичных экспонатов, собранных здесь для расширения вашего кругозора.

Наверняка вы встретите и другие разновидности этого паттерна; надеемся, что вы поможете нам пополнить каталог новыми видами. А пока давайте познакомимся с коллекцией в ее текущем состоянии:

Фильтрующий Заместитель управляет доступом к группам сетевых ресурсов, защищая их от «недобросовестных» клиентов.

Среда обитания: часто встречается в корпоративных системах защиты данных.

Опишите среду обитания

Умная Ссылка обеспечивает выполнение дополнительных действий при обращении к объекту (например, изменение счетчика ссылок).

Кэширующий Заместитель обеспечивает временное хранение результатов высокозатратных операций. Также может обеспечивать совместный доступ к результатам для предотвращения лишних вычислений или пересылки данных по сети.

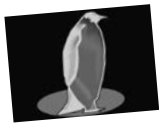
Среда обитания: часто встречается в веб-серверах, системах управления контентом и публикацией.

Встречается в окрестностях JavaSpaces; управляет синхронизацией доступа к группам объектов в распределенных средах.

Синхронизирующий заместитель предоставляет безопасный доступ к объекту из нескольких программных потоков.

Опишите среду обитания

Упрощающий заместитель скрывает сложность и управляет доступом к сложному набору классов. Иногда по очевидным соображениям называется Фасадным Заместителем. Упрощающий Заместитель отличается от паттерна Фасад тем, что первый управляет доступом, а второй только предоставляет альтернативный интерфейс.



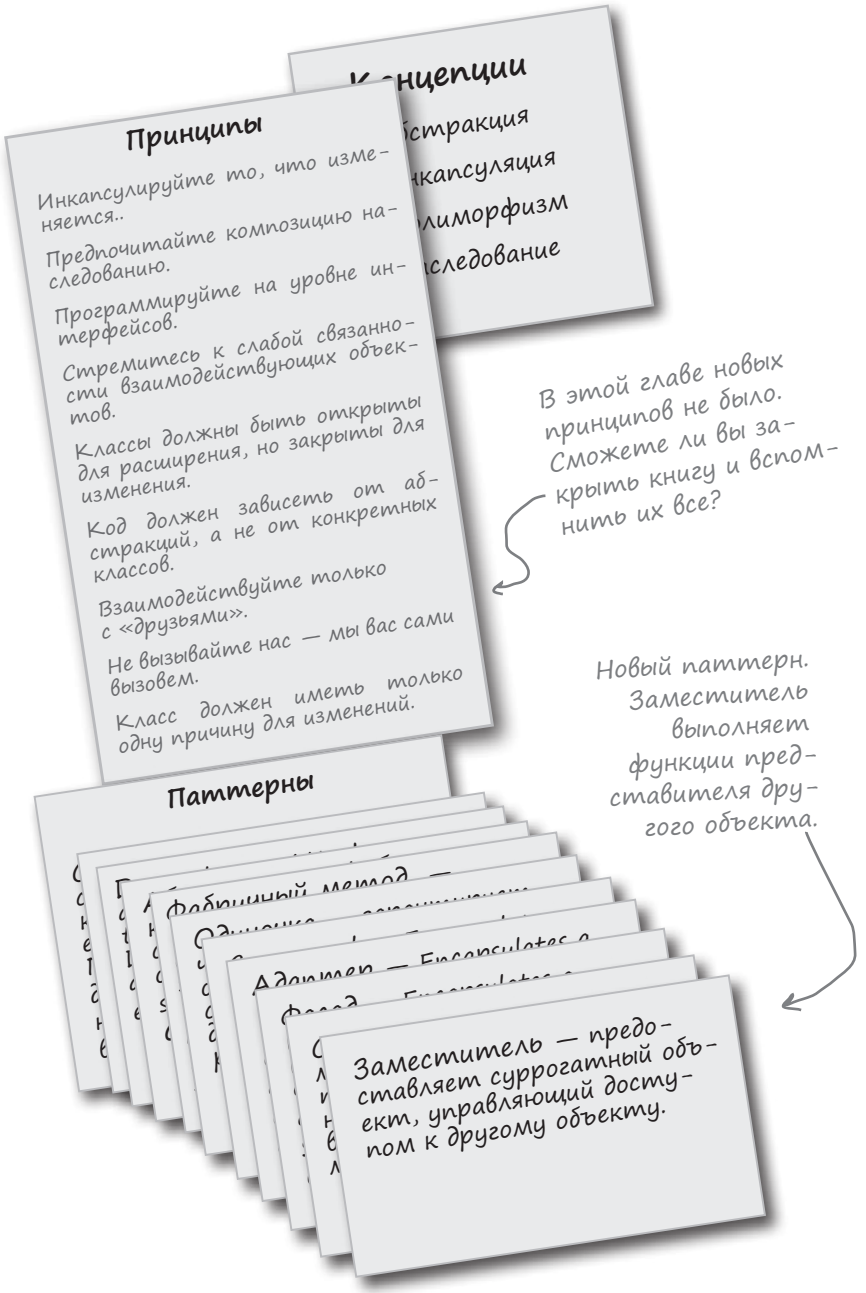
Заместитель Отложенного Копирования задерживает фактическое копирование объекта до момента выполнения операций с копией (разновидность Виртуального Заместителя).

Среда обитания: встречается в окрестностях CopyOnWriteArrayList (Java 5).

Запишите другие разновидности заместителей, с которыми вы столкнулись в естественной среде обитания:

Новые инструменты

Ваш инструментарий почти полон; у вас есть все необходимое для решения почти любой проблемы.



КЛЮЧЕВЫЕ МОМЕНТЫ

- Паттерн Заместитель предоставляет «суррогат» для управления доступом к другому объекту.
- Удаленный заместитель управляет взаимодействием клиента с удаленным объектом.
- Виртуальный заместитель управляет доступом к объекту, создание которого сопряжено с большими затратами.
- Защитный заместитель управляет доступом к методам объекта в зависимости от привилегий вызывающей стороны.
- Существует много других разновидностей паттерна Заместитель: кэширующий заместитель, синхронизирующий заместитель, фильтрующий заместитель и т. д.
- На структурном уровне паттерны Заместитель и Декоратор похожи, но они различаются по своим целям.
- Паттерн Декоратор расширяет поведение объекта, а Заместитель управляет доступом.
- Встроенная поддержка посредников в Java позволяет генерировать классы динамических заместителей во время выполнения и передавать все вызовы обработчику по вашему выбору.
- Заместители, как и любые «обертки», увеличивают количество классов и объектов в архитектуре.

Возьми в руку карандаш

Решение

Обработчик данных других пользователей NonOwnerInvocationHandler работает точно так же, как OwnerInvocationHandler, если не считать того, что он *разрешает* вызовы setHotOrNotRating() и *запрещает* вызовы всех остальных set-методов. Напишите код обработчика самостоятельно:

```
import java.lang.reflect.*;

public class NonOwnerInvocationHandler implements InvocationHandler {
    PersonBean person;

    public NonOwnerInvocationHandler(PersonBean person) {
        this.person = person;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws IllegalAccessException {

        try {
            if (method.getName().startsWith("get")) {
                return method.invoke(person, args);
            } else if (method.getName().equals("setHotOrNotRating")) {
                return method.invoke(person, args);
            } else if (method.getName().startsWith("set")) {
                throw new IllegalAccessException();
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

Головоломка

Решение

Похоже, класс ImageProxy обладает двумя состояниями, причем нужное состояние выбирается при помощи условной конструкции. Можете ли вы предложить другой паттерн для упрощения этого кода? Как бы вы переработали ImageProxy в контексте этого паттерна?

Используйте паттерн Состояние: реализуйте два состояния (ImageLoaded и ImageNotLoaded) и переместите код из конструкций if в соответствующие состояния. Начните с состояния ImageNotLoaded, а затем перейдите в состояние ImageLoaded после получения данных ImageIcon.

Возьми в руку карандаш
Решение

Код создания динамических заместителей нетривиален, но и сложным его никак не назовешь. Почему бы вам не написать собственную реализацию getNonOwnerProxy(), возвращающую заместителя для NonOwnerInvocationHandler?

```
PersonBean getNonOwnerProxy(PersonBean person) {

    return (PersonBean) Proxy.newProxyInstance(
            person.getClass().getClassLoader(),
            person.getClass().getInterfaces(),
            new NonOwnerInvocationHandler(person));
}
```

Кто и что делает?
Решение

Соедините каждый паттерн с его описанием:

Готово к употреблению

Код CD Cover Viewer

```
package headfirst.designpatterns.proxy.virtualproxy;

import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
public class ImageProxyTestDrive {
    ImageComponent imageComponent;
    JFrame frame = new JFrame("CD Cover Viewer");
    JMenuBar menuBar;
    JMenu menu;
    Hashtable<String, String> cds = new Hashtable<String, String>();

    public static void main (String[] args) throws Exception {
        ImageProxyTestDrive testDrive = new ImageProxyTestDrive();
    }

    public ImageProxyTestDrive() throws Exception{
        cds.put("Buddha Bar","http://images.amazon.com/images/P/B00009XBYK.01.LZZZZZZZ.
    jpg");
        cds.put("Ima","http://images.amazon.com/images/P/B000005IRM.01.LZZZZZZZ.jpg");
        cds.put("Karma","http://images.amazon.com/images/P/B000005DCB.01.LZZZZZZZ.gif");
        cds.put("MCMXC A.D.","http://images.amazon.com/images/P/B000002URV.01.LZZZZZZZ.
    jpg");
        cds.put("Northern Exposure","http://images.amazon.com/images/P/B000003SFN.01.
    LZZZZZZZ.jpg");
        cds.put("Selected Ambient Works, Vol. 2","http://images.amazon.com/images/P/
    B000002MNZ.01.LZZZZZZZ.jpg");

        URL initialURL = new URL((String)cds.get("Selected Ambient Works, Vol. 2"));
        menuBar = new JMenuBar();
        menu = new JMenu("Favorite CDs");
        menuBar.add(menu);
        frame.setJMenuBar(menuBar);
```

Готово к употреблению

Код CD Cover Viewer (продолжение)

```
        for(Enumeration e = cds.keys(); e.hasMoreElements();) {
            String name = (String)e.nextElement();
            JMenuItem menuItem = new JMenuItem(name);
            menu.add(menuItem);
            menuItem.addActionListener(event -> {
                imageComponent.setIcon(new ImageProxy(getCDUrl(event.getActionCommand())));
                frame.repaint();
            });
        }

        // set up frame and menus

        Icon icon = new ImageProxy(initialURL);
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(800,600);
        frame.setVisible(true);

    }
    URL getCDUrl(String name) {
        try {
            return new URL((String)cds.get(name));
        } catch (MalformedURLException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

```
package headfirst.designpatterns.proxy.virtualproxy;

import java.net.*;
import java.awt.*;
import javax.swing.*;

class ImageProxy implements Icon {
    volatile ImageIcon imageIcon;
    final URL imageURL;
    Thread retrievalThread;
    boolean retrieving = false;

    public ImageProxy(URL url) { imageURL = url; }

    public int getIconWidth() {
    if (imageIcon != null) {
        return imageIcon.getIconWidth();
    } else {
        return 800;
    }
}

public int getIconHeight() {
    if (imageIcon != null) {
        return imageIcon.getIconHeight();
    } else {
        return 600;
    }
}

synchronized void setImageIcon(ImageIcon imageIcon) {
    this.imageIcon = imageIcon;
}

public void paintIcon(final Component c, Graphics g, int x, int y) {
    if (imageIcon != null) {
        imageIcon.paintIcon(c, g, x, y);
    } else {
        g.drawString("Loading CD cover, please wait...", x+300, y+190);
        if (!retrieving) {
            retrieving = true;
```

Код CD Cover Viewer (продолжение)

```
                retrievalThread = new Thread(new Runnable() {
                    public void run() {
                        try {
                            setImageIcon(new ImageIcon(imageURL, "CD Cover"));
                            c.repaint();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                });
                retrievalThread.start();
            }
        }
    }
}
package headfirst.designpatterns.proxy.virtualproxy;

import java.awt.*;
import javax.swing.*;

class ImageComponent extends JComponent {
    private Icon icon;

    public ImageComponent(Icon icon) {
        this.icon = icon;
    }

    public void setIcon(Icon icon) {
        this.icon = icon;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int w = icon.getIconWidth();
        int h = icon.getIconHeight();
        int x = (800 - w)/2;
        int y = (600 - h)/2;
        icon.paintIcon(this, g, x, y);
    }
}
```

Паттерны паттернов

Кто бы мог предположить, что паттерны порой работают рука *об* руку? Вы уже были свидетелями ожесточенных перепалок в «Беседах у камина» (причем вы не видели «Смертельные поединки» паттернов — редактор заставил нас исключить их из книги!). И после этого оказывается, что мирное сосуществование все же возможно. Хотите верьте, хотите нет, но некоторые из самых мощных ОО-архитектур строятся на основе комбинаций нескольких паттернов.

Совместная работа паттернов

Чтобы паттерны проявили себя в полной мере, не ограничивайте их жесткими рамками и позвольте им взаимодействовать с другими паттернами. Чем шире вы используете паттерны, тем больше вероятность их объединения в ваших архитектурах. Для таких комбинаций паттернов, применимых в разных задачах, существует специальный термин. *составные паттерны*. Да, именно так: речь идет о паттернах, составленных из других паттернов!

Составные паттерны очень часто применяются в реальном программировании. Теперь, когда основные паттерны твердо запечатлелись в вашей памяти, вы сразу увидите, что составные паттерны — всего лишь комбинация уже известных вам паттернов.

В начале этой главы мы вернемся к старому примеру с утками. Утки помогут вам понять, как паттерны объединяются в одном решении. Однако совмещение паттернов еще не означает, что решение может быть отнесено к категории составных паттернов — для этого оно должно иметь общий характер и быть применимым для решения многих задач. Таким образом, во второй половине главы мы рассмотрим *настоящий* составной паттерн Модель-Представление-Контроллер (MVC). Если вы еще не слышали о нем, знайте: он станет одним из самых мощных паттернов в вашем инструментарии.

Паттерны часто используются вместе и комбинируются в реализациях проектировочных решений.

Составной паттерн объединяет два и более паттерна для решения распространенной или общей проблемы.

И снова утки

Как вы уже знаете, мы снова будем работать с утками. На этот раз они покажут вам, как паттерны могут сосуществовать — и даже сотрудничать — в одном решении.

Мы заново построим «утиную» программу и наделим ее интересными возможностями при помощи нескольких паттернов. Итак, за дело...

1 **Начинаем с создания интерфейса Quackable.**

Напомним, что программа будет написана «с нуля». На этот раз объекты Duck будут реализовывать интерфейс Quackable. От реализации этого интерфейса зависит поддержка метода quack() разными классами.

```
public interface Quackable {
    public void quack();
}
```

Интерфейс Quackable состоит из единственного метода quack().

2 **Переходим к объектам Duck, реализующим Quackable.**

Какой прок от интерфейса без реализующих его классов. Пора создать несколько конкретных разновидностей уток («газетные» не в счет).

Стандартная утка-кряква (Mallard).

```
public class MallardDuck implements Quackable {
    public void quack() {
        System.out.println("Quack");
    }
}
```

```
public class RedheadDuck implements Quackable {
    public void quack() {
        System.out.println("Quack");
    }
}
```

Чтобы программа получилась интересной, нужно добавить другие разновидности.

Но если все утки будут вести себя одинаково, программа получится неинтересной.

Помните предыдущую версию? В нее входили утиные манки DuckCall (те штуки, которыми пользуются охотники, — они определенно умеют крякать) и резиновые утки RubberDuck:

```
public class DuckCall implements Quackable {
    public void quack() {
        System.out.println("Kwak");
    }
}
```

Звук утиного манка несколько отличается от кряканья настоящей утки.

```
public class RubberDuck implements Quackable {
    public void quack() {
        System.out.println("Squeak");
    }
}
```

Резиновая утка пищит вместо того, чтобы нормально крякать.

3 **Утки есть, теперь нужен имитатор.**

Напишем небольшую программу, которая создает несколько уток и заставляет их крякать...

Метод main запускает имитатор.

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }

    void simulate() {
        Quackable mallardDuck = new MallardDuck();
        Quackable redheadDuck = new RedheadDuck();
        Quackable duckCall = new DuckCall();
        Quackable rubberDuck = new RubberDuck();

        System.out.println("\nDuck Simulator");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Создаем объект и вызываем его метод simulate().

Создаем по одному экземпляру для каждой реализации Quackable...

...и запускаем имитацию для каждой из них.

Перегруженная версия метода simulate имитирует одну утку.

Здесь мы полагаемся на волшебство полиморфизма: какая бы реализация Quackable ни была передана, simulate() вызовет ее метод quack().

Пока не впечатляет, но мы еще не добавили паттерны!

```
File Edit Window Help ItBetterGetBetterThanThis
% java DuckSimulator

Duck Simulator
Quack
Quack
Kwak
Squeak

%
```

Все объекты реализуют один интерфейс Quackable, но каждый делает это по-своему.

Вроде все работает, идем дальше.

4 **Там, где есть утки, найдется место и гусям.**
Класс Goose представляет другую водоплавающую птицу — гуся:

```
public class Goose {
    public void honk() {
        System.out.println("Honk");
    }
}
```

Гуси не крякают — вместо quack() они реализуют метод honk().

МОЗГОВОЙ ШТУРМ

Допустим, нужно, чтобы объект Goose мог использоваться везде, где могут использоваться объекты Duck. В конце концов, гуси тоже издают звуки, умеют летать и плавать. Так почему бы не включить их в программу?

Какой паттерн позволит легко смешать гусей с утками?

5 **Нам понадобится адаптер.**

Программа работает с реализациями Quackable. Поскольку гуси не поддерживают метод quack(), мы можем воспользоваться адаптером для превращения гуся в утку:

```
public class GooseAdapter implements Quackable {
    Goose goose;

    public GooseAdapter(Goose goose) {
        this.goose = goose;
    }

    public void quack() {
        goose.honk();
    }
}
```

Напоминаем: паттерн Адаптер реализует целевой интерфейс, которым в данном случае является Quackable.

Конструктор получает адаптируемый объект Goose.

Вызов quack() делегируется методу honk() класа Goose.

6 **Теперь гуси тоже смогут участвовать в нашей имитации.**

Нужно лишь создать объект Goose, упаковать его в адаптер, реализующий Quackable — и можно работать.

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }
    void simulate() {
        Quackable mallardDuck = new MallardDuck();
        Quackable redheadDuck = new RedheadDuck();
        Quackable duckCall = new DuckCall();
        Quackable rubberDuck = new RubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Goose Adapter");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Создаем объект Goose, замаскированный под Duck; для этого Goose упаковывается в GooseAdapter.

С адаптированным объектом Goose можно работать, как с обычным объектом Duck, реализующим Quackable.

7 **Пробный запуск....**

На этот раз список объектов, передаваемых методу simulate(), содержит объект Goose, упакованный в «утиный» адаптер. Результат? Мы видим вызовы honk()!

Вызов honk()! Теперь объекты Goose издают звуки вместе с объектами Duck.

```
File Edit Window Help GoldenEggs
% java DuckSimulator

Duck Simulator: With Goose Adapter
Quack
Quack
Kwak
Squeak
Honk

%
```

Утковедение

Ученые-утковеды в восторге от всех аспектов поведения Quackable. В частности, они всегда мечтали подсчитать общее количество кряков, издаваемых утиной стаей.

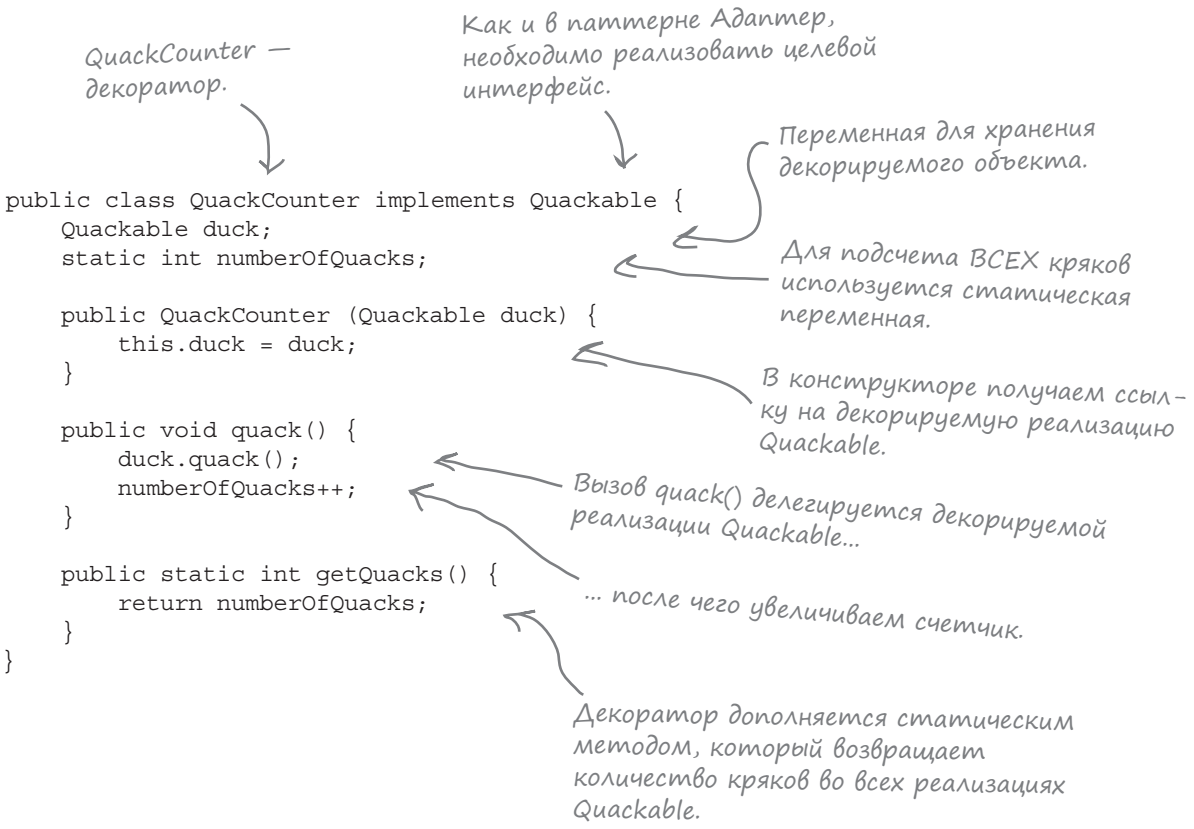
Как реализовать возможность подсчета без изменения классов уток?

Какой паттерн нам в этом поможет?

Дж. Брюэр, смотритель парка и знатный утковед.

8 **Мы осчастливим этих утковедов и реализуем механизм подсчета.**

Как это сделать? Мы наделим уток новым поведением (подсчет), упаковывая их в объекте-декораторе. Изменять код Duck для этого не придется.




```
public class QuackCounter implements Quackable {
    Quackable duck;
    static int numberOfQuacks;

    public QuackCounter (Quackable duck) {
        this.duck = duck;
    }

    public void quack() {
        duck.quack();
        numberOfQuacks++;
    }

    public static int getQuacks() {
        return numberOfQuacks;
    }
}
```

9 **Обновляем программу, чтобы в ней создавались декорированные реализации Quackable.**

Теперь каждый объект Quackable, экземпляр которого создается в программе, должен упаковываться в декоратор QuackCounter. Если этого не сделать, вызовы quack() не будут учитываться при подсчете.

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }
    void simulate() {
        Quackable mallardDuck = new QuackCounter(new MallardDuck());
        Quackable redheadDuck = new QuackCounter(new RedheadDuck());
        Quackable duckCall = new QuackCounter(new DuckCall());
        Quackable rubberDuck = new QuackCounter(new RubberDuck());
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Decorator");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);

        System.out.println("The ducks quacked " +
                           QuackCounter.getQuacks() + " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Каждая вновь создаваемая реализация Quackable упаковывается в декоратор.

Гусиные крики ученых не интересуют, поэтому объекты Goose не декорируются.

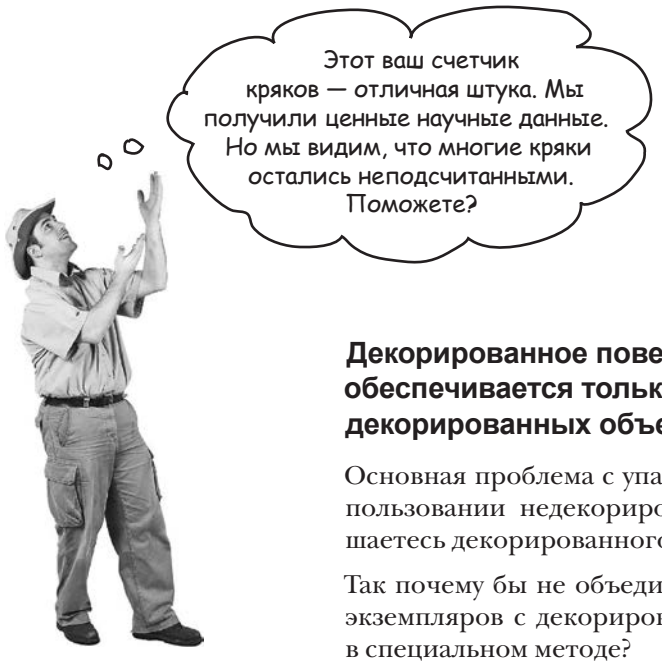
Вывод данных, собранных для утковедов.

Ничего не изменилось: декорированные объекты остаются реализациями Quackable.

Результаты

Напоминаем, что гусиные крики не подсчитываются.

```
File  Edit  Window  Help  DecoratedEggs
% java DuckSimulator
Duck Simulator: With Decorator
Quack
Quack
Kwak
Squeak
Honk
4 quacks were counted
%
```



Декорированное поведение обеспечивается только для декорированных объектов.

Основная проблема с упаковкой объектов: при использовании недекорированного объекта вы лишаетесь декорированного поведения.

Так почему бы не объединить операции создания экземпляров с декорированием, инкапсулируя их в специальном методе?

Не напоминает ли это вам какой-нибудь паттерн?

10 **Нужна фабрика для производства уток!**

Мы должны позаботиться о том, чтобы все утки обязательно были упакованы в декоратор. Для создания декорированных уток будет построена целая фабрика! Она должна производить целое семейство продуктов, состоящее из разных утиных разновидностей, поэтому мы воспользуемся паттерном Абстрактная Фабрика.

Начнем с определения AbstractDuckFactory:

Мы определяем абстрактную фабрику, которая будет реализовываться субклассами для создания разных продуктов.

```
public abstract class AbstractDuckFactory {

    public abstract Quackable createMallardDuck();
    public abstract Quackable createRedheadDuck();
    public abstract Quackable createDuckCall();
    public abstract Quackable createRubberDuck();
}
```

Каждый метод создает одну из разновидностей уток.

Первая версия фабрики создает уток без декораторов — просто для того, чтобы вы лучше усвоили, как работает фабрика:

```
public class DuckFactory extends AbstractDuckFactory {

    public Quackable createMallardDuck() {
        return new MallardDuck();
    }

    public Quackable createRedheadDuck() {
        return new RedheadDuck();
    }

    public Quackable createDuckCall() {
        return new DuckCall();
    }

    public Quackable createRubberDuck() {
        return new RubberDuck();
    }
}
```

DuckFactory расширяет абстрактную фабрику.

Каждый метод создает продукт: конкретную разновидность утки. Программа не знает фактический класс создаваемого продукта — ей известно лишь то, что создается реализация Quackable.

А теперь перейдем к той фабрике с декораторами, которая нам нужна:

CountingDuckFactory также расширяет класс абстрактной фабрики.

```
public class CountingDuckFactory extends AbstractDuckFactory {

    public Quackable createMallardDuck() {
        return new QuackCounter(new MallardDuck());
    }

    public Quackable createRedheadDuck() {
        return new QuackCounter(new RedheadDuck());
    }

    public Quackable createDuckCall() {
        return new QuackCounter(new DuckCall());
    }

    public Quackable createRubberDuck() {
        return new QuackCounter(new RubberDuck());
    }
}
```

Каждый метод упаковывает Quackable в декоратор. Программа этого не замечает: она получает то, что ей нужно, т. е. реализацию Quackable. Но теперь ученые могут быть твердо уверены в том, что каждый кряк будет подсчитан.

⑪ Переведем имитатор на использование фабрики.

Помните, как работает паттерн Абстрактная Фабрика? Мы создаем полиморфный метод, который получает фабрику и использует ее для создания объектов. Передавая разные фабрики, мы можем создавать разные семейства продуктов в одном методе.

Новая версия метода simulate() получает фабрику и использует ее для создания уток.

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        AbstractDuckFactory duckFactory = new CountingDuckFactory();

        simulator.simulate(duckFactory);
    }

    void simulate(AbstractDuckFactory duckFactory) {
        Quackable mallardDuck = duckFactory.createMallardDuck();
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        Quackable duckCall = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Abstract Factory");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);

        System.out.println("The ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Сначала создаем фабрику, которая будет передаваться методу simulate().

Метод simulate() получает AbstractDuckFactory и использует фабрику для создания уток (вместо непосредственного создания экземпляров).

Здесь ничего не изменилось!

Результат с использованием фабрики...

То же, что в предыдущей версии — но на этот раз каждая утка заведомо декорируется, потому что мы используем CountingDuckFactory.

```
File Edit Window Help EggFactory
% java DuckSimulator
Duck Simulator: With Abstract Factory
Quack
Quack
Kwak
Squeak
Honk
4 quacks were counted
%
```

Возьми в руку карандаш

Однако экземпляры гусей по-прежнему создаются непосредственно, а код зависит от конкретных классов. Удастся ли вам написать абстрактную фабрику для гусей? Как она должна создавать «гусей, замаскированных под уток»?

Теперь он хочет управлять утиными стаями.

Хороший вопрос: а почему мы работаем с разными утками по отдельности?

Не очень-то удобно!

```
Quackable mallardDuck = duckFactory.createMallardDuck();
Quackable redheadDuck = duckFactory.createRedheadDuck();
Quackable duckCall = duckFactory.createDuckCall();
Quackable rubberDuck = duckFactory.createRubberDuck();
Quackable gooseDuck = new GooseAdapter(new Goose());

simulate(mallardDuck);
simulate(redheadDuck);
simulate(duckCall);
simulate(rubberDuck);
simulate(gooseDuck);
```

Необходим механизм создания коллекций уток — и даже субколлекций (раз уж заказчик говорит о «семействах»). Также было бы желательно иметь возможность применять операции ко всему множеству уток.

Какой паттерн нам в этом поможет?

12 **Создаем утиную стаю (а точнее стаю реализаций Quackable).**

Помните паттерн Компоновщик — тот, что позволял интерпретировать коллекцию объектов как отдельный объект? Применим его к реализациям Quackable!

Вот как это делается:

Комбинация должна реализовать тот же интерфейс, что и листовые элементы (Quackable в нашем примере).

Каждая стая (Flock) содержит ArrayList для хранения реализаций Quackable, входящих в эту стаю.

```
public class Flock implements Quackable {
    ArrayList<Quackable> quackers = new ArrayList<Quackable>();

    public void add(Quackable quacker) {
        quackers.add(quacker);
    }

    public void quack() {
        Iterator<Quackable> iterator = quackers.iterator();
        while (iterator.hasNext()) {
            Quackable quacker = iterator.next();
            quacker.quack();
        }
    }
}
```

Метод add() включает реализацию Quackable в Flock.

Теперь метод quack() — в конце концов, Flock тоже является реализацией Quackable. Метод quack() для объекта Flock должен применяться ко всем уткам стаи. Мы перебираем элементы ArrayList и вызываем quack() для каждого элемента.

Код под увеличительным стеклом

А вы заметили, что мы тайком подсунули вам еще один паттерн, ни словом не упомянув о нем?

```
public void quack() {
    Iterator<Quackable> iterator = quackers.iterator();
    while (iterator.hasNext()) {
        Quackable quacker = iterator.next();
        quacker.quack();
    }
}
```

Вот он! Паттерн Итератор!

13 **Теперь нужно изменить программу.**

Осталось добавить немного кода, чтобы утки вписывались в новую структуру.

```
public class DuckSimulator {
    // Метод main

    void simulate(AbstractDuckFactory duckFactory) {
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        Quackable duckCall = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());
        System.out.println("\nDuck Simulator: With Composite – Flocks");

        Flock flockOfDucks = new Flock();

        flockOfDucks.add(redheadDuck);
        flockOfDucks.add(duckCall);
        flockOfDucks.add(rubberDuck);
        flockOfDucks.add(gooseDuck);

        Flock flockOfMallards = new Flock();

        Quackable mallardOne = duckFactory.createMallardDuck();
        Quackable mallardTwo = duckFactory.createMallardDuck();
        Quackable mallardThree = duckFactory.createMallardDuck();
        Quackable mallardFour = duckFactory.createMallardDuck();

        flockOfMallards.add(mallardOne);
        flockOfMallards.add(mallardTwo);
        flockOfMallards.add(mallardThree);
        flockOfMallards.add(mallardFour);

        flockOfDucks.add(flockOfMallards);

        System.out.println("\nDuck Simulator: Whole Flock Simulation");
        simulate(flockOfDucks);

        System.out.println("\nDuck Simulator: Mallard Flock Simulation");
        simulate(flockOfMallards);

        System.out.println("\nThe ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Создание реализаций Quackables (как и прежде).

Создаем объект Flock и заполняем его реализациями Quackable.

Затем создаем новый объект Flock, предназначенный только для крякв (MallardDuck).

Создаем несколько объектов...

...и добавляем их в контейнер Flock.

А теперь стая крякв добавляется в основную стаю.

Сначала тестируем всю стаю!

А теперь — только стаю крякв.

Данные для утковедов.

Ничего менять не нужно — Flock является реализацией Quackable!

Запускаем...

```
File Edit Window Help FlockADuck
% java DuckSimulator
Duck Simulator: With Composite – Flocks
Duck Simulator: Whole Flock Simulation
Quack
Kwak
Squeak
Honk
Quack
Quack
Quack
Quack

Duck Simulator: Mallard Flock Simulation
Quack
Quack
Quack
Quack

The ducks quacked 11 times
```

Первый объект Flock.

Второй объект Flock.

Вроде все хорошо (помните: гуси не в счет!)

Безопасность и прозрачность

Возможно, вы помните, что в главе, посвященной паттерну Компоновщик, листовые узлы (MenuItem) и комбинации (Menu) имели *совпадающие* наборы методов. Из-за этого для MenuItem могли вызываться методы, не имеющие смысла (например, метод add()). Преимуществом такого подхода была *прозрачность*: клиенту не нужно было знать, имеет ли он дело с листом или комбинацией — он просто вызывал одни и те же методы в обеих ситуациях.

Здесь мы решили отделить методы комбинаций, относящиеся к управлению дочерними элементами, от листовых узлов: другими словами, только объекты Flock содержат метод add(). Мы знаем, что вызов add() для Duck не имеет смысла и в данной реализации он невозможен. Метод add() может вызываться только для Flock. Такая архитектура обладает большей *безопасностью* (клиент не сможет вызывать для компонентов бессмысленные методы), но она менее прозрачна.

В ходе ОО-проектирования часто приходится принимать компромиссные решения. Учитывайте их при проектировании своих архитектур.

Паттерн Компоновщик отлично работает! Теперь у нас противоположная просьба: мы хотим наблюдать за отдельными утками. Как нам получать информацию о них в реальном времени?

«Наблюдать», говорите?

Похоже, наш знакомый хочет наблюдать за поведением отдельных уток. Этот путь ведет нас прямиком к паттерну, предназначенному для наблюдения за поведением объектов. Конечно, речь идет о паттерне Наблюдатель.

14 **Сначала определяем интерфейс наблюдаемого объекта.**

Напомним, что интерфейс наблюдаемого объекта содержит методы регистрации и оповещения наблюдателей. Также можно включить в него метод удаления наблюдателей, но мы опустим его, чтобы по возможности упростить реализацию.

Чтобы за Quackable можно было наблюдать, они должны реализовать интерфейс QuackObservable.

```
public interface QuackObservable {
    public void registerObserver(Observer observer);
    public void notifyObservers();
}
```

Метод регистрации наблюдателей. Любой объект, реализующий интерфейс Observer, сможет получать оповещения. Определение интерфейса Observer приводится ниже.

Также имеется метод оповещения наблюдателей.

Проследите за тем, чтобы интерфейс был реализован всеми Quackable...

```
public interface Quackable extends QuackObservable {
    public void quack();
}
```

Таким образом, интерфейс Quackable расширяет QuackObservable.

15 **Теперь нужно позаботиться о том, чтобы все конкретные классы, реализующие Quackable, могли использоваться в качестве QuackObservable.**

Задачу можно решить, реализуя регистрацию и оповещение в каждом классе (как было сделано в главе 2). На этот раз мы поступим немного иначе: инкапсулируем код регистрации и оповещения в другом классе — Observable — и объединим его с QuackObservable. В этом случае реальный код пишется только один раз, а в QuackObservable достаточно включить код делегирования вызовов вспомогательному классу Observable.

Начнем со вспомогательного класса Observable...

Observable реализует всю функциональность, необходимую Quackable для наблюдения.

Класс Observable должен реализовать QuackObservable.

```
public class Observable implements QuackObservable {
    ArrayList<Observer> observers = new ArrayList<Observer>();
    QuackObservable duck;

    public Observable(QuackObservable duck) {
        this.duck = duck;
    }

    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers() {
        Iterator iterator = observers.iterator();
        while (iterator.hasNext()) {
            Observer observer = iterator.next();
            observer.update(duck);
        }
    }
}
```

Конструктору передается объект QuackObservable, который используется им для управления наблюдением. Посмотрите на приведенный ниже метод notify(): при оповещении Observable передает этот объект, чтобы наблюдатель знал, в каком объекте произошло наблюдаемое событие.

Код регистрации наблюдателя.

Код оповещения.

Посмотрим, как Quackable использует этот вспомогательный класс...

16 **Интеграция вспомогательного класса Observable с классами Quackable.**

Задача не такая уж сложная. Все, что нужно, — это позаботиться о том, чтобы классы Quackable содержали комбинированные объекты Observable и умели делегировать им операции. После этого они готовы к использованию в качестве Observable. Ниже приводится реализация MallardDuck; остальные классы выглядят аналогично.

```
public class MallardDuck implements Quackable {
    Observable observable;

    public MallardDuck() {
        observable = new Observable(this);
    }

    public void quack() {
        System.out.println("Quack");
        notifyObservers();
    }

    public void registerObserver(Observer observer) {
        observable.registerObserver(observer);
    }

    public void notifyObservers() {
        observable.notifyObservers();
    }
}
```

Каждая реализация Quackable содержит объект Observable.

В конструкторе создаем объект Observable и передаем ему ссылку на объект MallardDuck.

Наблюдатели оповещаются о вызовах quack().

Два метода QuackObservable. Обратите внимание на делегирование операций вспомогательному объекту.

Возьми в руку карандаш

Мы еще не изменили реализацию декоратора QuackCounter. Его тоже необходимо интегрировать с Observable. Почему бы вам не сделать это самостоятельно?

17 **Работа почти завершена! Осталось разобраться с кодом на стороне Observer.**

Мы реализовали все необходимое для наблюдаемых объектов; теперь нужны наблюдатели. Начнем с интерфейса Observer:

Интерфейс Observer состоит из единственного метода update(), которому передается реализация QuackObservable.

```
public interface Observer {
    public void update(QuackObservable duck);
}
```

Теперь нужен наблюдатель:

Наблюдатель должен реализовать интерфейс Observer, иначе его не удастся зарегистрировать с QuackObservable.

```
public class Quackologist implements Observer {

    public void update(QuackObservable duck) {
        System.out.println("Quackologist: " + duck + " just quacked.");
    }
}
```

Класс наблюдателя прост: единственный метод update() выводит информацию о реализации Quackable, от которой поступило оповещение.

Возьми в руку карандаш

А если наблюдатель захочет понаблюдать за целой стаей? Как вообще это следует понимать? Понимайте так: наблюдение за комбинацией означает наблюдение за *каждым* ее элементом. Таким образом, при регистрации комбинация Flock должна позаботиться о регистрации всех своих дочерних элементов, среди которых могут быть другие комбинации.

Прежде чем двигаться дальше, напишите код наблюдения для Flock...

18 **Все готово к наблюдению. Давайте обновим программу и опробуем ее в деле:**

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        AbstractDuckFactory duckFactory = new CountingDuckFactory();

        simulator.simulate(duckFactory);
    }

    void simulate(AbstractDuckFactory duckFactory) {

        // Создание фабрик и объектов Duck

        // Создание объектов Flock

        System.out.println("\nDuck Simulator: With Observer");
        Quackologist quackologist = new Quackologist();
        flockOfDucks.registerObserver(quackologist);

        simulate(flockOfDucks);

        System.out.println("\nThe ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Создаем объект Quackologist и назначаем его наблюдателем для Flock.

На этот раз simulate() выполняется для Flock.

Смотрим, что же получилось!

Приближается грандиозный финал. Пять... нет, шесть паттернов объединились для создания потрясающего имитатора утиного пруда!

```
File  Edit  Window  Help  DucksAreEverywhere

% java DuckSimulator
Duck Simulator: With Observer
Quack
Quackologist: Redhead Duck just quacked.
Kwak
Quackologist: Duck Call just quacked.
Squeak
Quackologist: Rubber Duck just quacked.
Honk
Quackologist: Goose pretending to be a Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
Quack
Quackologist: Mallard Duck just quacked.
The Ducks quacked 7 times.

%
```

После каждого вызова quack() (независимо от его реализации!) наблюдатель получает оповещение.

А наблюдатель получает правильные данные.

В: Так это был составной паттерн?

О: Нет, это был простой пример совместного использования паттернов. Составной паттерн представляет собой набор паттернов, объединенных для решения типичной задачи. Например, рассматриваемый ниже паттерн Модель-Представление-Контроллер снова и снова используется во многих проектировочных решениях.

Часто Задаваемые Вопросы

В: Выходит, я беру задачу и начинаю применять к ней паттерны, пока не приду к решению. Верно?

О: Неверно. Мы привели этот пример, чтобы показать, что паттерны *могут* использоваться совместно. Никогда не следуйте нашему примеру при проектировании. В некоторых аспектах архитектуры нашего примера применение паттернов было явным перебором. Иногда задача решается простым соблюдением принципов ОО-проектирования.

Мы подробнее поговорим на эту тему в следующей главе, но паттерны следует применять только тогда, когда это оправданно. Не старайтесь применять паттерны только ради того, чтобы они присутствовали в вашей архитектуре.

Что мы сделали?

Мы начали с реализаций Quackable...

Сначала потребовалось, чтобы класс Goose тоже мог использоваться в качестве Quackable. Мы воспользовались паттерном Адаптер, чтобы преобразовать Goose в Quackable.

Затем было решено, что вызовы quack() должны подсчитываться. Мы воспользовались паттерном Декоратор и добавили декоратор QuackCounter.

Но клиент мог забыть добавить декоратор QuackCounter к объекту. Мы воспользовались паттерном Абстрактная Фабрика для создания экземпляров. Теперь каждый раз, когда клиент хотел создать объект Duck, он запрашивал его у фабрики и получал объект вместе с декоратором (а если ему нужен был объект без декоратора – использовал другую фабрику).

У нас возникли проблемы управления многочисленными объектами Duck, Goose и Quackable. Мы воспользовались паттерном Компоновщик для группировки объектов в коллекции. Паттерн также позволяет создавать субколлекции для управления подмножествами объектов. В нашей реализации также был задействован паттерн Итератор (мы воспользовались итератором ArrayList из пакета java.util).

Наконец, потребовалось организовать оповещение клиента о вызовах quack(). Мы воспользовались паттерном Наблюдатель, чтобы объекты Quackologist могли регистрироваться в качестве наблюдателей Quackable. В этой реализации тоже был применен паттерн Итератор. Следует отметить, что паттерн Наблюдатель может применяться не только к отдельным объектам, но и к комбинациям.

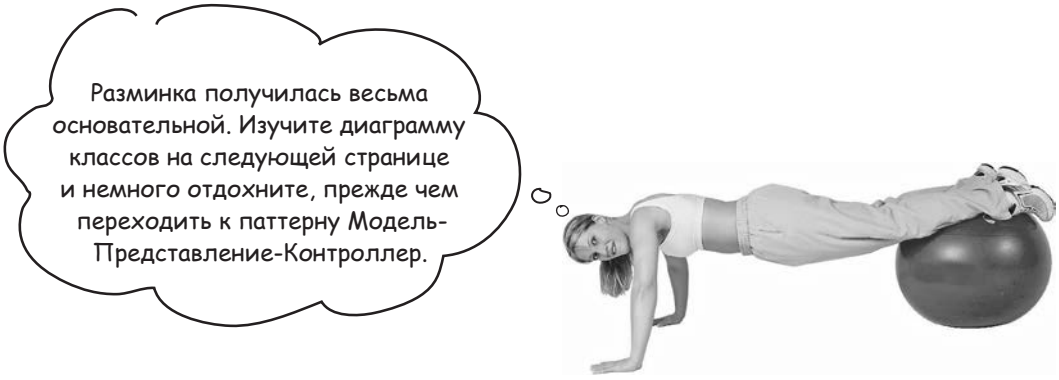
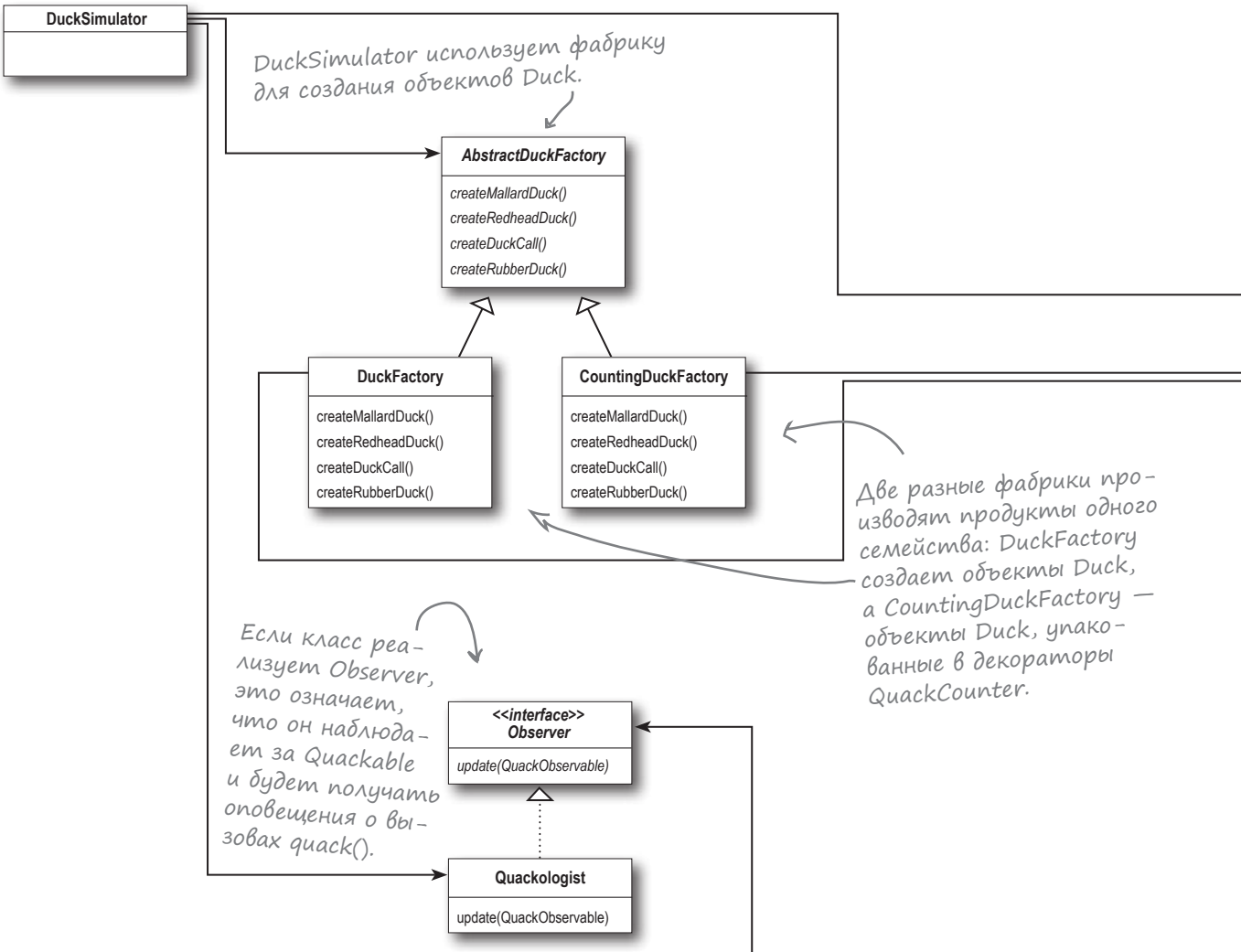
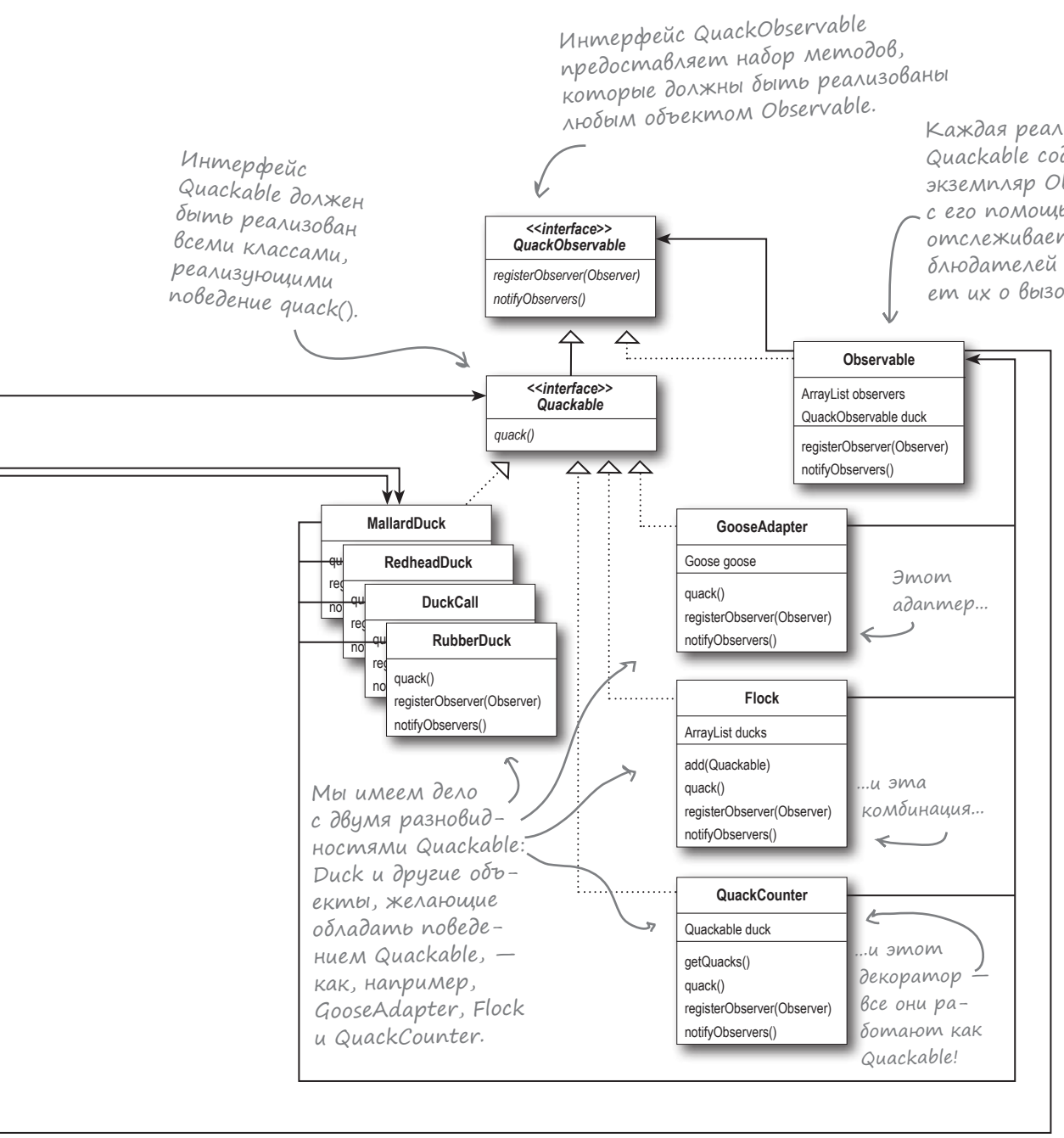


Диаграмма классов с высоты ~~птичьего~~ утиного полета

В одной маленькой программе поместилось много паттернов! Общая картина того, что было сделано:




Интерфейс QuackObservable предоставляет набор методов, которые должны быть реализованы любым объектом Observable.
Каждая реализация Quackable содержит экземпляр Observable; с его помощью она отслеживает своих наблюдателей и оповещает их о вызовах quack().
Интерфейс Quackable должен быть реализован всеми классами, реализующими поведение quack().
<<interface>>
QuackObservable
registerObserver(Observer)
notifyObservers()
<<interface>>
Quackable
quack()
Observable
ArrayList observers
QuackObservable duck
registerObserver(Observer)
notifyObservers()
MallardDuck
RedheadDuck
DuckCall
RubberDuck
quack()
registerObserver(Observer)
notifyObservers()
GooseAdapter
Goose goose
quack()
registerObserver(Observer)
notifyObservers()
Этот адаптер...
Flock
ArrayList ducks
add(Quackable)
quack()
registerObserver(Observer)
notifyObservers()
...и эта комбинация...
QuackCounter
Quackable duck
getQuacks()
quack()
registerObserver(Observer)
notifyObservers()
...и этот декоратор — все они работают как Quackable!
Мы имеем дело с двумя разновидностями Quackable: Duck и другие объекты, желающие обладать поведением Quackable, — как, например, GooseAdapter, Flock и QuackCounter.

Паттерны проектирования — ключ к MVC.

У вас уже был неудачный опыт использования паттерна Модель-Представление-Контроллер (MVC)? Вы не одиноки. Возможно, вы слышали от других программистов, что этот паттерн изменил их жизнь и что он помогает обрести внутреннюю гармонию. Безусловно, это мощный составной паттерн, и хотя внутренней гармонии мы не гарантируем, после освоения он сэкономит вам немало времени.

Но сначала его нужно освоить, верно? Однако на этот раз сделать это будет значительно проще, потому что *вы знаете паттерны!*

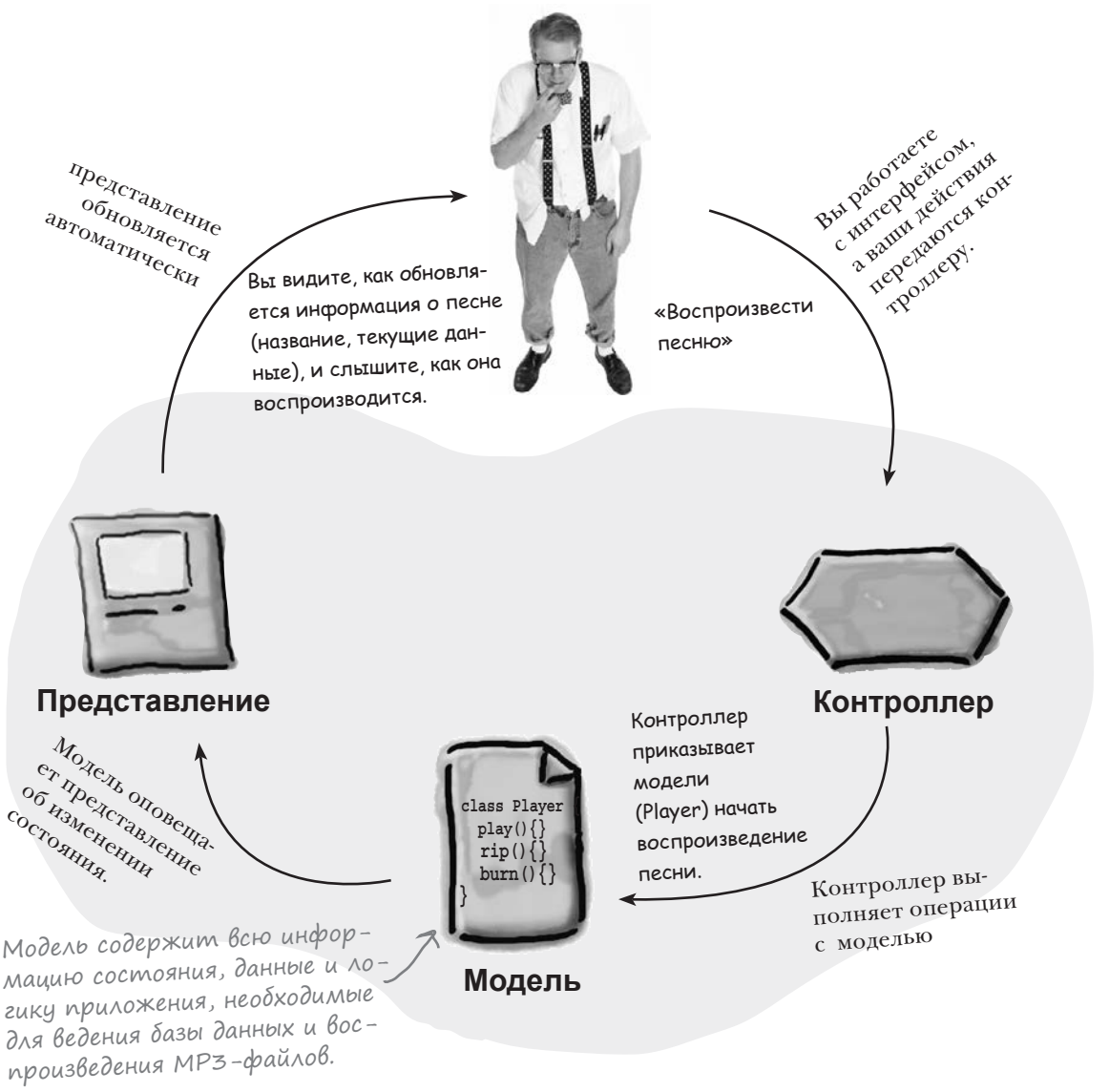
Да, паттерны — ключ к MVC. Осваивать MVC «сверху вниз» сложно; лишь немногие программисты справляются с этой задачей. Вы должны понять главный секрет изучения MVC: это всего лишь *комбинация нескольких паттернов*. Если при изучении паттерна MVC вы обращаете внимание на составляющие его паттерны, все внезапно становится на свои места.

Приступаем к делу. На этот раз MVC от вас не уйдет!

Знакомьтесь! Паттерн Модель-Представление-Контроллер

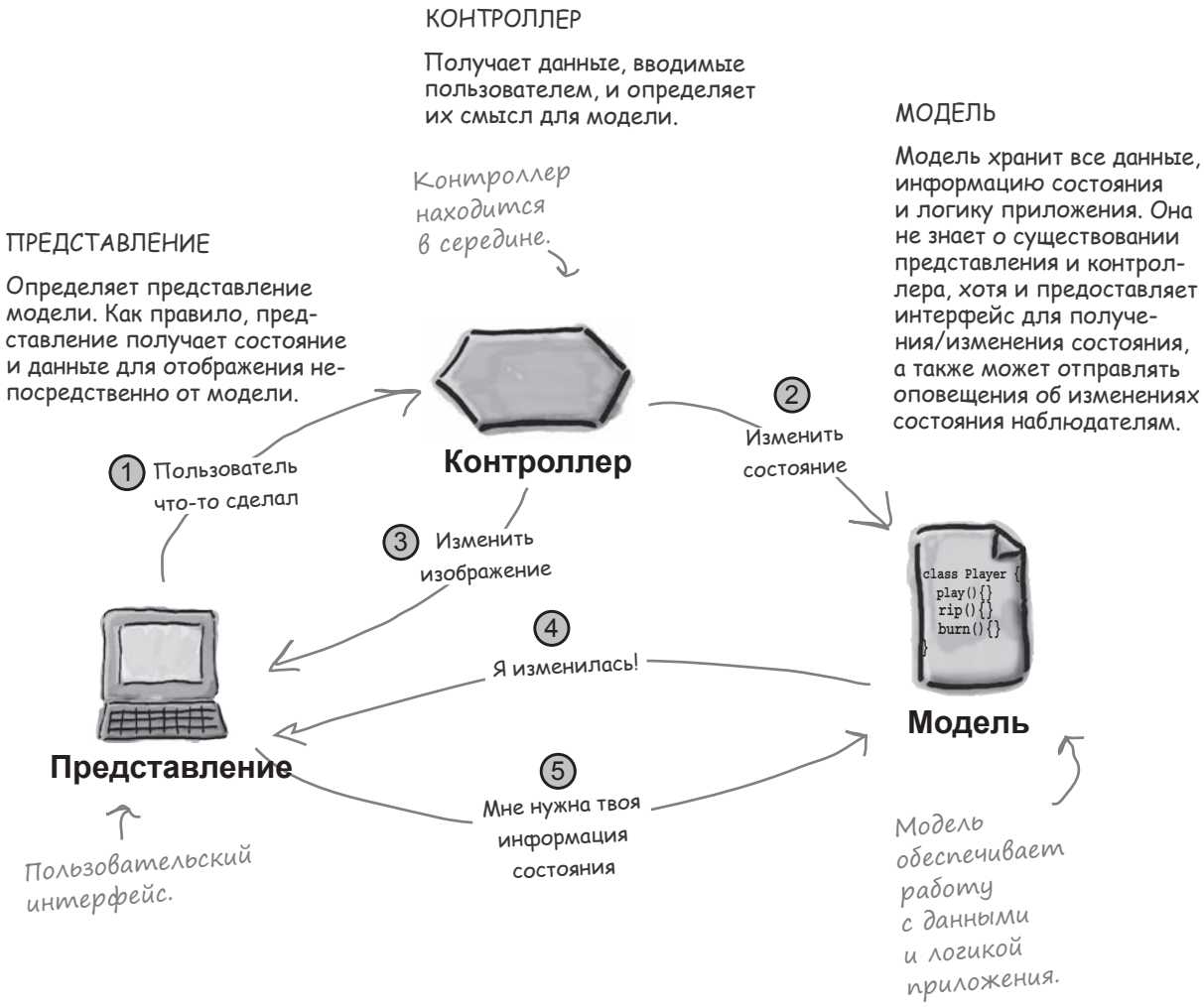
Представьте, что вы используете свой любимый MP3-проигрыватель (скажем, iTunes). Интерфейс программы используется для добавления новых песен, управления списками воспроизведения и переименования. Проигрыватель ведет небольшую базу данных с названиями и информацией о песнях. Кроме того, он воспроизводит их, причем в ходе воспроизведения пользовательский интерфейс постоянно обновляется: в нем выводится название текущей песни, позиция воспроизведения и т. д.

А в основе этой модели заложен паттерн Модель-Представление-Контроллер...



Присмотримся повнимательнее...

Описание MP3-проигрывателя в общих чертах демонстрирует работу MVC, но оно совершенно не объясняет все технические тонкости работы и построения составного паттерна – и вообще зачем он нужен. Начнем с анализа отношений между моделью, представлением и контроллером, а затем рассмотрим эту архитектуру в контексте паттернов проектирования.



1. **Пользователь взаимодействует с моделью.**
   Представление — «окно», через которое пользователь воспринимает модель. Когда вы делаете что-то с представлением (скажем, щелкаете на кнопке воспроизведения), представление сообщает контроллеру, какая операция была выполнена. Контроллер должен обработать это действие.

2. **Контроллер обращается к модели с запросами об изменении состояния.**
   Контроллер получает действия пользователя и интерпретирует их. Если вы щелкаете на кнопке, контроллер должен разобраться, что это значит и какие операции с моделью должны быть выполнены при данном действии.

3. **Контроллер также может обратиться к представлению с запросом об изменении.**
   Когда контроллер получает действие от представления, в результате его обработки он может обратиться к представлению с запросом на изменение (скажем, заблокировать некоторые кнопки или команды меню).

4. **Модель оповещает представление об изменении состояния.**
   Когда в модели что-то изменяется (вследствие действий пользователя или других внутренних изменений — скажем, перехода к следующей песне в списке), модель оповещает представление об изменении состояния.

5. **Представление запрашивает у модели информацию состояния.**
   Представление получает отображаемую информацию состояния непосредственно от модели. Например, когда модель оповещает представление о начале воспроизведения новой песни, представление запрашивает название песни и отображает его. Представление также может запросить у модели информацию состояния в результате запроса на изменение состояния со стороны контроллера.

Часто Задаваемые Вопросы

В: Контроллер бывает наблюдателем для модели?

О: Конечно. В некоторых архитектурах контроллер регистрируется у модели и оповещается о ее изменениях — например, если какие-либо аспекты модели напрямую влияют на пользовательский интерфейс (скажем, в некоторых состояниях модели отдельные элементы интерфейса могут блокироваться).

В: Выходит, задача контроллера сводится к получению пользовательского ввода в представлении и передаче его модели? Так почему бы не разместить соответствующий код прямо в представлении? Ведь обычно работа контроллера сводится к вызову метода модели?

О: Функции контроллера не сводятся к передаче данных модели. Контроллер отвечает за интерпретацию ввода и выполнение соответствующих операций с моделью.

Почему нельзя «сделать все в коде представления»? Можно, но не стоит по двум причинам. Во-первых, это усложнит код представления — у него появится две обязанности: управление пользовательским интерфейсом и логика управления моделью. Во-вторых, между представлением и моделью формируется жесткая привязка. О повторном использовании кода представления с другой моделью можно забыть. Логическая изоляция представления и контроллера способствует формированию более гибкой и расширяемой архитектуры, которая лучше адаптируется к возможным изменениям.

MVC с точки зрения паттернов

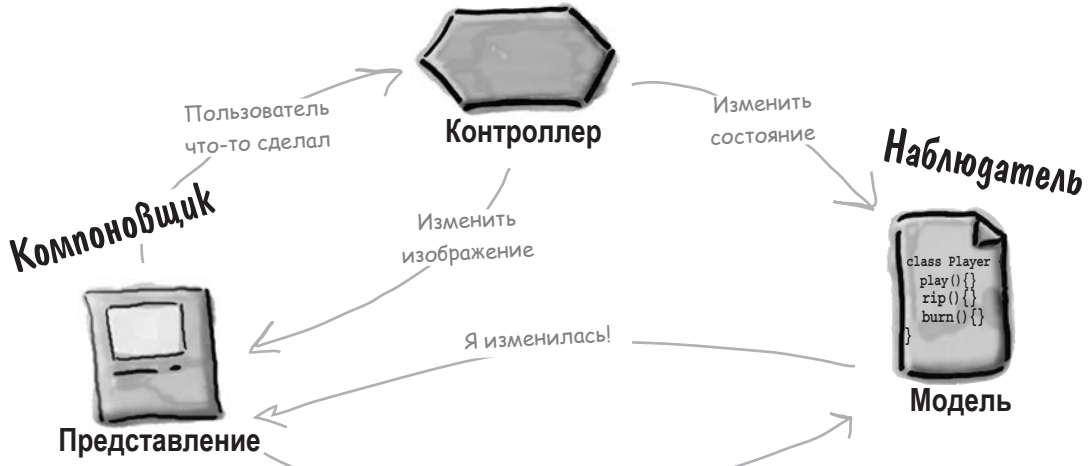
Как мы уже говорили, MVC лучше всего изучать как совокупность паттернов, работающих совместно в одной архитектуре.

Начнем с модели. Как вы, возможно, догадались, модель использует паттерн Наблюдатель для оповещения представлений и контроллеров об изменениях состояния. С другой стороны, представление и контроллер реализуют паттерн Стратегия. Контроллер определяет поведение представления и может быть легко заменен другим контроллером при необходимости смены поведения. Само представление тоже использует внутренний паттерн Компоновщик для управления окнами, кнопками и другими компонентами изображения.

Присмотримся повнимательнее:

Стратегия

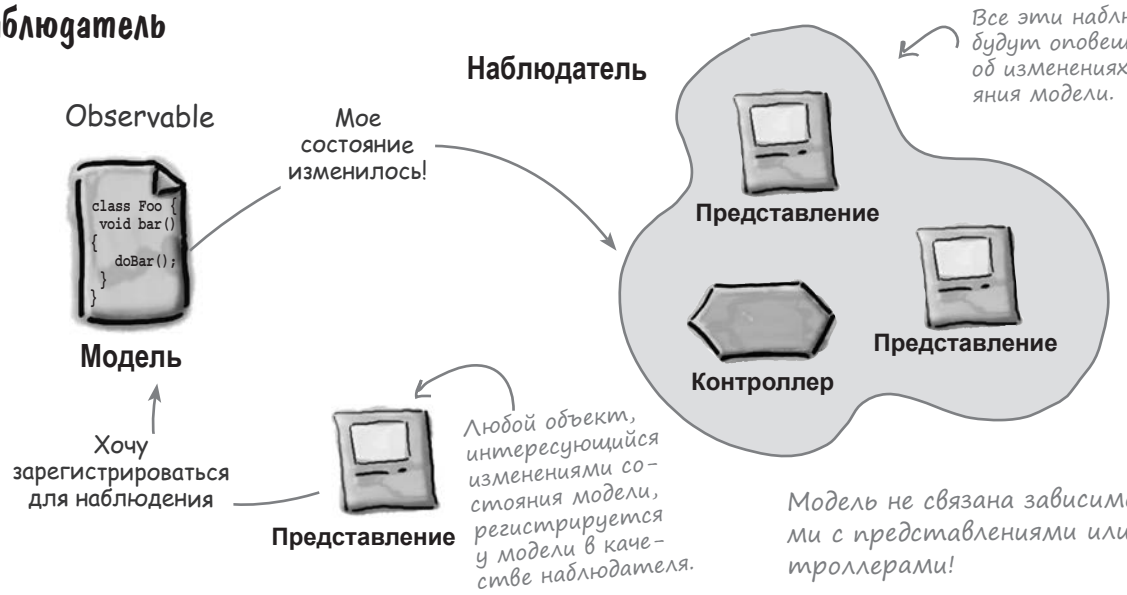
Представление и контроллер реализуют классический паттерн Стратегия. Представление — объект со сменной стратегией, контроллер эту стратегию предоставляет. Представление интересуют только визуальные аспекты приложения, а все решения относительно поведения интерфейса делегируются контроллеру. Применение паттерна Стратегия также сохраняет логическую изоляцию представления от модели, потому что все взаимодействия с моделью для выполнения пользовательских запросов осуществляются контроллером. Представлению о них ничего не известно.



Изображение состоит из вложенных окон, панелей, кнопок, надписей и т. д. Каждый компонент является комбинацией (окно) или листом (кнопка). Когда контроллер приказывает представлению обновиться, он обращается к верхнему компоненту, а паттерн Компоновщик делает все остальное.

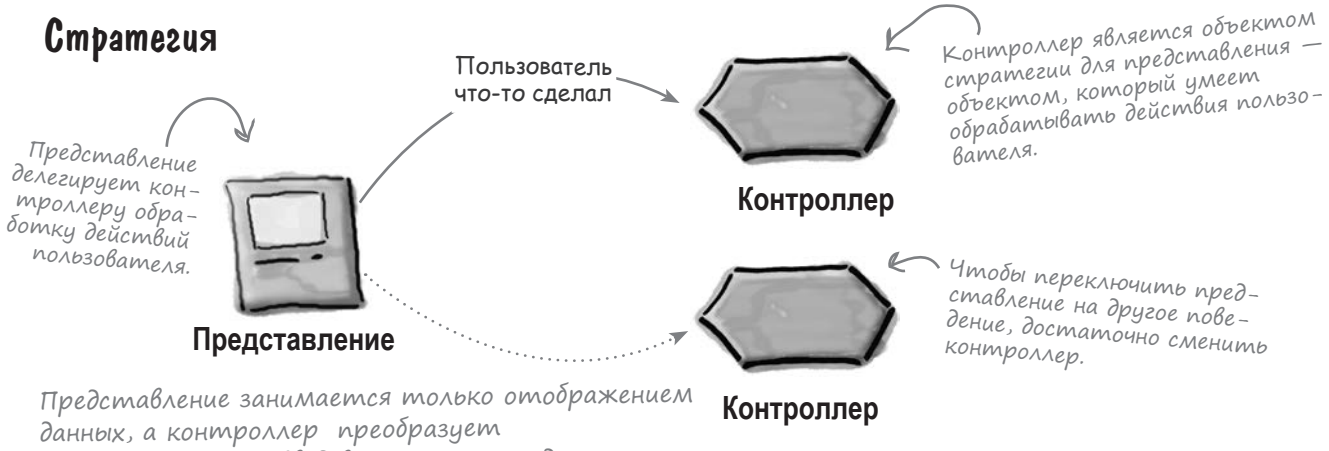
Модель реализует паттерн Наблюдатель для оповещения заинтересованных объектов об изменениях состояния. Паттерн Наблюдатель обеспечивает полную независимость модели от представлений и контроллеров. Он позволяет использовать разные представления с одной моделью, или даже несколько представлений одновременно.

Наблюдатель



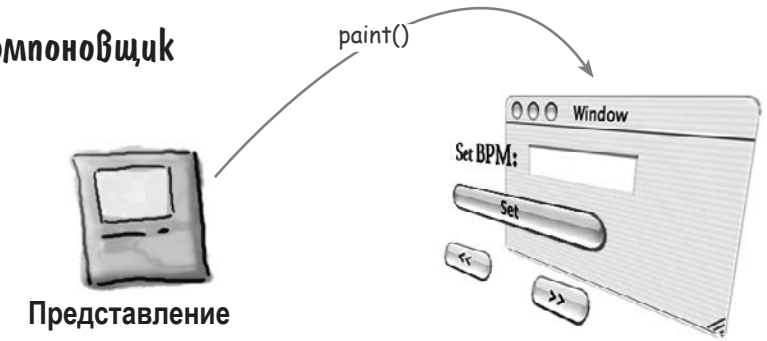
Модель не связана зависимостями с представлениями или контроллерами!

Стратегия



Представление занимается только отображением данных, а контроллер преобразует пользовательский ввод в операции с моделью.

Компоновщик



Представление является комбинацией компонентов GUI (надписи, кнопки, текстовые поля и т. д.). Компонент верхнего уровня содержит другие компоненты, которые содержат другие компоненты и т. д. вплоть до листовых узлов.

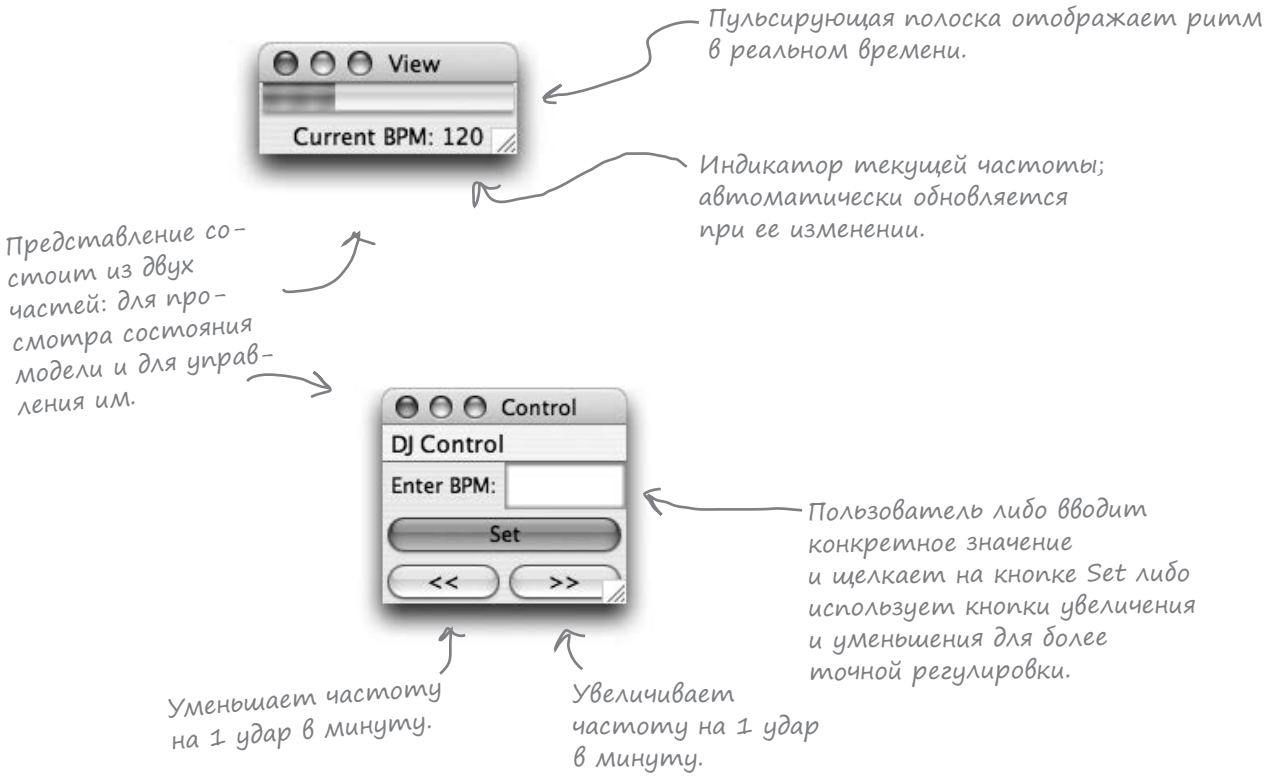
Использование MVC для управления ритмом...

Для диджея главное — это ритм. Вы начинаете свой микс с медленных 95 ударов в минуту (BPM), а потом доводите толпу до бешеного техно-транса на 140 BPM. Серия должна завершаться спокойным эмбиент-миксом на 80 BPM.

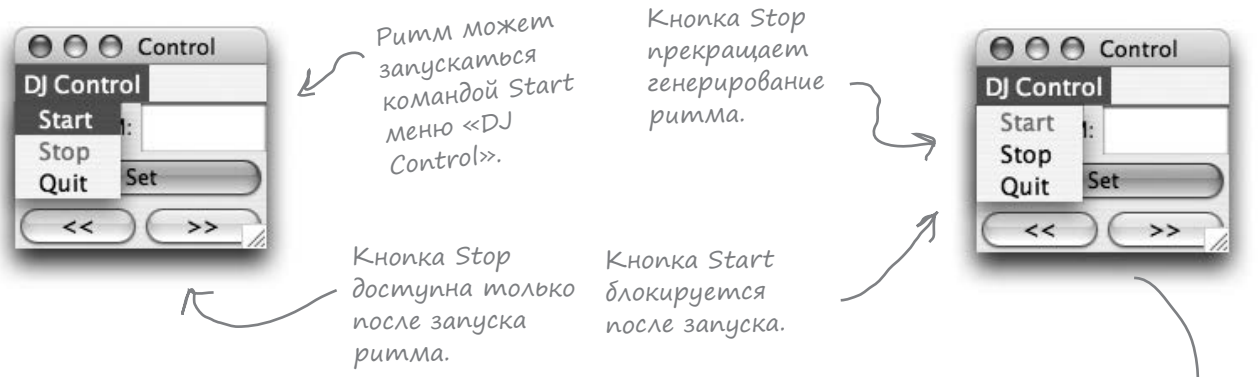
Как это сделать? Разумеется, нам придется создать специальные инструменты для управления ритмом.

Представление для диджея

Начнем с **представления** нашего нового инструмента. Оно позволяет создать заводной ритм и задать количество ударов в минуту...



Еще несколько способов управления представлением для диджеев...

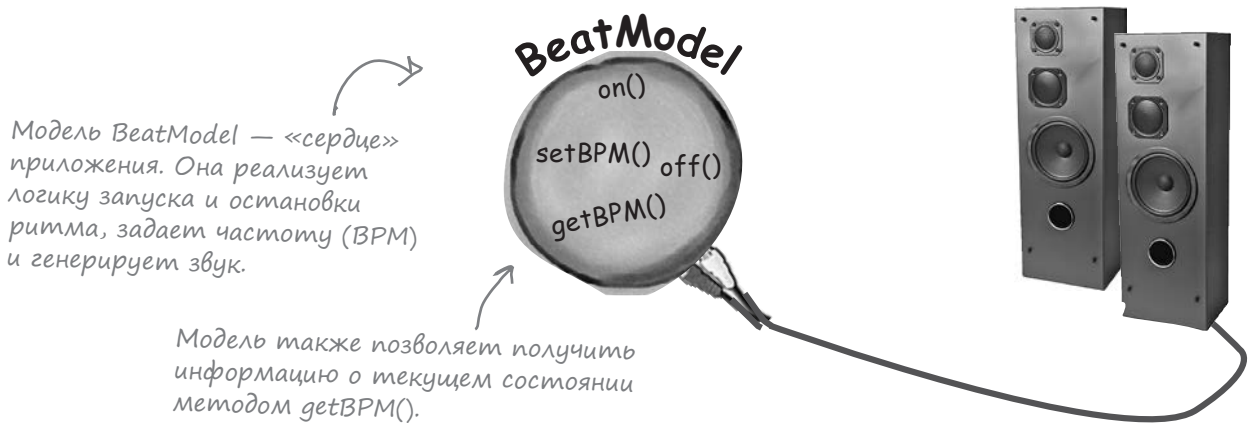


Контроллер находится в середине...

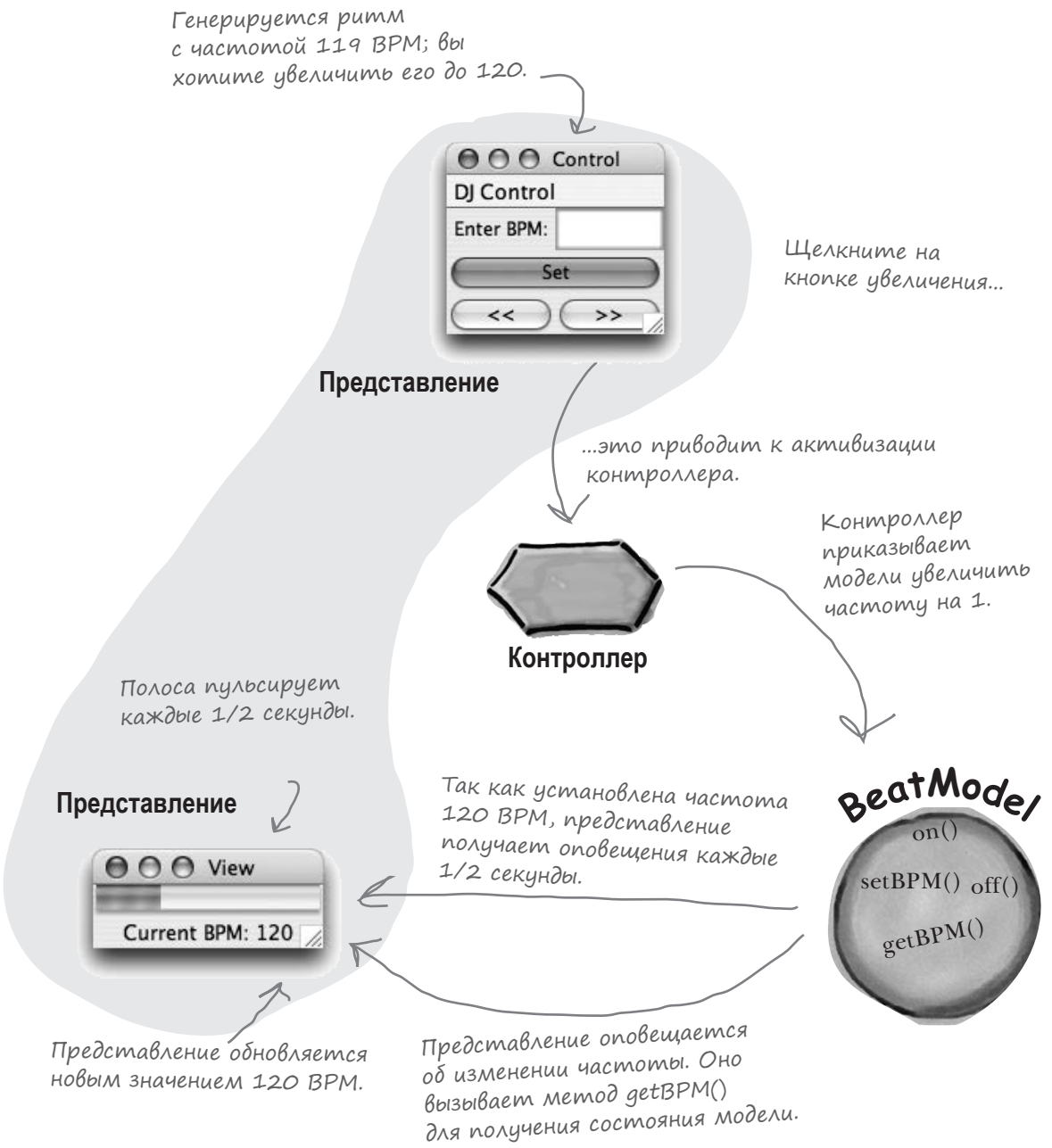
Контроллер находится между представлением и моделью. Он получает ввод от пользователя (скажем, выбор команды Start) и преобразует его в действие с моделью (запуск ритма).

И не забывайте о модели, которая лежит в основе...

Модель не видна, но хорошо слышна. Она лежит в основе всего происходящего, управляя ритмом и динамиками через интерфейс MIDI.



Все вместе


Генерируется ритм с частотой 119 BPM; вы хотите увеличить его до 120.
Control
DJ Control
Enter BPM:
Set
<<
>>
Представление
Щелкните на кнопке увеличения...
...это приводит к активизации контроллера.
Контроллер
Контроллер приказывает модели увеличить частоту на 1.
BeatModel
on()
setBPM()
off()
getBPM()
Так как установлена частота 120 BPM, представление получает оповещения каждые 1/2 секунды.
Представление оповещается об изменении частоты. Оно вызывает метод getBPM() для получения состояния модели.
Представление обновляется новым значением 120 BPM.
Представление
View
Current BPM: 120
Полоса пульсирует каждые 1/2 секунды.

Построение компонентов

Итак, модель отвечает за хранение всех данных, информации состояния и логики приложения. Что будет хранить конкретная модель BeatModel? Ее главная задача — управление ритмом, поэтому в ней должна храниться текущая частота и код, генерирующий события MIDI для создания того ритма, который мы слышим. Кроме того, модель предоставляет интерфейс, через который контроллер управляет ритмом, и позволяет представлению и контроллеру возможность получения состояния модели. А поскольку модель использует паттерн Наблюдатель, также понадобятся методы для регистрации объектов и отправки оповещений.

Прежде чем обращаться к реализации, рассмотрим интерфейс BeatModelInterface:

```
public interface BeatModelInterface {
    void initialize();

    void on();

    void off();

    void setBPM(int bpm);

    int getBPM();

    void registerObserver(BeatObserver o);

    void removeObserver(BeatObserver o);

    void registerObserver(BPMObserver o);

    void removeObserver(BPMObserver o);
}
```

Вызывается после создания экземпляра BeatModel.

Методы, используемые контроллером для управления моделью на основании действий пользователя.

Методы запуска и остановки генератора ритма.

Метод задает частоту ритма (удары в минуту). Частота изменяется сразу же после его вызова.

Метод getBPM() возвращает текущую частоту или 0, если генератор отключен.

Методы, при помощи которых представление и контроллер получают информацию состояния и изменяют свой статус наблюдателя.

Выглядят знакомо: методы регистрации объектов для оповещения об изменениях состояния.

Наблюдатели делятся на две группы: те, которые должны оповещаться о каждом ударе, и те, которые должны оповещаться только об изменениях частоты.

Перед вами конкретный класс BeatModel:

Реализует BeatModelInterface.

Необходимо для MIDI-кода.

sequencer — генератор ритма (того, что вы слышите!).

```
public class BeatModel implements BeatModelInterface, MetaEventListener {
    Sequencer sequencer;
    ArrayList<BeatObserver> beatObservers = new ArrayList<BeatObserver>();
    ArrayList<BPMObserver> bpmObservers = new ArrayList<BPMObserver>();
    int bpm = 90;
    // Другие переменные экземпляров

    public void initialize() {
        setUpMidi();
        buildTrackAndStart();
    }

    public void on() {
        sequencer.start();
        setBPM(90);
    }

    public void off() {
        setBPM(0);
        sequencer.stop();
    }

    public void setBPM(int bpm) {
        this.bpm = bpm;
        sequencer.setTempoInBPM(getBPM());
        notifyBPMObservers();
    }

    public int getBPM() {
        return bpm;
    }

    void beatEvent() {
        notifyBeatObservers();
    }

    // Код регистрации и оповещения наблюдателей

    // MIDI-код
}
```

В контейнерах ArrayList хранятся две категории наблюдателей.

В переменной bpm хранится частота ритма — по умолчанию 90 BPM.

Метод настраивает генератор и готовит музыку для воспроизведения.

Метод on() запускает генератор и устанавливает частоту по умолчанию (90 BPM).

Метод off() останавливает генератор и задает частоту равной 0.

Метод setBPM() используется контроллером для управления ритмом. Он выполняет три операции:

(1) Присваивание значения переменной bpm

(2) Запрос к генератору на изменение частоты.

(3) Оповещение всех BPM-наблюдателей об изменении частоты.

Метод getBPM() просто возвращает значение переменной bpm, определяющее текущую частоту ритма.

Метод beatEvent(), не входящий в BeatModelInterface, вызывается MIDI-кодом при каждом новом ударе. Метод оповещает всех наблюдателей BeatObserver.

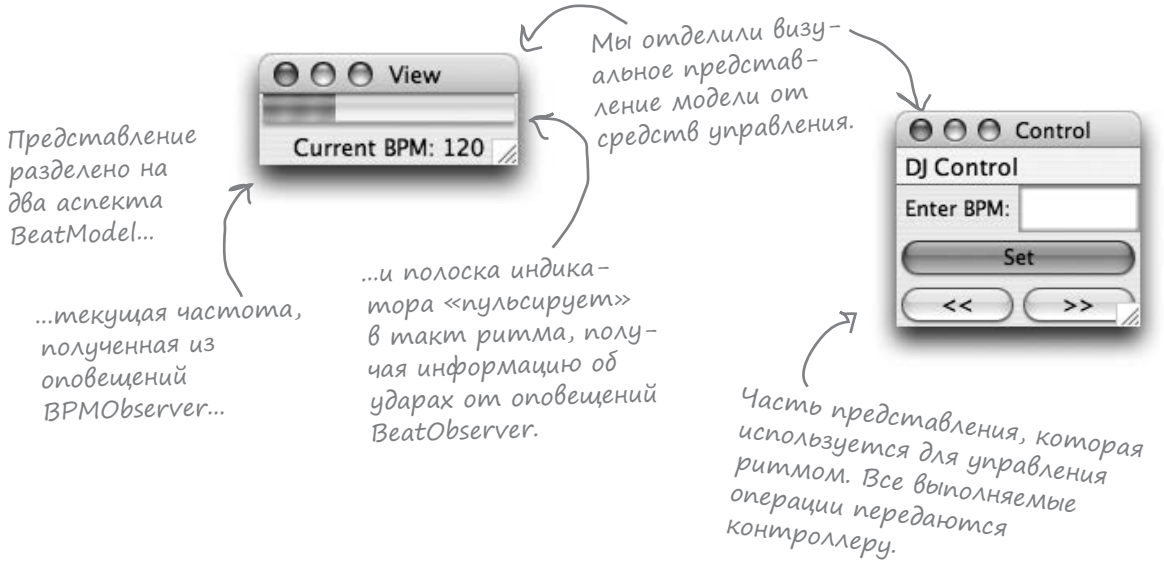
Готово к употреблению

Модель использует MIDI-поддержку Java для генерирования ритма. С полной реализацией всех классов можно ознакомиться на сайте headfirstlabs.com; полный листинг приведен в конце главы.

Представление

А теперь начинается самое интересное: мы подключим представление и обеспечим визуальное отображение BeatModel!

Первая важная особенность нашего представления заключается в том, что оно отображается в двух разных окнах. Одно окно содержит текущую частоту и индикатор ударов, а в другом отображаются элементы управления. Почему мы выбрали такое решение? Потому что хотели подчеркнуть разницу между интерфейсом, обеспечивающим отображение модели, и средствами управления. Давайте получше присмотримся к двум составляющим представления:



МОЗГОВОЙ ШТУРМ

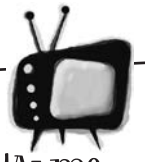
Наша модель BeatModel никак не зависит от представления. Она реализуется на базе паттерна Наблюдатель: любое представление, зарегистрированное в качестве наблюдателя, просто оповещается об изменении состояний. Представление работает с состоянием через API. Мы реализовали одно из возможных представлений; можете ли вы предложить другие представления, использующие оповещения и состояние BeatModel?

Световое шоу, управляемое ритмом в реальном времени.

Бегущая строка с названием музыкального жанра в зависимости от частоты (эмбиент, техно и т. д.)

Реализация представления

Две части нашего представления — визуальная информация модели и элементы управления — отображаются в разных окнах, но находятся в одном классе Java. Сначала мы рассмотрим код создания визуального представления, в котором выводится текущая частота и индикатор ударов. На следующей странице будет представлен код создания элементов управления: текстового поля для ввода частоты и кнопок.



Будьте осторожны!

Код на этих двух страницах разделен условно!

Мы разбили ОДИН класс на ДВА; одна часть приведена на этой странице, а другая — на следующей. В действительности весь код находится в ОДНОМ классе DJView.java. Полный листинг приведен в конце главы.

DJView — наблюдатель для событий обоих видов (удары в реальном времени и изменения частоты).

```
public class DJView implements ActionListener,  BeatObserver, BPMObserver {
    BeatModelInterface model;
    ControllerInterface controller;
    JFrame viewFrame;
    JPanel viewPanel;
    BeatBar beatBar;
    JLabel bpmOutputLabel;

    public DJView(ControllerInterface controller, BeatModelInterface model) {
        this.controller = controller;
        this.model = model;
        model.registerObserver((BeatObserver)this);
        model.registerObserver((BPMObserver)this);
    }

    public void createView() {
        // Create all Swing components here
    }

    public void updateBPM() {
        int bpm = model.getBPM();
        if (bpm == 0) {
            bpmOutputLabel.setText("offline");
        } else {
            bpmOutputLabel.setText("Current BPM: " + model.getBPM());
        }
    }

    public void updateBeat() {
        beatBar.setValue(100);
    }
}
```

В представлении хранятся ссылки на модель и на контроллер. Контроллер используется только управляющими элементами, до которых мы скоро доберемся...

Создание отображаемых компонентов.

Конструктор получает ссылки на контроллер и модель и сохраняет их в переменных.

Представление регистрируется в качестве наблюдателя BeatObserver и BPMObserver модели.

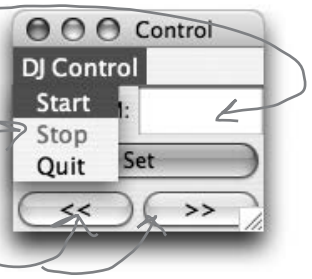
Метод updateBPM() вызывается при изменении состояния модели. Изображение обновляется текущим значением частоты, которое запрашивается непосредственно у модели.

Метод updateBeat() вызывается в начале нового удара. Когда это происходит, необходимо отобразить импульс на «индикаторе ритма». Для этого индикатору присваивается максимально возможное значение (100).

Реализация представления (продолжение)

Переходим к коду управляющего интерфейса представления. Это представление позволяет управлять моделью: вы сообщаете контроллеру, что необходимо сделать, а он, в свою очередь, указывает модели, что ей делать. Еще раз напомним, что этот код находится в том же файле класса, что и код другой части представления.

```
public class DJView implements ActionListener,  BeatObserver, BPMObserver {
    BeatModelInterface model;
    ControllerInterface controller;
    JLabel bpmLabel;
    JTextField bpmTextField;
    JButton setBPMButton;
    JButton increaseBPMButton;
    JButton decreaseBPMButton;
    JMenuBar menuBar;
    JMenu menu;
    JMenuItem startMenuItem;
    JMenuItem stopMenuItem;

    public void createControls() {
        // Create all Swing components here
    }
    public void enableStopMenuItem() {
        stopMenuItem.setEnabled(true);
    }

    public void disableStopMenuItem() {
        stopMenuItem.setEnabled(false);
    }

    public void enableStartMenuItem() {
        startMenuItem.setEnabled(true);
    }

    public void disableStartMenuItem() {
        startMenuItem.setEnabled(false);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == setBPMButton) {
            int bpm = Integer.parseInt(bpmTextField.getText());
            controller.setBPM(bpm);
        } else if (event.getSource() == increaseBPMButton) {
            controller.increaseBPM();
        } else if (event.getSource() == decreaseBPMButton) {
            controller.decreaseBPM();
        }
    }
}
```



Метод создает все элементы управления и размещает их в интерфейсе приложения, а также создает меню. При выборе команды Start или Stop вызываются соответствующие методы контроллера.

Методы установления и снятия блокировки команд меню Start и Stop. Как будет вскоре показано, контроллер использует эти методы для управления интерфейсом.

Метод вызывается при щелчке на кнопке.

Если выбрана кнопка Set, действие передается контроллеру вместе с новой частотой.

Если выбрана кнопка увеличения или уменьшения частоты, эта информация также передается контроллеру.

А теперь — контроллер

Осталось написать последний отсутствующий компонент: контроллер. Напомним, что контроллер представляет собой объект стратегии, который подключается к представлению для управления действиями последнего.

Как обычно, реализация паттерна Стратегия начинается с определения интерфейса, реализации которого могут подключаться к представлению. Мы назовем его ControllerInterface:

Методы контроллера, которые могут вызываться представлением.

```
public interface ControllerInterface {
    void start();
    void stop();
    void increaseBPM();
    void decreaseBPM();
    void setBPM(int bpm);
}
```

После изучения визуального интерфейса модели эти методы выглядят знакомо. Они выполняют те же операции: запуск и остановка ритма, изменение частоты. Этот интерфейс «шире» интерфейса BeatModel, потому что в нем присутствует возможность увеличения и уменьшения частоты.

Головоломка

Представление и контроллер совместно используют паттерн Стратегия. Можете ли вы нарисовать для них диаграмму классов, представляющую этот паттерн?

Реализация контроллера:

```
public class BeatController implements ControllerInterface {
    BeatModelInterface model;
    DJView view;

    public BeatController(BeatModelInterface model) {
        this.model = model;
        view = new DJView(this, model);
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
        model.initialize();
    }

    public void start() {
        model.on();
        view.disableStartMenuItem();
        view.enableStopMenuItem();
    }

    public void stop() {
        model.off();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
    }

    public void increaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm + 1);
    }

    public void decreaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm – 1);
    }

    public void setBPM(int bpm) {
        model.setBPM(bpm);
    }
}
```

Контроллер реализует ControllerInterface.

Контроллер получает объекты модели и представления и связывает их воедино.

Контроллер получает модель в конструкторе и создает представление.

При выборе команды Start контроллер активизирует модель и изменяет пользовательский интерфейс: команда Start блокируется, а команда Stop становится доступной.

И наоборот, при выборе команды контроллер деактивизирует модель, команда Start становится доступной, а команда Stop блокируется.

При щелчке на кнопке увеличения контроллер получает текущую частоту от модели, увеличивает ее на 1 и задает результат как новое значение частоты.

То же самое, только частота уменьшается на 1.

Наконец, если пользователь ввел произвольную частоту, контроллер приказывает модели перейти на новое значение.

ВНИМАНИЕ: все разумные решения за представление принимает контроллер. Представление умеет только устанавливать и снимать блокировку команд меню; оно не знает, в каких ситуациях это следует делать.

Объединяем все компоненты...

У нас есть все необходимое: модель, представление и контроллер. Пора объединить их в архитектуру MVC и посмотреть (а также услышать), как эти компоненты работают в сочетании друг с другом.

Также необходимо написать небольшой фрагмент тестового кода; это не займет много времени:

```
public class DJTestDrive {
    public static void main (String[] args) {
        BeatModelInterface model = new BeatModel();
        ControllerInterface controller = new BeatController(model);
    }
}
```

Сначала создается модель...

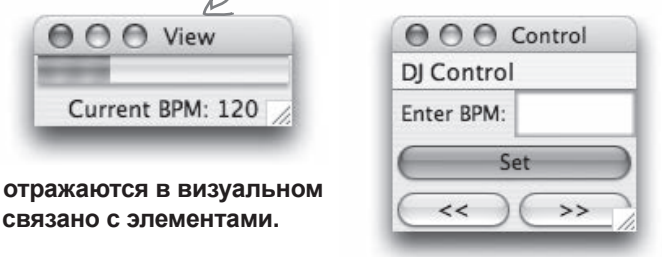
...затем создаем контроллер и передаем ему модель. Напомним, что представление создается контроллером, поэтому нам это делать не нужно.

А теперь тестовый запуск...

```
File Edit Window Help LetTheBassKick
% java DJTestDrive
%
```

Введите эту команду...

...и вы получите такой результат:



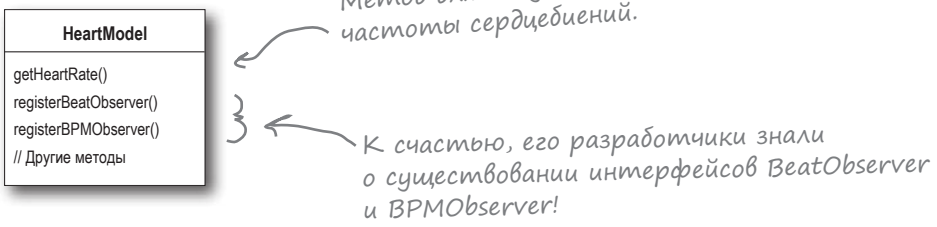
Что нужно сделать

1. **Запустите генератор ритма командой меню Start. Обратите внимание на то, что контроллер блокирует эту команду.**
2. **Измените частоту при помощи элементов управления. Обратите внимание: изменения отражаются в визуальном представлении, хотя логически оно никак не связано с элементами.**
3. **Индикатор постоянно синхронизируется с ритмом, потому что он зарегистрирован в качестве наблюдателя модели.**
4. **Включите свою любимую песню и попробуйте подогнать ритм при помощи кнопок увеличения и уменьшения.**
5. **Остановите генератор. Обратите внимание: контроллер блокирует команду Stop и снимает блокировку с команды Start.**

Анализ паттерна Стратегия

Давайте чуть подробнее разберемся в том, как паттерн Стратегия используется в архитектуре MVC. На этом пути нам встретится другой полезный паттерн, часто участвующий в трио MVC: паттерн Адаптер.

Итак, наше представление выводит частоту ритма и содержит индикатор текущего удара. Вам это ничего не напоминает? Например, ритм биения сердца? У нас совершенно случайно под рукой оказался класс для получения данных о сердечной деятельности; вот как выглядит его диаграмма классов:



МОЗГОВОЙ ШТУРМ

Конечно, было бы удобно использовать текущее представление с HeartModel, но нам понадобится контроллер, работающий с этой моделью. Кроме того, интерфейс HeartModel не соответствует этому представлению — он содержит метод getHeartRate() вместо метода getBPM(). Как спроектировать классы, которые позволяли бы использовать представление с новой моделью?

Адаптация модели

Прежде всего заметим, что HeartModel необходимо адаптировать к BeatModel. Если этого не сделать, то представление не сможет работать с моделью, потому что модель работает только с getBPM(), а эквивалентный метод HeartModel называется getHeartRate(). Как это сделать? Конечно, мы воспользуемся паттерном Адаптер! Адаптация модели к существующим контроллерам и представлениям является стандартным приемом при работе с MVC.

Код адаптера, преобразующего HeartModel к BeatModel:

```
public class HeartAdapter implements BeatModelInterface {
    HeartModelInterface heart;

    public HeartAdapter(HeartModelInterface heart) {
        this.heart = heart;
    }
    public void initialize() {}

    public void on() {}

    public void off() {}

    public int getBPM() {
        return heart.getHeartRate();
    }

    public void setBPM(int bpm) {}

    public void registerObserver(BeatObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BeatObserver o) {
        heart.removeObserver(o);
    }

    public void registerObserver(BPMObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BPMObserver o) {
        heart.removeObserver(o);
    }
}
```

Необходимо реализовать целевой интерфейс BeatModelInterface.

Сохранение ссылки на HeartModel.

Оставляем эти методы пустыми.

Вызов getBPM() преобразуется в вызов getHeartRate() модели HeartModel.

Еще одна пустая операция.

Вызовы методов наблюдателя делегируются встроенному объекту HeartModel.

Можно переходить к HeartController

Располагая адаптером HeartAdapter, мы можем создать контроллер и организовать работу представления с HeartModel. Повторное использование кода в действии!

HeartController реализует ControllerInterface (по аналогии с классом BeatController).

```
public class HeartController implements ControllerInterface {
    HeartModelInterface model;
    DJView view;

    public HeartController(HeartModelInterface model) {
        this.model = model;
        view = new DJView(this, new HeartAdapter(model));
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.disableStartMenuItem();
    }

    public void start() {}

    public void stop() {}

    public void increaseBPM() {}

    public void decreaseBPM() {}

    public void setBPM(int bpm) {}
}
```

Как и прежде, контроллер создает представление и связывает все компоненты.

Изменение: передается HeartModel, а не BeatModel...

...и до передачи представлению модель должна быть упакована в адаптер.

HeartController блокирует ненужные команды меню.

Пустые операции; в отличие от генератора ритма, сердцем невозможно управлять напрямую.

Вот и все! Переходим к написанию тестового кода...

```
public class HeartTestDrive {
    public static void main (String[] args) {
        HeartModel heartModel = new HeartModel();
        ControllerInterface model = new HeartController(heartModel);
    }
}
```

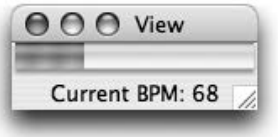
Создаем контроллер и передаем ему HeartModel.

Тестовый запуск

```
File  Edit  Window  Help  CheckMyPulse
% java HeartTestDrive
%
```

Введите эту команду...

...и вот что вы увидите:



Нормальный здоровый пульс.

Что нужно сделать

1. **Представление отлично работает с новой моделью! Так как HeartModel поддерживает BPM- и BeatObserver, на индикаторе ударов успешно отображается пульс.**

2. **В частоте биения сердца имеются естественные отклонения, поэтому датчик периодически обновляется новым значением.**

3. **При каждом обновлении частоты адаптер преобразует вызовы getBPM() в вызовы getHeartRate().**

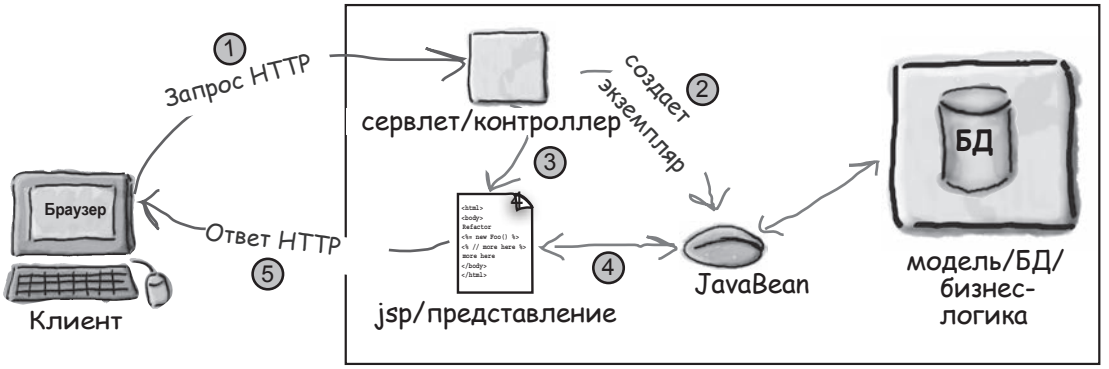
4. **Команды меню Start и Stop недоступны, потому что они блокируются контроллером.**

5. **Другие кнопки работают, но щелчки на них ни к чему не приводят, потому что контроллер не реализует операции этих кнопок. Представление можно было бы изменить, чтобы этот факт был отражен визуально.**

MVC и Веб

Вскоре после широкого распространения Интернета разработчики начали адаптировать паттерн Модель-Представление-Контроллер к модели «браузер/сервер». Основная адаптация «Модель 2» использует комбинацию сервлетов и технологии JSP для достижения той же степени логической изоляции модели, представления и контроллера, как в традиционных графических интерфейсах.

«Модель 2» работает так:



1 **Запрос HTTP принимается сервлетом.**
Пользователь в браузере выдает запрос HTTP. Обычно при этом передаются данные формы (например, имя и пароль). Сервлет принимает данные формы и разбирает их.

2 **Сервер выступает в роли контроллера.**
Сервер играет роль контроллера и обрабатывает ваш запрос — вероятно, с выдачей запросов к модели (обычно к базе данных). Результат обработки запроса обычно упаковывается в форме JavaBean.

3 **Контроллер передает запрос представлению.**
Представлению соответствует код JSP, единственная задача которого — сгенерировать страницу с представлением модели (**4**) и всеми элементами, необходимыми для дальнейших действий.

5 **Представление возвращает страницу браузеру через HTTP.**
Страница возвращается браузеру для отображения. Пользователь создает дальнейшие запросы, которые обрабатываются по той же схеме.

Модель 2 — нечто большее, чем элегантная архитектура.

Вероятно, преимущества разделения представления, модели и контроллера уже достаточно очевидны. Но мы должны поведать и «продолжение истории» с моделью 2 — она спасла от хаоса многие интернет-магазины.

Каким образом? Модель 2 не только разделяет компоненты в архитектурном контексте, но и обеспечивает разделение *технологических обязанностей*. В прежние времена каждый, кто имел доступ к вашему коду JSP, мог написать любой Java-код, верно? И среди них было немало людей, не способных отличить jar-файл от банки арахисового масла. На практике веб-дизайнеры хорошо разбираются в *контенте и HTML, но не в программировании*.

К счастью, Модель 2 помогла решить эту проблему. В Модели 2 программирование является уделом специалистов, хорошо знающих свои сервлеты, а веб-дизайнеры работают с простыми JSP-страницами и имеют доступ только к HTML и простым компонентам JavaBean.

Модель 2: управление ритмом с мобильного телефона

Сами понимаете, мы не могли расстаться с этой темой без переноса модели BeatModel в Интернет. Только подумайте: всем оборудованием диджея можно управлять с веб-страницы на мобильном телефоне! Теперь ничто не помешает вам покинуть пульт диджея и смешаться с толпой. Так чего же вы ждете?

1. **Модификация модели.**
   Вообще-то ничего изменять не придется, модель прекрасно сработает в исходном виде!

2. **Создание сервлета-контроллера.**
   Нам нужен простой сервлет, способный принимать запросы HTTP и выполнять некоторые операции с моделью: запуск, остановка и изменение частоты.

3. **Создание представления HTML.**
   Мы создадим в JSP простое представление. Оно будет получать от контроллера компонент JavaBean со всей информацией для вывода.

Для любознательных

Настройка рабочей среды сервлетов

Настройка рабочей среды сервлетов немного выходит за рамки книги, посвященной паттернам проектирования, — если, конечно, вы не хотите, чтобы эта книга весила больше вас!

Запустите браузер и откройте страницу контейнера сервлетов Tomcat проекта Apache Jakarta по адресу *http://jakarta.apache.org/tomcat/*. Здесь вы найдете все необходимое.

Шаг 1: модель

Помните, что в архитектуре MVC модели не нужно ничего знать о представлениях и контроллерах. Модели известно лишь то, что у нее могут быть наблюдатели, которых необходимо оповещать. В этом проявляется красота паттерна Наблюдатель. Кроме того, модель предоставляет интерфейс, который может использоваться представлениями и контроллерами для получения и изменения состояния.

Нам остается лишь адаптировать ее для веб-среды, но, так как модель не зависит ни от каких внешних классов, никакие реальные изменения не понадобятся. Мы можем использовать класс BeatModel в исходном виде. Без лишних задержек переходим к шагу 2!

Шаг 2: сервлет-контроллер

Напоминаем, что серверу в нашей архитектуре отведена роль контроллера; он получает входные данные от браузера в виде запросов HTTP и преобразует их в действия, применимые к модели.

Вероятно, исходя из основных принципов Веб, представление должно быть возвращено браузеру. Для этого управление передается представлению, которое принимает форму JSP. Мы разберемся с ним на шаге 3.

Ниже представлена основная схема сервлета, а полная реализация приведена на следующей странице.

Расширяем класс HttpServlet для получения доступа к основной функциональности сервлета (в частности, получению запросов HTTP).

We need the serialization id because HttpServlet implements Serializable.

Метод init вызывается при создании сервлета.

Сначала создаем объект BeatModel...

...и сохраняем ссылку на него в контексте сервлета для удобства дальнейших обращений.

```
public class DJViewServlet extends HttpServlet {
    private static final long serialVersionUID = 2L;

    public void init() throws ServletException {
        BeatModel beatModel = new BeatModel();
        beatModel.initialize();
        getServletContext().setAttribute("beatModel", beatModel);
    }

    // doGet method here

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
            throws IOException, ServletException
    {
         // Реализация
    }
}
```

Вся основная работа выполняется в методе doGet(). Его реализация приведена на следующей странице.

Реализация метода doGet() с предыдущей страницы:

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
        throws IOException, ServletException
{
    BeatModel beatModel =
        (BeatModel)getServletContext().getAttribute("beatModel");

    String bpm = request.getParameter("bpm");
    if (bpm == null) {
        bpm = beatModel.getBPM() + "";
    }

    String set = request.getParameter("set");
    if (set != null) {
        int bpmNumber = 90;
        bpmNumber = Integer.parseInt(bpm);
        beatModel.setBPM(bpmNumber);
    }

    String decrease = request.getParameter("decrease");
    if (decrease != null) {
        beatModel.setBPM(beatModel.getBPM() - 1);
    }
    String increase = request.getParameter("increase");
    if (increase != null) {
        beatModel.setBPM(beatModel.getBPM() + 1);
    }
    String on = request.getParameter("on");
    if (on != null) {
        beatModel.start();
    }
    String off = request.getParameter("off");
    if (off != null) {
        beatModel.stop();
    }

    request.setAttribute("beatModel", beatModel);

    RequestDispatcher dispatcher =
        request.getRequestDispatcher("/jsp/DJView.jsp");
    dispatcher.forward(request, response);
}
```

Получаем ссылку на модель из контекста сервлета.

Затем получаем все команды/параметры HTTP...

Если получена команда записи, получаем присваиваемое значение и сообщаем модели.

Получаем текущую частоту от модели и увеличиваем/уменьшаем на 1.

Если получена команда запуска или остановки, выполняем соответствующее действие с моделью.

Работа контроллера закончена. Передаем управления представлению, чтобы оно сгенерировало код HTML.

В соответствии с определением Модели 2 мы передаем JSP JavaBean с состоянием модели. В нашем случае передается сама модель, поскольку она реализована в виде JavaBean.

Теперь нужно представление...

Остается создать представление — и наш генератор ритмов, управляемый из браузера, готов к работе! В модели 2 представлением является обычный код JSP, а вся получаемая им информация — это компонент JavaBean, получаемый от контроллера. В нашем случае это модель, а код JSP использует его свойство BPM для получения текущей частоты. По этим данным создается визуальное представление и элементы пользовательского интерфейса.

Компонент JavaBean, передаваемый сервлетом.

```
<jsp:useBean id="beatModel" scope="request"
             class="headfirst.designpatterns.combined.djview.BeatModel" />
<!doctype html>
<html>
<head>
  <meta charset=»utf-8»>
  <title>DJ View</title>
  <style>...</style>
</head>
<body>

<h1>DJ View</h1>
Beats per minutes = <jsp:getProperty name="beatModel" property="BPM" />
<br><hr><br>

<form method=»post» action=»/djview/servlet/DJViewServlet»>
BPM: <input type=text name="bpm"
            value=»<jsp:getProperty name='beatModel' property='BPM' />»>

<input type="submit" name="set" value="set"><br />
<input type="submit" name="decrease" value="<<">
<input type="submit" name="increase" value=">>"><br />
<input type="submit" name="on" value="on">
<input type="submit" name="off" value="off"><br />
</form>

</body>
</html>
```

Начало кода HTML.

Из JavaBean читается свойство BPM.

Генерируем представление с выводом текущей частоты.

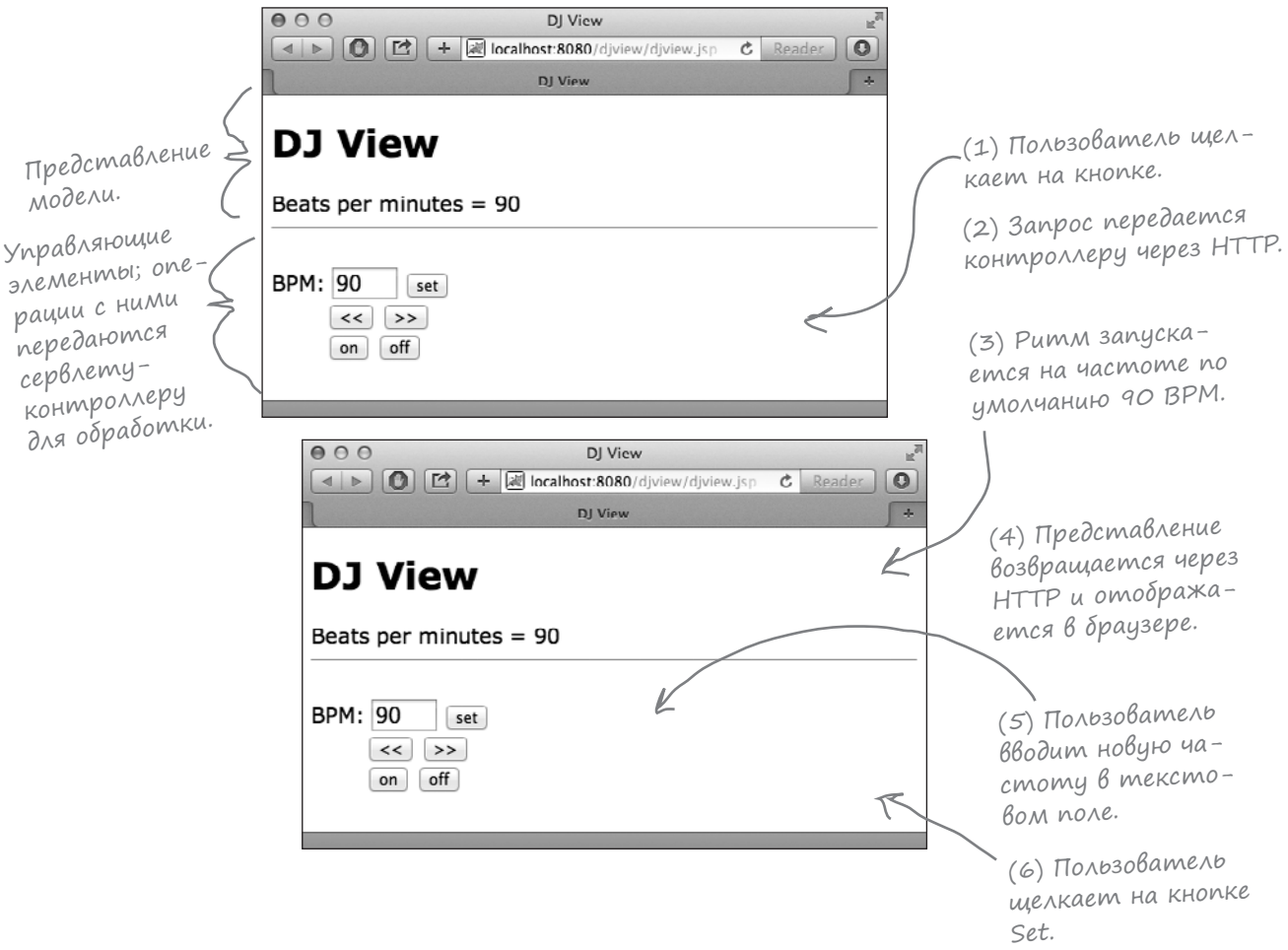
Управляющие элементы в представлении: текстовое поле для ввода частоты с кнопками увеличения/уменьшения и запуска/остановки.

Конец кода HTML.

ВНИМАНИЕ: в Модели 2, как и в MVC, представление не изменяет модель (это работа контроллера): оно только использует ее состояние!

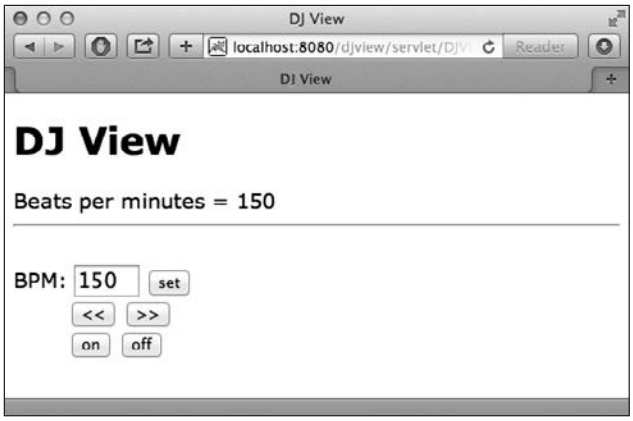
Тестирование Модели 2...

Запускаем браузер, загружаем сервлет DJView и проверяем работу системы...



(8) Контроллер изменяет модель, устанавливая частоту 150 BPM.

(9) Представление возвращает код HTML, соответствующий текущей модели.



Что нужно сделать

1. **Загрузите веб-страницу; вы увидите, что в поле BPM содержится значение 0. Щелкните на кнопке «on».**

2. **В поле частоты появляется значение по умолчанию 90 BPM. На компьютере, на котором работает сервер, генерируется ритм.**

3. **Введите новую частоту (скажем, 120) и щелкните на кнопке «set». Страница обновляется новым значением, а частота ритма нарастает.**

4. **Поэкспериментируйте с кнопками увеличения/уменьшения.**

5. **Представьте, как работает каждый аспект системы. Интерфейс HTML обращается с запросом к серверу. Интерфейс HTML создает запрос к сервлету (контроллеру); сервлет разбирает пользовательский ввод и выдает запросы к модели. Затем сервлет передает управление представлению (JSP); оно генерирует код HTML, который возвращается и отображается в браузере.**

Паттерны проектирования и Модель 2

После реализации схемы веб-управления на базе Модели 2 так и хочется спросить: а куда же делись паттерны? Мы создаем представление HTML из JSP, но представление уже не является наблюдателем событий модели. У нас есть контроллер – сервлет, получающий запросы HTTP, но продолжаем ли мы использовать паттерн Стратегия? И как насчет паттерна Компоновщик? Представление состоит из кода HTML и отображается в браузере, не потерялся ли паттерн Компоновщик?

Модель 2: адаптация MVC для Интернета

Хотя Модель 2 не похожа на «классическую» архитектуру MVC, все составляющие остались на своих местах; просто они видоизменились в соответствии со спецификой браузерной модели. Присмотримся повнимательнее...

Наблюдатель

Представление уже не является наблюдателем модели в классическом смысле, то есть оно не регистрируется у модели для получения извещений об изменении состояния.

Однако представление косвенно получает аналоги оповещений от контроллера при изменении модели. Контроллер передает объект JavaBean, при помощи которого представление получает информацию состояния модели.

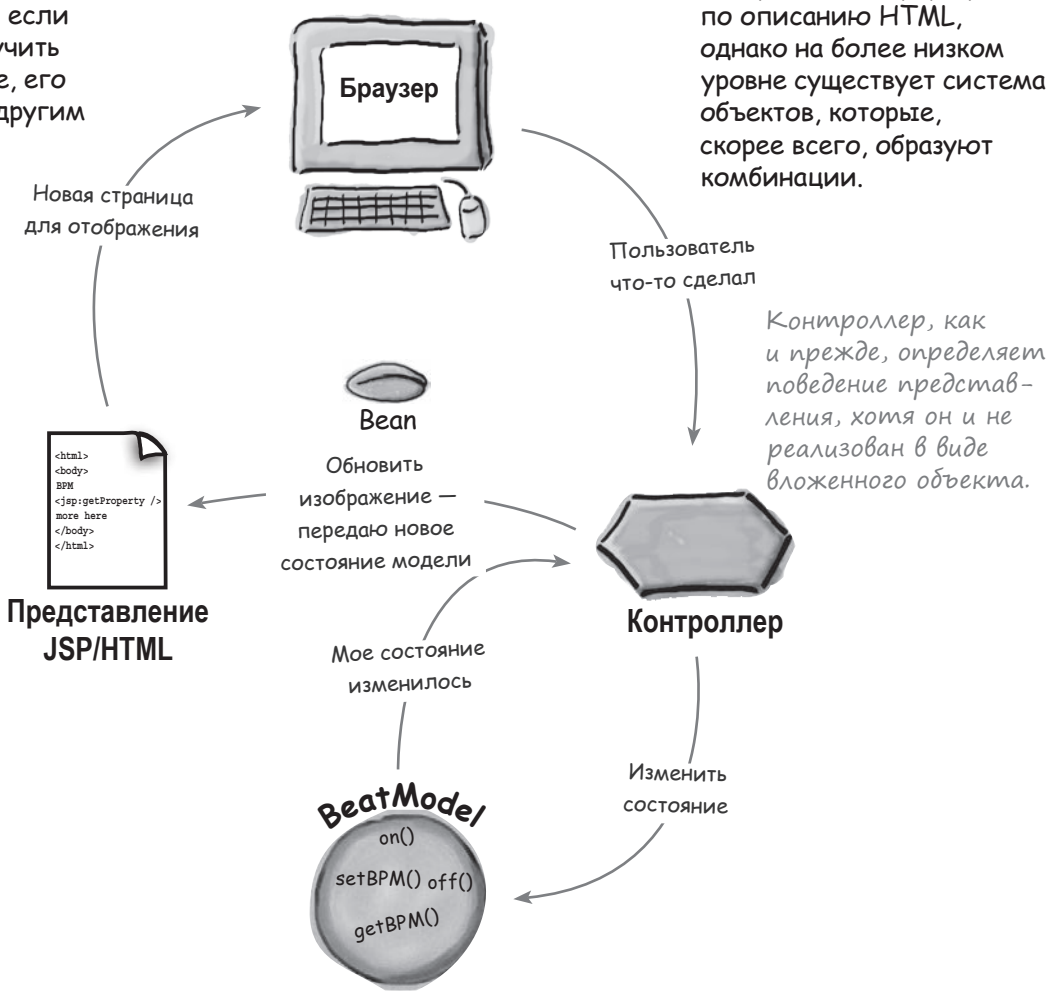
В этом проявляется специфика браузерной модели: обновленная информация состояния нужна представлению только при возвращении браузеру ответа HTTP, а в любой другой момент оповещение не имеет смысла. Только при создании и возвращении страницы создается представление, содержащее информацию состояния модели.

Стратегия

В Модели 2 объектом стратегии остается сервлет-контроллер, хотя он не связывается напрямую с представлением в классическом понимании. Однако при этом он остается объектом, реализующим поведение представления, и если потребуется получить другое поведение, его можно заменить другим контроллером.

Компоновщик

Как и в случае с Swing GUI, представление в конечном итоге строится из вложенных графических компонентов. В нашем примере они отображаются браузером по описанию HTML, однако на более низком уровне существует система объектов, которые, скорее всего, образуют комбинации.



Часто Задаваемые Вопросы

В: Вы так усиленно убеждаете нас, что паттерн Компоновщик на самом деле присутствует в MVC... А он точно здесь есть?

О: Да, паттерн Компоновщик присутствует в MVC. Но на самом деле это очень хороший вопрос. Современные GUI-пакеты — такие, как Swing, — стали настолько сложными, что мы уже не видим их внутренней структуры и роли комбинаций в построении и обновлении изображения. Комбинации еще труднее разглядеть при использовании браузеров, которые получают язык разметки и преобразуют его в пользовательский интерфейс.

В те времена, когда архитектура MVC была открыта впервые, создание графического интерфейса требовало значительной ручной работы, а роль паттерна была более очевидной.

В: Контроллер реализует хоть какую-нибудь прикладную логику?

О: Нет, контроллер реализует поведение представления. Именно он решает задачу преобразования действий с представлением в действия с моделью. Модель получает эти действия и реализует прикладную логику для принятия решения о том, как следует поступить в ответ на эти действия. Возможно, контроллеру придется проделать некую работу для определения того, какой метод модели следует вызвать, но это к «прикладной логике» не относится. Прикладной логикой называется код управления и обработки данных, и этот код находится в модели.

В: Вы часто упоминали состояние модели. Означает ли это, что в архитектуре MVC задействован паттерн Состояние?

О: Нет, мы говорили о состоянии вообще. Но, конечно, некоторые модели используют паттерн Состояние для управления своим внутренним состоянием.

В: Я видел описания MVC, в которых контроллер назывался «посредником» между представлением и моделью. Контроллер реализует паттерн Посредник?

О: Паттерн Посредник в книге еще не рассматривался (хотя его краткое описание приведено в приложении), поэтому сейчас мы не будем углубляться в подробности, но паттерн Посредник предназначен для инкапсуляции взаимодействий между объектами и устранения жестких связей, обусловленных явными ссылками объектов друг на друга. Таким образом, контроллер в определенной степени может рассматриваться как посредник, так как представление никогда не задает состояние непосредственно в модели, а всегда действует через контроллер. Однако следует помнить, что представление хранит ссылку на модель для получения информации состояния. Если бы контроллер был полноценным посредником, то представлению пришлось бы обращаться к контроллеру и за информацией состояния модели.

В: Представление всегда должно обращаться к модели за информацией состояния? Почему бы не использовать модель активной передачи и не отправлять состояние модели с оповещениями об обновлениях?

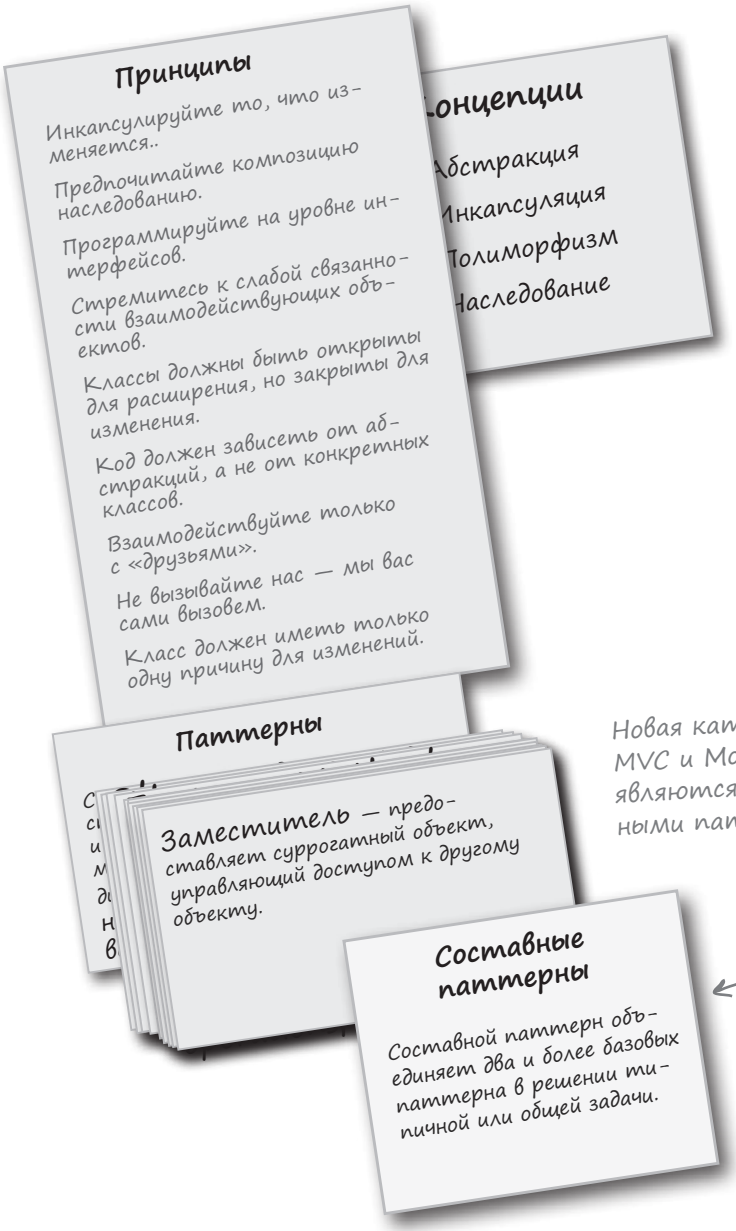
О: Да, безусловно, модель может отправлять свое состояние с оповещениями. Более того, если вы снова просмотрите представление JSP/HTML, то поймете, что именно это мы и делаем. Вся модель передается в форме объекта JavaBean, который используется представлением для обращения к состоянию. Нечто похожее можно было сделать и с BeatModel. Однако в главе, посвященной паттерну Наблюдатель, упоминалась пара недостатков такого подхода. Если вы забыли, о чем идет речь, — вернитесь и посмотрите еще раз.

В: Если в архитектуре задействовано несколько представлений, всегда ли в ней должно использоваться несколько контроллеров?

О: Как правило, во время выполнения для каждого представления создается один контроллер; тем не менее один класс контроллера может легко обслуживать сразу несколько представлений.

Новые инструменты

К этому моменту ваш инструментарий достиг впечатляющих размеров. Только посмотрите, сколько в нем всевозможных принципов и паттернов!



КЛЮЧЕВЫЕ МОМЕНТЫ

- Паттерн Модель-Представление-Контроллер (MVC) — составной паттерн, состоящий из паттернов Наблюдатель, Стратегия и Компоновщик.
- Модель использует паттерн Наблюдатель, чтобы наблюдатели оповещались об изменениях состояния, без формирования сильных связей.
- Контроллер определяет стратегию для представления. Представление может использовать разные реализации контроллера для обеспечения разного поведения.
- Представление использует паттерн Компоновщик для реализации пользовательского интерфейса, который обычно состоит из иерархии компонентов (панели, кнопки и т. д.).
- Совместная работа паттернов обеспечивает слабую связанность всех трех компонентов модели MVC, благодаря чему архитектура сохраняет гибкость и четкость.
- Модель 2 является адаптацией MVC для веб-приложений.
- В Модели 2 контроллер реализует в форме сервлета, а код JSP и HTML формирует представление.

Возьми в руку карандаш
Решение

QuackCounter также реализует Quackable. Когда мы изменяем Quackable, расширяя QuackObservable, нам придется изменить каждый класс, реализующий Quackable, в том числе и QuackCounter:

QuackCounter реализует Quackable, так что теперь он также реализует и QuackObservable.

```
public class QuackCounter implements Quackable {
    Quackable duck;
    static int numberOfQuacks;

    public QuackCounter(Quackable duck) {
        this.duck = duck;
    }

    public void quack() {
        duck.quack();
        numberOfQuacks++;
    }

    public static int getQuacks() {
        return numberOfQuacks;
    }

    public void registerObserver(Observer observer) {
        duck.registerObserver(observer);
    }

    public void notifyObservers() {
        duck.notifyObservers();
    }
}
```

Объект Duck, декорируемый QuackCounter.

Весь этот код остается таким же, как в предыдущей версии QuackCounter.

Два метода QuackObservable. Оба вызова просто делегируются декорируемому объекту Duck.

Возьми в руку карандаш
Решение

А если наблюдатель захочет понаблюдать за целой стаей? Как вообще это следует понимать? Понимайте так: наблюдение за комбинацией означает наблюдение за *каждым* ее элементом. Таким образом, при регистрации комбинация Flock должна позаботиться о регистрации всех своих дочерних элементов, среди которых могут быть другие комбинации.

Класс Flock реализует Quackable, поэтому теперь он также реализует QuackObservable.

```
public class Flock implements Quackable {
    ArrayList<Quackable> quackers = new ArrayList<Quackable>();

    public void add(Quackable duck) {
        ducks.add(duck);
    }

    public void quack() {
        Iterator<Quackable> iterator = quackers.iterator();
        while (iterator.hasNext()) {
            Quackable duck = iterator.next();
            duck.quack();
        }
    }

    public void registerObserver(Observer observer) {
        Iterator<Quackable> iterator = ducks.iterator();
        while (iterator.hasNext()) {
            Quackable duck = iterator.next();
            duck.registerObserver(observer);
        }
    }

    public void notifyObservers() { }
}
```

Объекты Quackable, входящие в контейнер Flock.

При регистрации Flock как наблдателя автоматически регистрируется все, что содержится во Flock, — то есть все реализации Quackable, будь то Duck или другой объект Flock.

Перебираем все реализации Quackables в Flock и делегируем вызов каждому объекту. Если реализация Quackable представляет собой Flock, то же самое происходит на следующем уровне.

Каждая реализация Quackable выполняет оповещение самостоятельно, поэтому Flock ничего делать не придется.

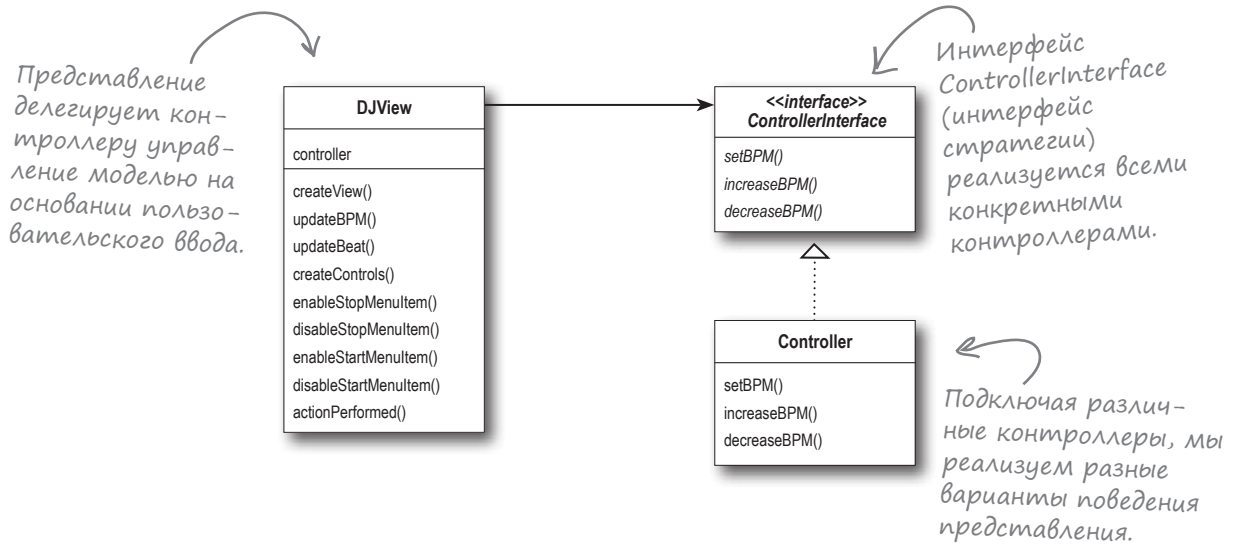
Возьми в руку карандаш
Решение

Однако экземпляры гусей по-прежнему создаются непосредственно, а код зависит от конкретных классов. Удастся ли вам написать абстрактную фабрику для гусей? Как она должна создавать «гусей, замаскированных под уток»?

Можно включить в существующую фабрику метод createGooseDuck(). А можно создать отдельную фабрику для Goose.

Головоломка
Решение

Представление и контроллер совместно используют паттерн Стратегия. Можете ли вы нарисовать для них диаграмму классов, представляющую этот паттерн?



Готово к употреблению

Далее приводится полная реализация DJView. В ней представлен весь MIDI-код, необходимый для генерирования звука, а также все компоненты Swing для создания представления. Этот код также можно загрузить на сайте http://www.headfirstlabs.com

```
package headfirst.designpatterns.combined.djview;

public class DJTestDrive {
    public static void main (String[] args) {
        BeatModelInterface model = new BeatModel();
        ControllerInterface controller = new BeatController(model);
    }
}
```

BeatModel

```
package headfirst.designpatterns.combined.djview;

public interface BeatModelInterface {
    void initialize();

    void on();

    void off();

    void setBPM(int bpm);

    int getBPM();

    void registerObserver(BeatObserver o);

    void removeObserver(BeatObserver o);

    void registerObserver(BPMObserver o);

    void removeObserver(BPMObserver o);
}
```

```
package headfirst.designpatterns.combined.djview;

import javax.sound.midi.*;
import java.util.*;
public class BeatModel implements BeatModelInterface, MetaEventListener {
    Sequencer sequencer;
    ArrayList<BeatObserver> beatObservers = new ArrayList<BeatObserver>();
    ArrayList<BPMObserver> bpmObservers = new ArrayList<BPMObserver>();
    int bpm = 90;
    Sequence sequence;
    Track track;

    public void initialize() {
        setUpMidi();
        buildTrackAndStart();
    }

    public void on() {
        System.out.println("Starting the sequencer");
        sequencer.start();
        setBPM(90);
    }

    public void off() {
        setBPM(0);
        sequencer.stop();
    }

    public void setBPM(int bpm) {
        this.bpm = bpm;
        sequencer.setTempoInBPM(getBPM());
        notifyBPMObservers();
    }

    public int getBPM() {
        return bpm;
    }

    void beatEvent() {
        notifyBeatObservers();
    }


    public void registerObserver(BeatObserver o) {
        beatObservers.add(o);
    }
```

Готово к употреблению

```
public void notifyBeatObservers() {
    for(int i = 0; i < beatObservers.size(); i++) {
        BeatObserver observer = (BeatObserver)beatObservers.get(i);
        observer.updateBeat();
    }
}

public void registerObserver(BPMObserver o) {
    bpmObservers.add(o);
}

public void notifyBPMObservers() {
    for(int i = 0; i < bpmObservers.size(); i++) {
        BPMObserver observer = (BPMObserver)bpmObservers.get(i);
        observer.updateBPM();
    }
}

public void removeObserver(BeatObserver o) {
    int i = beatObservers.indexOf(o);
    if (i >= 0) {
        beatObservers.remove(i);
    }
}

public void removeObserver(BPMObserver o) {
    int i = bpmObservers.indexOf(o);
    if (i >= 0) {
        bpmObservers.remove(i);
    }
}

public void meta(MetaMessage message) {
    if (message.getType() == 47) {
        beatEvent();
        sequencer.start();
        setBPM(getBPM());
    }
}

public void setUpMidi() {
    try {
        sequencer = MidiSystem.getSequencer();
        sequencer.open();
        sequencer.addMetaEventListener(this);
```

```
            sequence = new Sequence(Sequence.PPQ,4);
            track = sequence.createTrack();
            sequencer.setTempoInBPM(getBPM());
            sequencer.setLoopCount(Sequencer.LOOP_CONTINUOUSLY);
        } catch(Exception e) {
                e.printStackTrace();
        }
    }

     public void buildTrackAndStart() {
        int[] trackList = {35, 0, 46, 0};

        sequence.deleteTrack(null);
        track = sequence.createTrack();

        makeTracks(trackList);
        track.add(makeEvent(192,9,1,0,4));
        try {
            sequencer.setSequence(sequence);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

    public void makeTracks(int[] list) {

       for (int i = 0; i < list.length; i++) {
          int key = list[i];

          if (key != 0) {
             track.add(makeEvent(144,9,key, 100, i));
             track.add(makeEvent(128,9,key, 100, i+1));
          }
       }
    }

    public  MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
        MidiEvent event = null;
        try {
            ShortMessage a = new ShortMessage();
            a.setMessage(comd, chan, one, two);
            event = new MidiEvent(a, tick);

        } catch(Exception e) {
            e.printStackTrace();
        }
        return event;
    }
}
```

Представление

Готово к употреблению

```
package headfirst.designpatterns.combined.djview;

public interface BeatObserver {
    void updateBeat();
}
```

```
package headfirst.designpatterns.combined.djview;

public interface BPMObserver {
    void updateBPM();
}
```

```
package headfirst.designpatterns.combined.djview;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DJView implements ActionListener,  BeatObserver, BPMObserver {
    BeatModelInterface model;
    ControllerInterface controller;
    JFrame viewFrame;
    JPanel viewPanel;
    BeatBar beatBar;
    JLabel bpmOutputLabel;
    JFrame controlFrame;
    JPanel controlPanel;
    JLabel bpmLabel;
    JTextField bpmTextField;
    JButton setBPMButton;
    JButton increaseBPMButton;
    JButton decreaseBPMButton;
    JMenuBar menuBar;
    JMenu menu;
    JMenuItem startMenuItem;
    JMenuItem stopMenuItem;


    public DJView(ControllerInterface controller, BeatModelInterface model) {
        this.controller = controller;
        this.model = model;
        model.registerObserver((BeatObserver)this);
        model.registerObserver((BPMObserver)this);
    }
```

```
public void createView() {
    // Создание всех компонентов Swing
    viewPanel = new JPanel(new GridLayout(1, 2));
    viewFrame = new JFrame("View");
    viewFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    viewFrame.setSize(new Dimension(100, 80));
    bpmOutputLabel = new JLabel("offline", SwingConstants.CENTER);
    beatBar = new BeatBar();
    beatBar.setValue(0);
    JPanel bpmPanel = new JPanel(new GridLayout(2, 1));
    bpmPanel.add(beatBar);
    bpmPanel.add(bpmOutputLabel);
    viewPanel.add(bpmPanel);
    viewFrame.getContentPane().add(viewPanel, BorderLayout.CENTER);
    viewFrame.pack();
    viewFrame.setVisible(true);
}


public void createControls() {
    // Создание всех компонентов Swing
    JFrame.setDefaultLookAndFeelDecorated(true);
    controlFrame = new JFrame("Control");
    controlFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    controlFrame.setSize(new Dimension(100, 80));

    controlPanel = new JPanel(new GridLayout(1, 2));

    menuBar = new JMenuBar();
    menu = new JMenu("DJ Control");
    startMenuItem = new JMenuItem("Start");
    menu.add(startMenuItem);
    startMenuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            controller.start();
        }
    });
    stopMenuItem = new JMenuItem("Stop");
    menu.add(stopMenuItem);
    stopMenuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            controller.stop();
        }
    });
    JMenuItem exit = new JMenuItem("Quit");
    exit.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            System.exit(0);
        }
    });
```

Готово к употреблению

```
        menu.add(exit);
        menuBar.add(menu);
        controlFrame.setJMenuBar(menuBar);

        bpmTextField = new JTextField(2);
        bpmLabel = new JLabel("Enter BPM:", SwingConstants.RIGHT);
        setBPMButton = new JButton("Set");
        setBPMButton.setSize(new Dimension(10,40));
        increaseBPMButton = new JButton(">>");
        decreaseBPMButton = new JButton("<<");
        setBPMButton.addActionListener(this);
        increaseBPMButton.addActionListener(this);
        decreaseBPMButton.addActionListener(this);

        JPanel buttonPanel = new JPanel(new GridLayout(1, 2));

        buttonPanel.add(decreaseBPMButton);
        buttonPanel.add(increaseBPMButton);

        JPanel enterPanel = new JPanel(new GridLayout(1, 2));
        enterPanel.add(bpmLabel);
        enterPanel.add(bpmTextField);
        JPanel insideControlPanel = new JPanel(new GridLayout(3, 1));
        insideControlPanel.add(enterPanel);
        insideControlPanel.add(setBPMButton);
        insideControlPanel.add(buttonPanel);
        controlPanel.add(insideControlPanel);

        bpmLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
        bpmOutputLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

        controlFrame.getRootPane().setDefaultButton(setBPMButton);
        controlFrame.getContentPane().add(controlPanel, BorderLayout.CENTER);

        controlFrame.pack();
        controlFrame.setVisible(true);
    }

    public void enableStopMenuItem() {
        stopMenuItem.setEnabled(true);
    }

    public void disableStopMenuItem() {
        stopMenuItem.setEnabled(false);
    }
```

```
    public void enableStartMenuItem() {
        startMenuItem.setEnabled(true);
    }

    public void disableStartMenuItem() {
        startMenuItem.setEnabled(false);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == setBPMButton) {
            int bpm = Integer.parseInt(bpmTextField.getText());
            controller.setBPM(bpm);
        } else if (event.getSource() == increaseBPMButton) {
            controller.increaseBPM();
        } else if (event.getSource() == decreaseBPMButton) {
            controller.decreaseBPM();
        }
    }

    public void updateBPM() {
        int bpm = model.getBPM();
        if (bpm == 0) {
            bpmOutputLabel.setText("offline");
        } else {
            bpmOutputLabel.setText("Current BPM: " + model.getBPM());
        }
    }

    public void updateBeat() {
        beatBar.setValue(100);
    }
}
```

Контроллер

```
package headfirst.designpatterns.combined.djview;

public interface ControllerInterface {
    void start();
    void stop();
    void increaseBPM();
    void decreaseBPM();
    void setBPM(int bpm);
}
```

Готово к употреблению

```
package headfirst.designpatterns.combined.djview;

public class BeatController implements ControllerInterface {
    BeatModelInterface model;
    DJView view;

    public BeatController(BeatModelInterface model) {
        this.model = model;
        view = new DJView(this, model);
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
        model.initialize();
    }

    public void start() {
        model.on();
        view.disableStartMenuItem();
        view.enableStopMenuItem();
    }

    public void stop() {
        model.off();
        view.disableStopMenuItem();
        view.enableStartMenuItem();
    }

    public void increaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm + 1);
    }

    public void decreaseBPM() {
        int bpm = model.getBPM();
        model.setBPM(bpm – 1);
    }

    public void setBPM(int bpm) {
        model.setBPM(bpm);
    }
}
```

HeartModel

```
package headfirst.designpatterns.combined.djview;

public class HeartTestDrive {

    public static void main (String[] args) {
        HeartModel heartModel = new HeartModel();
        ControllerInterface model = new HeartController(heartModel);
    }
}

package headfirst.designpatterns.combined.djview;

public interface HeartModelInterface {
    int getHeartRate();
    void registerObserver(BeatObserver o);
    void removeObserver(BeatObserver o);
    void registerObserver(BPMObserver o);
    void removeObserver(BPMObserver o);
}

package headfirst.designpatterns.combined.djview;

import java.util.*;

public class HeartModel implements HeartModelInterface, Runnable {
    AArrayList<BeatObserver> beatObservers = new ArrayList<BeatObserver>();
    ArrayList<BPMObserver> bpmObservers = new ArrayList<BPMObserver>();
    int time = 1000;
    int bpm = 90;
    Random random = new Random(System.currentTimeMillis());
    Thread thread;

    public HeartModel() {
        thread = new Thread(this);
        thread.start();
    }

    public void run() {
        int lastrate = -1;

        for(;;) {
            int change = random.nextInt(10);
            if (random.nextInt(2) == 0) {
                change = 0 - change;
            }
            int rate = 60000/(time + change);
```

Готово к употреблению

```
            if (rate < 120 && rate > 50) {
                time += change;
                notifyBeatObservers();
                if (rate != lastrate) {
                    lastrate = rate;
                    notifyBPMObservers();
                }
            }
            try {
                Thread.sleep(time);
            } catch (Exception e) {}
        }
    }
    public int getHeartRate() {
        return 60000/time;
    }

    public void registerObserver(BeatObserver o) {
        beatObservers.add(o);
    }

    public void removeObserver(BeatObserver o) {
        int i = beatObservers.indexOf(o);
        if (i >= 0) {
            beatObservers.remove(i);
        }
    }

    public void notifyBeatObservers() {
        for(int i = 0; i < beatObservers.size(); i++) {
            BeatObserver observer = (BeatObserver)beatObservers.get(i);
            observer.updateBeat();
        }
    }

    public void registerObserver(BPMObserver o) {
        bpmObservers.add(o);
    }

    public void removeObserver(BPMObserver o) {
        int i = bpmObservers.indexOf(o);
        if (i >= 0) {
            bpmObservers.remove(i);
        }
    }

    public void notifyBPMObservers() {
        for(int i = 0; i < bpmObservers.size(); i++) {
            BPMObserver observer = (BPMObserver)bpmObservers.get(i);
            observer.updateBPM();
        }
    }
}
```

HeartAdapter

```
package headfirst.designpatterns.combined.djview;

public class HeartAdapter implements BeatModelInterface {
    HeartModelInterface heart;

    public HeartAdapter(HeartModelInterface heart) {
        this.heart = heart;
    }

    public void initialize() {}

    public void on() {}

    public void off() {}

    public int getBPM() {
        return heart.getHeartRate();
    }

    public void setBPM(int bpm) {}

    public void registerObserver(BeatObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BeatObserver o) {
        heart.removeObserver(o);
    }

    public void registerObserver(BPMObserver o) {
        heart.registerObserver(o);
    }

    public void removeObserver(BPMObserver o) {
        heart.removeObserver(o);
    }
}
```

Контроллер

Готово к употреблению

```
package headfirst.designpatterns.combined.djview;

public class HeartController implements ControllerInterface {
    HeartModelInterface model;
    DJView view;

    public HeartController(HeartModelInterface model) {
        this.model = model;
        view = new DJView(this, new HeartAdapter(model));
        view.createView();
        view.createControls();
        view.disableStopMenuItem();
        view.disableStartMenuItem();
    }

    public void start() {}

    public void stop() {}

    public void increaseBPM() {}

    public void decreaseBPM() {}

    public void setBPM(int bpm) {}
}
```

Паттерны в реальном мире

Вы стоите на пороге дивного нового мира, населенного паттернами проектирования. Но прежде чем открывать дверь, желательно изучить некоторые технические тонкости, с которыми вы можете столкнуться, — в реальном мире жизнь немного сложнее, чем здесь, в Объективиле. К счастью, у вас имеется хороший путеводитель, который упростит ваши первые шаги...

Руководство по эффективному использованию паттернов

Вашему вниманию предлагается руководство с полезными советами, которые помогут вам использовать паттерны в реальном программировании. В этом руководстве вы

- познакомитесь с распространенными заблуждениями относительно определения «паттерна проектирования»;
- узнаете, что такое каталоги паттернов и для чего они нужны;
- обойдете неприятности, связанные с несвоевременным использованием паттернов;
- изучите классификацию паттернов;
- увидите, что построение паттернов доступно не только экспертам; прочитайте нашу краткую сводку правил, и вы тоже сможете создавать свои паттерны;
- узнаете состав таинственной «Банды Четырех»;
- научитесь тренировать свой разум, как истинный мастер Дзен;
- освоите основную терминологию паттернов.

Определение паттерна проектирования

Вероятно, к концу книги вы уже достаточно хорошо представляете себе, что такое «паттерн проектирования». Однако мы нигде не приводили формальное определение паттерна проектирования. Возможно, широко распространенное определение вас слегка удивит:

> **Паттерн** – решение задачи в контексте.

Не самое понятное определение, вы не находите? Не беспокойтесь, мы разберем все его составляющие – все эти контексты, задачи и решения:

Контекстом называется ситуация, в которой применяется паттерн. Ситуация должна быть достаточно типичной и распространенной.

Пример: имеется коллекция объектов.

Задачей называется цель, которой вы хотите добиться в контексте, в совокупности со всеми ограничениями, присущими контексту.

Требуется перебрать объекты без раскрытия внутренней реализации коллекции.

Решением называется обобщенная архитектура, которая достигает заданной цели при соблюдении набора ограничений.

Перебор инкапсулируется в отдельном классе.

Возможно, вы не сразу усвоите суть этого определения; не торопитесь, продвигайтесь шаг за шагом.

Кто-то сочтет, что мы тратим слишком много времени, разбираясь с тем, что же такое «паттерн проектирования». В конце концов, мы уже знаем, что паттерны предназначены для решения типичных задач, возникающих в ходе проектирования. К чему такие формальности? Дело в том, что наличие формального механизма описания позволяет создавать чрезвычайно полезные *каталоги* паттернов.

Возможно, вы правы; давайте подумаем... Необходимыми компонентами паттерна являются *задача*, *решение* и *контекст*.

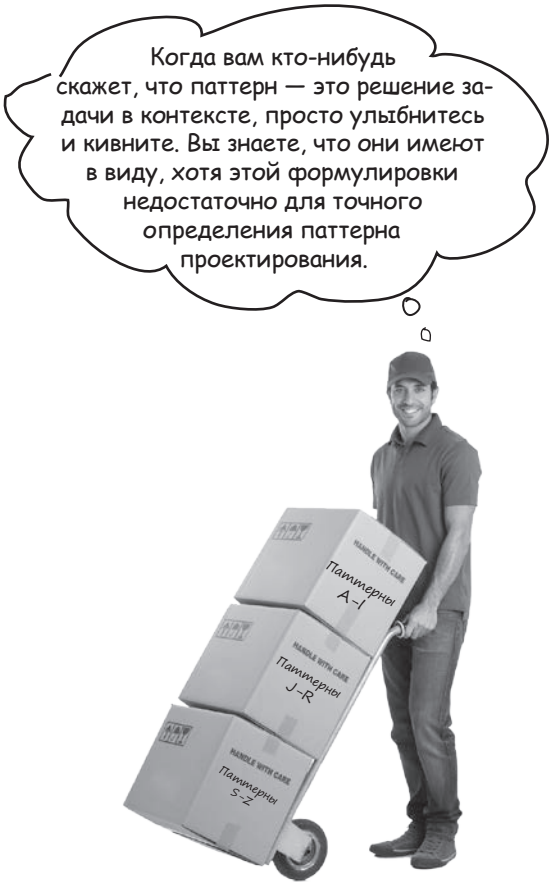
> **Задача:** Как мне во время попасть на работу?
>
> **Контекст:** Я нечаянно закрыл ключи в машине.
>
> **Решение:** Разбить окно, залезть в машину, запустить двигатель и поехать на работу.

В этом определении присутствуют все компоненты из определения: задача, которая включает в себя цель (попасть на работу) и ограничения по времени и расстоянию (и, вероятно, другие факторы). Также имеется контекст: недоступность ключей к машине. И есть решение, которое позволит нам добраться до ключей и справиться с ограничениями времени/расстояния. Мы создали паттерн! Верно?

МОЗГОВОЙ ШТУРМ

Следуя определению паттерна проектирования, мы определили задачу, контекст и решение (которое работает!). Можно ли назвать такое решение паттерном? А если нет, то почему? Возможны ли аналогичные ошибки при определении паттернов ОО-проектирования?

Подробнее об определении паттерна проектирования



Наш пример внешне соответствует определению, но назвать его паттерном нельзя. Почему? Прежде всего, мы знаем, что паттерн должен применяться для решения типичных, то есть часто встречающихся задач. Хотя рассеянный владелец может часто забывать ключи в машине, разбитое окно вряд ли можно назвать решением, которое будет применяться снова и снова (по крайней мере, если мы установим еще одно ограничение: затраты).

У нашего решения также есть пара других недостатков: во-первых, если вы возьмете это описание и передадите его другому человеку, у него могут возникнуть трудности с решением его конкретной проблемы. Во-вторых, мы нарушили важное, хотя и простое свойство паттернов: не присвоили ему имя! Без имени паттерн не станет частью общего «языка», понятного другим разработчикам.

К счастью, паттерны не описываются в простых понятиях задачи, контекста и решения. Существуют более эффективные способы описания паттернов и их объединения в *каталоги паттернов*.

Часто Задаваемые Вопросы

В: Так я не увижу описания паттернов, выраженные в понятиях задачи, контекста и решения?

О: Описания паттернов, приводимые в каталогах, обычно содержат более подробную информацию. В них больше внимания уделяется цели паттерна, причинам для его применения и ситуациям, в которых они уместны, а также архитектуре решения и возможным последствиям его применения (как положительным, так и отрицательным).

В: Можно ли слегка изменить паттерн, чтобы он лучше соответствовал моей архитектуре? Или необходимо четко придерживаться исходного определения?

О: Конечно, паттерны можно изменять. Как и принципы проектирования, паттерны не должны рассматриваться как непреложные правила; это *рекомендации*, которые вы можете изменять в соответствии со своими потребностями. Как вы уже видели, многие реальные примеры отличаются от классических определений.

В: Где найти каталог паттернов?

О: Первый и самый авторитетный каталог паттернов представлен в книге «Приемы объектно-ориентированного проектирования» (Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес, издательство «Питер»). В этом каталоге приведены 23 фундаментальных паттерна. Вскоре мы поговорим об этой книге более подробно.

Сейчас публикуется немало других каталогов паттернов для конкретных предметных областей: параллельных систем, бизнес-систем и т. д.

Для любознательных

Да пребудет с вами Сила

Из определения паттернов следует, что *задача* состоит из *цели* и *набора ограничений*. У экспертов в области паттернов существует специальный термин: они называют эти составляющие силами. Почему? Наверняка у них есть свои причины, но если вы помните фильм «Звездные войны», сила «определяет форму и управляет всем происходящим во Вселенной». Силы в паттернах тоже определяют форму и управляют решением. Только когда решение выдерживает баланс между двумя сторонами сил (светлая сторона — ваша цель, темная сторона — ограничения), можно сказать, что вы создали полезный паттерн.

Когда вы впервые встречаете термин «сила»при обсуждении паттернов, он выглядит довольно таинственно. Помните, что у сил есть две стороны (цели и ограничения), и что они должны быть сбалансированы для создания паттерна. Не бойтесь технического жаргона, и да пребудет с вами Сила!

Фрэнк: Расскажи, о чем идет речь, Джим. Я пока только прочитал несколько статей о паттернах.

Джим: В каталоге подробно описывается группа паттернов в контексте их отношений с другими паттернами.

Джо: Значит, существуют разные каталоги паттернов?

Джим: Ну конечно; в одних каталогах рассматриваются фундаментальные паттерны, а в других — паттерны конкретной предметной области (скажем, паттерны EJB).

Фрэнк: Какой каталог ты сейчас рассматриваешь?

Джим: Классический каталог «Банды Четырех»: в нем описаны 23 фундаментальных паттерна проектирования.

Фрэнк: «Банды Четырех»?

Джим: Да, именно. Это те парни, которые составили самый первый каталог паттернов.

Джо: И что в этом каталоге?

Джим: Набор взаимосвязанных паттернов. Для каждого паттерна приводится подробное описание, построенное по определенному шаблону. В частности, каждому паттерну присваивается *имя*.

Фрэнк: Просто потрясающе — имя! Кто бы мог подумать.

Джим: Не торопись, Фрэнк; имена важны. Они позволяют нам обсуждать этот паттерн в беседах с другими разработчиками; «единая номенклатура» и все такое.

Фрэнк: Ладно, ладно... Я пошутил. Что еще там есть?

Джим: Как я уже говорил, описания паттернов следуют определенному шаблону. Для каждого паттерна приводится имя и несколько разделов с дополнительной информацией о паттерне. Например, в разделе «Предназначение» рассказано, для чего применяется данный паттерн, а в разделах «Мотивация» и «Область применения» — где и когда данный паттерн может использоваться.

Джо: А как насчет самой архитектуры?

Джим: Далее описывается архитектура классов вместе с самими классами и их ролями. Также имеется раздел с описанием реализации паттерна, и довольно часто — с примерами кода, показывающими, как это делается.

Фрэнк: Похоже, они ничего не упустили.

Джим: Но это не всё. Также приводятся примеры использования паттерна в реальных системах, и раздел, который лично мне кажется самым полезным: отношение паттерна с *другими* паттернами.

Фрэнк: Где тебе объясняют, чем *состояние* отличается от *стратегии*?

Джим: Вот именно!

Джо: Скажи, а как ты работаешь с каталогом? Когда у тебя возникает проблема, ты заглядываешь в него в поисках решения?

Джим: Сначала я стараюсь ознакомиться со всеми паттернами и их отношениями. Когда мне понадобится паттерн, я уже в общих чертах представляю, что мне нужно. Я открываю книгу и в разделах «Мотивация» и «Область применения» убеждаюсь в том, что я правильно себе представляю происходящее. Также есть еще один важный раздел — «Последствия». Я всегда заглядываю в него, чтобы избежать непредвиденного влияния моего паттерна на архитектуру.

Фрэнк: Что ж, логично. И когда ты решаешь, что паттерн выбран правильно, как ты подходишь к его реализации и включению в архитектуру?

Джим: Я пользуюсь диаграммой классов. Сначала я читаю раздел «Структура» с обзором диаграммы, а затем раздел «Участники», чтобы убедиться в том, что я правильно понимаю роль каждого класса. Далее я адаптирую диаграмму для своей архитектуры и вношу те изменения, которые считаю нужными. Наконец, разделы «Реализация» и «Примеры кода» помогают убедиться в том, что я знаю обо всех полезных приемах или ловушках, которые могут встретиться на этом пути.

Джо: Похоже, каталог сделает мою работу с паттернами более эффективной!

Фрэнк: Безусловно. Джим, можешь показать нам пример описания паттерна?

Каждый паттерн в каталоге обладает именем. Без имени паттерн не сможет стать частью единой номенклатуры, понятной другим разработчикам.

Конкретный сценарий с описанием задачи и способа ее решения.

Ситуации, в которых применяется этот паттерн.

Классы и объекты, задействованные в архитектуре; описание их ролей и обязанностей в паттерне.

Описание возможных последствий от применения этого паттерна (как положительных, так и отрицательных).

Приемы, которые понадобятся для реализации этого паттерна, а также общие аспекты, на которые следует обратить внимание.

Примеры использования паттерна в реальных системах.

Категория, к которой относится паттерн. Мы поговорим о категориях чуть позднее.

Короткое описание сути паттерна. Также может рассматриваться как определение (аналог определений, приводимых в этой книге).

Диаграмма отношений между классами, участвующими в паттерне.

Схема взаимодействия участников паттерна.

Фрагменты кода, которые могут пригодиться в вашей реализации.

Описание отношений между этим и другими паттернами.

Одиночка

Object Creational

Предназначение

Et aliquat, velesto ent lore feuis acillao rperci tat, quat nonsequam il ea at nim nos do enim qui eratio ex ea faci tet, sequis dion utat, volore magnisi.

Мотивация

Et aliquat, velesto ent lore feuis acillao rperci tat, quat nonsequam il ea at nim nos do enim qui eratio ex ea faci tet, sequis dion utat, volore magnisi.Rud modolore dit laoreet augiam iril el dipis dionsequis dignibh eummy nibh esequat. Duis nulputem ipisim esecte conullut wissi.

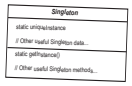
Os nisissenim et lumsandre do con el utpatuero corercipis augue doloreet luptat amet vel iusciduns digna feugue dunt num etummy nim dui blaor sequat num vel etue magna augiat.

Aliquis nonse vel exer se minissequis do dolortis ad magnit, sim zzrillut ipsummo dolorem dignibh euguer sequam ea am quate magnim illam zzrit ad magna feu facinit delit ut

Область применения

Duis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Структура



Участники

Duis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er

- A dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er
  - A feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissec
  - Ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit

Взаимодействия

- Feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore.

Последствия

Duis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre:

1. Dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.
2. Modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem.
3. Dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.
4. Modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem.

Реализация/Примеры кода

DuDuis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

Nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Известные применения

DuDuis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

DuDuis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er. alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Связанные паттерны

Elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er. alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Часто Задаваемые Вопросы

В: Можно ли создавать собственные паттерны? Или на это способны только «гуру» в области паттернов?

О: Прежде всего запомните, что паттерны *открываются*, а не создаются. Кто угодно может найти новый паттерн и составить его описание; тем не менее это довольно трудно, и такие открытия происходят нечасто. Работа первооткрывателя паттернов требует целеустремленности и терпения.

Прежде всего подумайте, зачем вам это нужно. Большинство разработчиков не *определяет* паттерны, а использует их. Но возможно, вы работаете в специализированной области, в которой новые паттерны могли бы пригодиться, или обнаружили решение проблемы, которая, на ваш взгляд, является типичной; или просто хотите участвовать в жизни сообщества паттернов.

В: Я готов! С чего начинать?

О: Как и в любой научной дисциплине, чем больше вы знаете — тем лучше. Изучайте существующие паттерны, разбирайтесь в том, как они работают и как связаны с другими паттернами. Это очень важно — вы не только поймете, как создаются паттерны, но и не будете «изобретать велосипед».

В: Как я узнаю, что у меня действительно получился паттерн?

О: Вы можете быть уверены в этом только после того, как другие разработчики опробуют ваше решение и убедятся в его работоспособности. В общем случае следует руководствоваться «правилом трех»: оно гласит, что паттерн может быть признан таковым только после того, как он будет применен в реальных программах не менее трех раз.

Хотите быть экспертом по созданию паттернов? Тогда слушайте, что я вам скажу.

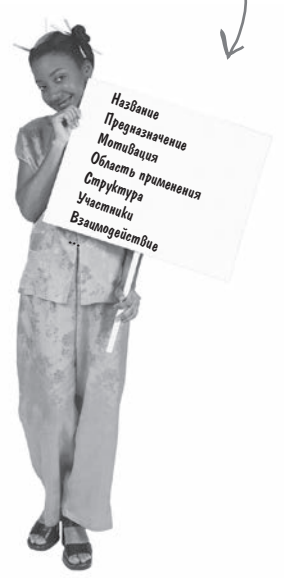
Выберите каталог паттернов и не жалейте времени на его изучение. А когда вы сможете правильно сформулировать определение и три разработчика воспользуются вашим решением, значит, это действительно паттерн.

Хотите создавать паттерны?

Изучайте матчасть. Прежде чем создать новый паттерн, необходимо хорошо разбираться в уже существующих паттернах. Многие паттерны, которые кажутся новыми, в действительности представляют собой разновидности существующих паттернов. Изучая паттерны, вы научитесь узнавать их и связывать с другими паттернами.

Анализируйте и оценивайте. Идеи паттернов рождаются из практического опыта — задач, с которыми вы сталкиваетесь, и примененных вами решений. Выделите время на то, чтобы проанализировать свой опыт и преобразовать его в новаторские решения типичных задач. Помните, что большинство архитектурных решений строится на разновидностях существующих паттернов, а не на новых. И даже когда вы обнаруживаете действительно новое решение, обычно выясняется, что его область применения слишком узка, чтобы его можно было признать полноценным паттерном.

Определите свой паттерн по одному из существующих шаблонов. Эти шаблоны были тщательно продуманы, а их формат хорошо знаком разработчикам, пользующимся паттернами.



Изложите свои идеи на бумаге. Изобретение нового паттерна не принесет особой пользы, если другие не смогут воспользоваться вашей находкой; вы должны документировать потенциальный паттерн, чтобы другие люди могли прочитать ваше описание, понять и применить его в своем решении, а потом предоставить обратную связь. К счастью, вам не придется изобретать собственную систему записи паттернов. Как было показано на примере шаблона «Банды Четырех», процесс описания паттернов и их характеристик хорошо проработан.

Предложите другим использовать ваши паттерны, улучшите их... и продолжайте улучшать. Не надейтесь, что паттерн будет правильно сформулирован с первого раза. Относитесь к своим паттернам как к текущей работе, которая должна совершенствоваться со временем. Пусть другие разработчики проанализируют ваше предложение, опробуют его и выскажут свое мнение. Включите данные обратной связи в описание и попробуйте снова. Ваше описание никогда не окажется идеальным, но в какой-то момент оно будет достаточно хорошо проработано, чтобы другие разработчики смогли прочитать и понять его.

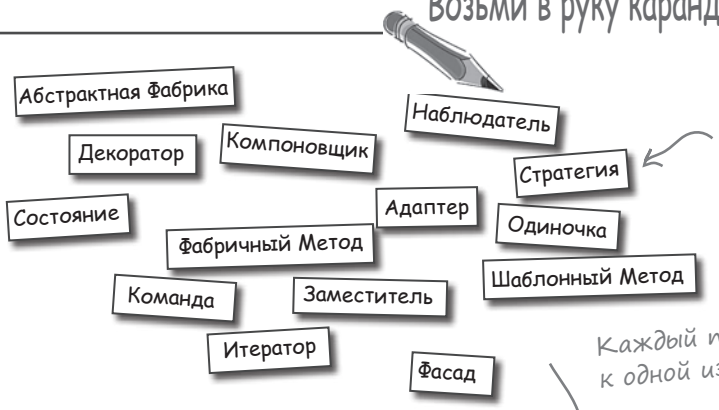
Соедините каждый паттерн с его описанием:

Паттерн	Описание
Декоратор	Упаковывает объект и предоставляет другой интерфейс к нему.
Состояние	Субклассы решают, как реализовать шаги в алгоритме.
Итератор	Субклассы решают, какие конкретные классы должны создаваться.
Фасад	Обеспечивает создание одного (и только одного!) экземпляра класса.
Стратегия	Инкапсулирует взаимозаменяемые варианты поведения и выбирает один из них посредством делегирования.
Заместитель	Клиент выполняет однородные операции с объектами и коллекциями.
Фабричный Метод	Инкапсулирует поведение, связанное с состоянием, с делегированием поведения объекту текущего состояния.
Адаптер	Обеспечивает механизм перебора коллекций объектов без раскрытия реализации.
Наблюдатель	Упрощает интерфейс группы классов.
Шаблонный Метод	Упаковывает объект для реализации нового поведения.
Компоновщик	Позволяет клиенту создавать семейства объектов без указания их конкретных классов.
Одиночка	Обеспечивает оповещение объектов об изменении состояния.
Абстрактная Фабрика	Упаковывает объект для управления доступом к нему.
Команда	Инкапсулирует запрос в виде объекта.

Классификация паттернов проектирования

С ростом количества общепринятых паттернов появляется необходимость в их классификации, чтобы мы могли структурировать их, сократить поиск до небольшого подмножества паттернов и выполнять сравнение внутри групп. В большинстве каталогов используется одна из нескольких общепринятых схем классификации. Самая распространенная схема, выбранная в самом первом каталоге паттернов, разбивает паттерны на три категории в зависимости от их цели: порождающие, поведенческие и структурные.

Возьми в руку карандаш

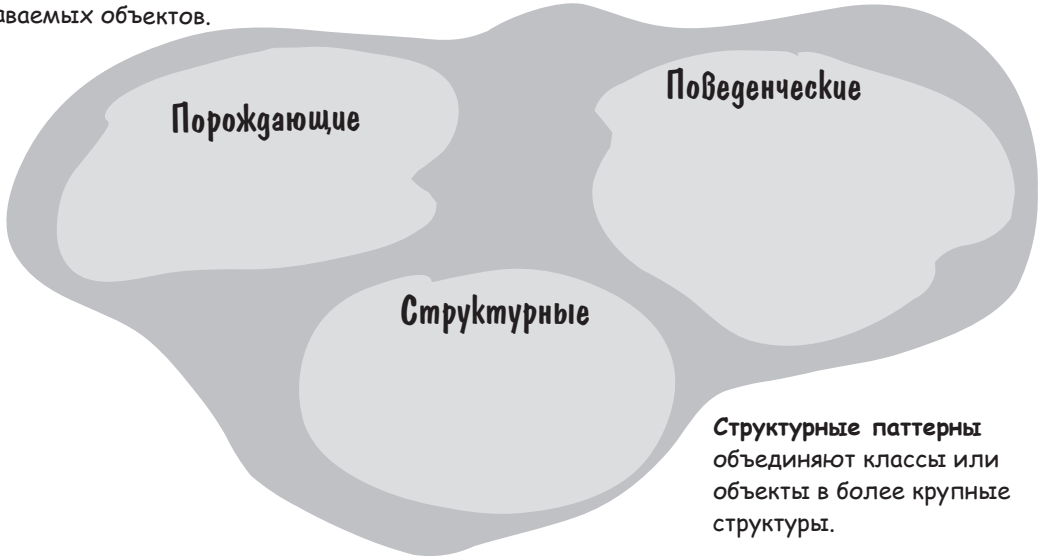


Прочитайте описание каждой категории и попробуйте распределить паттерны по категориям. Задача не из простых! Приложите максимум усилий и сверьтесь с ответами на следующей странице.

Каждый паттерн относится к одной из этих категорий.

Порождающие паттерны связаны с созданием экземпляров объектов; все они обеспечивают средства логической изоляции клиента от создаваемых объектов.

Паттерны, принадлежащие к **поведенческой категории,** относятся к взаимодействиям и распределению обязанностей между классами и объектами.



Структурные паттерны объединяют классы или объекты в более крупные структуры.

Возьми в руку карандаш
Решение

Классификация паттернов

Перед вами паттерны, сгруппированные по категориям. Вероятно, это упражнение было довольно трудным, потому что многие паттерны можно отнести сразу к нескольким категориям. Не беспокойтесь, проблемы с выбором правильной категории возникают у всех разработчиков.

Порождающие паттерны связаны с созданием экземпляров объектов; все они обеспечивают средства логической изоляции клиента от создаваемых объектов.

Паттерны, относящиеся к **поведенческой категории,** относятся к взаимодействиям и распределению обязанностей между классами и объектами.

Серым цветом выделены паттерны, которых мы еще не видели. Краткий обзор этих паттернов приведен в приложении.

Структурные паттерны объединяют классы или объекты в более крупные структуры.

Паттерны также часто классифицируются по другому атрибуту: в зависимости от того, относится паттерн к классам или объектам.

Паттерны классов описывают определение отношений между классами посредством наследования. Отношения в паттернах классов определяются на стадии компиляции.

Паттерны объектов описывают отношения между объектами, прежде всего относящиеся к композиции. Отношения в паттернах объектов обычно определяются на стадии выполнения, а следовательно, обладают большей динамичностью и гибкостью.

Обратите внимание: паттернов объектов намного больше, чем паттернов классов!

Часто Задаваемые Вопросы

В: Существуют только эти схемы классификации?

О: Нет, есть и другие. Некоторые схемы начинаются с трех основных категорий, а затем вводят субкатегории (скажем, «Паттерны логической изоляции»). Желательно знать основные схемы организации паттернов, но ничто не мешает вам создавать собственные схемы, если они помогут вам лучше понять паттерны.

В: Деление паттернов на категории действительно помогает их запомнить?

О: Бесспорно, классификация формирует основу для сравнения. Однако многие разработчики путаются в категориях порождающих, структурных и поведенческих паттернов: многие паттерны могут быть отнесены к нескольким категориям. Главное — знать сами паттерны и отношения между ними. Но если классификация вам поможет, используйте ее!

В: Почему паттерн Декоратор отнесен к структурной категории? На мой взгляд, это скорее поведенческий паттерн, ведь он добавляет новое поведение!

О: Да, так считают многие разработчики! Классификация «Банды Четырех» базировалась на следующих соображениях: структурные паттерны описывают способы объединения классов и объектов для создания новых структур и новой функциональности. Паттерн Декоратор упаковывает один объект в другой для расширения его функциональности. Таким образом, его основной сутью является динамическое объединение объектов для расширения функциональности, а не взаимодействия между объектами, являющиеся содержанием поведенческих паттернов. Следует помнить о различиях между целями этих паттернов; в них часто кроется ключ к пониманию того, к какой категории следует отнести тот или иной паттерн.

Учитель и Ученик...

Учитель: *Ты чем-то обеспокоен?*

Ученик: *Да, я только что узнал о классификации паттернов, и мой разум пребывает в смятении.*

Учитель: *Продолжай...*

Ученик: *После всего, что я узнал о паттернах, мне говорят, что каждый паттерн относится к одной из трех категорий: порождающие, структурные и поведенческие. Зачем нужна эта классификация?*

Учитель: *Имея дело с большой коллекцией чего угодно, мы естественным образом ищем категории для классификации ее элементов, чтобы рассматривать их на более абстрактном уровне.*

Ученик: *Учитель, вы можете привести пример?*

Учитель: *Конечно. Возьмем машины; существует много разных моделей, а мы делим их на категории: малолитражки, спортивные, внедорожники, грузовики и так далее... Тебя что-то удивляет?*

Ученик: *Учитель, я потрясен тем, что вы столько знаете о машинах!*

Учитель: *Нельзя же* ***все*** *сравнивать с цветком лотоса или чашкой риса. Я могу продолжать?*

Ученик: *Да-да, прошу прощения, продолжайте.*

Учитель: *Наличие классификации упрощает описание различий между разными категориями: «Если ты собираешься ехать из Кремниевой Долины в Санта-Крус по горной дороге, лучшим вариантом будет спортивная машина с хорошим управлением». Или: «С нынешними ценами на бензин лучше купить малолитражку, они эффективнее расходуют топливо».*

Ученик: *Следовательно, категории позволяют нам говорить о группе паттернов как о едином целом. Допустим, мы знаем, что нам нужен порождающий паттерн, — хотя и не знаем, какой именно.*

Учитель: *Да, а также сравнивать элемент с остальными элементами категории («Мини —*

самая стильная машина из малолитражек») и сужать поиск («Мне нужна машина с эффективным расходом топлива»).

Ученик: *Понимаю. Получается, я могу сказать, что Адаптер — лучший структурный паттерн для изменения интерфейса объекта.*

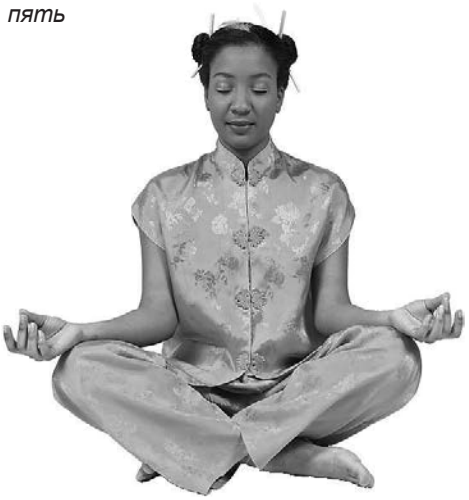
Учитель: *Да. Категории также используются еще для одной цели: для планирования выхода в новые области (скажем, «Мы хотим создать спортивный автомобиль с характеристиками Ferrari и по цене Miata»).*

Ученик: *Так не бывает.*

Учитель: *Извини, ты что-то сказал?*

Ученик: *Я говорю: «Да, понятно». Значит, категории помогают нам уяснить отношения между группами паттернов и между паттернами в пределах одной группы. Кроме того, они упрощают экстраполяцию новых паттернов. Но почему именно три категории — не четыре и не пять?*

Учитель: *Категории подобны звездам в ночном небе — их столько, сколько ты захочешь видеть. Три — удобное число. Многие люди считают, что такая классификация паттернов оптимальна. Впрочем, другие предпочитают четыре, пять и более категорий.*



Мыслить паттернами

Контексты, ограничения, силы, каталоги, классификация... Начинает отдавать академической скукой, вы не находите? Конечно, все это важно, а *знание — сила*, но давайте признаем: даже если вы обладаете академическими познаниями, но не имеете *опыта* и практики использования паттернов, это никак не повлияет на вашу жизнь.

Далее приводится краткое руководство, которое поможет вам научиться *мыслить паттернами*. Что мы имеем в виду? Состояние, в котором вы смотрите на архитектуру и сразу находите в ней естественные возможности для применения паттернов.

Мозг, мыслящий паттернами

Будьте проще

Прежде всего при проектировании следует использовать самые простые решения. Вашей целью должна быть простота, а не «как бы применить паттерн в этой задаче». Не стоит думать, что без использования паттернов вас не будут считать грамотным специалистом. Другие разработчики по достоинству оценят простоту архитектуры. Впрочем, иногда самые простые и гибкие решения основаны на применении паттернов.

Паттерны — не панацея

Как вы знаете, паттерны представляют собой общие решения типичных задач. Важное преимущество паттернов заключается в том, что они были проверены многими разработчиками. Таким образом, когда вы видите возможность для применения паттерна, будьте уверены: многие разработчики уже побывали в вашей ситуации и решили задачу аналогичными средствами.

И все же паттерны — не панацея. Не надейтесь, что вы включите паттерн в свое решение, откомпилируете программу и отправитесь на обед пораньше. Чтобы использовать паттерн, необходимо продумать все последствия для остальных аспектов архитектуры.

Используйте паттерн, когда...

Самый важный вопрос: когда использовать паттерн? Тогда, когда вы уверены, что он необходим для решения задачи из вашей архитектуры. Если может сработать более простое решение, проанализируйте эту возможность до выбора паттерна.

Понять, когда следует применять паттерны, — в этом вам помогут опыт и знания. Если вы твердо уверены, что простое решение не соответствует вашим потребностям, проанализируйте задачу вместе с ограничениями на решение — это упростит выбор паттерна для задачи. Если вы хорошо разбираетесь в паттернах, возможно, вам уже известен хороший кандидат, а если нет — переберите паттерны, которые могли бы подойти для нее. В этом вам пригодятся разделы «Предназначение» и «Область применения». Обнаружив подходящий паттерн, убедитесь в том, что его последствия приемлемы в вашей ситуации, и проанализируйте его влияние на другие аспекты архитектуры. И если все хорошо — действуйте!

В одной ситуации паттерны используются даже при наличии более простого решения: если вы ожидаете, что некоторые аспекты вашей системы будут изменяться. Как вы уже знаете, идентификация областей возможных изменений в архитектуре часто является верным признаком необходимости применения паттернов. Только следите за тем, чтобы паттерны были ориентированы на *реальные*, а не на чисто теоретические изменения.

Рефакторинг — время применения паттернов!

Рефакторингом называется процесс изменения кода для совершенствования его структуры. Он направлен на улучшение организации кода, а не на изменение поведения. Рефакторинг идеально подходит для анализа архитектуры и возможностей улучшения ее структуры с применением паттернов. Например, обилие условных конструкций может свидетельствовать о необходимости применения паттерна Состояние. А может быть, пришло время избавиться от привязки к конкретным классам при помощи паттерна Фабрика. На тему рефакторинга с использованием паттернов были написаны целые книги, и по мере накопления практического опыта вам стоит дополнительно изучить эту тему.

Убирайте то, без чего можно обойтись. Не бойтесь исключать паттерны из своей архитектуры.

Никто не говорит о том, когда следует исключать паттерны из архитектуры — словно это какое-то святотатство! Но мы взрослые люди, мы это переживем.

В каких случаях паттерны следует исключать из архитектуры? Если ваша система стала чрезмерно сложной, а изначально запланированная гибкость оказалась излишней... Проще говоря, когда более простое решение предпочтительно.

Не делайте сейчас то, что может понадобиться потом.

Паттерны проектирования обладают большими возможностями. Разработчики любят создавать красивые архитектуры, готовые к изменениям в любом направлении.

Боритесь с искушением. Если поддержка изменений в архитектуре обусловлена практической необходимостью, используйте паттерн, который эти изменения упростит. Но если изменения являются чисто гипотетическими, паттерн лишь приведет к напрасному усложнению системы.

Учитель и Ученик...

Учитель: *Твое обучение почти завершено. Чем ты собираешься заняться?*

Ученик: *Съезжу в Диснейленд! А потом буду писать длинный код с паттернами!*

Учитель: *Постой, не торопись. Оружие не следует применять без необходимости.*

Ученик: *Что вы имеете в виду? Я потратил столько сил на изучение паттернов, а теперь не должен использовать их в своих разработках для достижения максимальной мощи, гибкости и удобства управления?*

Учитель: *Нет. Паттерны — инструмент, который следует применять только при необходимости. Ты потратил много времени на изучение принципов проектирования. Всегда начинай с принципов и пиши самый простой код, который позволит решить задачу. Но если ты видишь необходимость в использовании паттерна — тогда и используй его.*

Ученик: *Значит, я не должен строить свои архитектуры на базе паттернов?*

Учитель: *Это не должно быть твоей изначальной целью. Пусть паттерны появляются естественным образом в ходе проектирования.*

Ученик: *Если паттерны так хороши, почему я должен осторожничать с их использованием?*

Учитель: *Паттерны усложняют архитектуру, а лишняя сложность никому не нужна. С другой стороны, при правильном применении паттерны обладают выдающейся мощью. Как известно, в них воплощен проверенный временем опыт проектирования, который помогает избежать типичных ошибок. Кроме того, паттерны определяют единую номенклатуру для обсуждения концепций проектирования с другими разработчиками.*

Ученик: *Так когда же в архитектуре уместно применять паттерны?*

Учитель: *Тогда, когда ты уверен в их необходимости для решения текущей задачи — или в их необходимости для адаптации будущих изменений в требованиях к приложению.*

Ученик: *Похоже, мне еще придется многому научиться, хотя я уже понимаю многие паттерны.*

Учитель: *Да, умению адаптироваться к сложности и изменениям приходится учиться всю жизнь. Но теперь ты знаешь довольно много паттернов и сможешь применять их в своих архитектурах, продолжая изучение других паттернов.*

Ученик: *Погодите, выходит, я знаю НЕ ВСЕ паттерны?*

Учитель: *Ты узнал лишь самые основные паттерны, однако их гораздо больше — включая паттерны специализированных областей (параллельного программирования, enterprise-систем и т. д.). Но теперь, когда ты владеешь основами, тебе будет намного проще изучать их!*

Разум и паттерны

Разум новичка

«Мне нужен паттерн для программы „Hello World“».

Новичок использует паттерны повсюду. И это не так плохо: он накапливает опыт их практического применения. Кроме того, новичок думает: «Чем больше паттернов я использую, тем лучше будет архитектура». Со временем он узнает, что это не так, а архитектуры должны быть по возможности простыми. Сложность и паттерны уместны только тогда, когда они реально необходимы для решения будущих проблем расширения.

Разум опытного разработчика

«Вероятно, здесь можно использовать Адаптер».

По мере обучения разум опытного разработчика начинает понимать, где паттерны уместны, а где нет. Разработчик все еще нередко выбирает неподходящие паттерны для решения тех или иных задач, но он уже понимает, что паттерны можно адаптировать к тем ситуациям, для которых канонические определения не подойдут.

Просветленный разум

«Естественная ситуация для паттерна Декоратор».

Просветленный разум видит паттерны там, где их применение наиболее естественно. Просветленный разум не зациклен на использовании паттернов; он ищет простые решения, которые лучше всего подходят для конкретной задачи. Он мыслит объектно-ориентированными принципами и их достоинствами/недостатками. Обнаружив естественную необходимость в использовании паттерна, просветленный разум применяет его, адаптируя при необходимости. *Просветленный разум сродни разуму новичка* — он не позволяет своим знаниям о паттернах слишком сильно влиять на архитектурные решения.

ВНИМАНИЕ!!! Злоупотребление паттернами проектирования приводит к чрезмерному усложнению кода. Всегда выбирайте самое простое решение и используйте паттерны только в случае необходимости.

Конечно, мы хотим, чтобы вы использовали паттерны!

Но мы еще больше хотим, чтобы вы были хорошим ОО-проектировщиком.

Когда архитектура требует применения паттерна, вы пользуетесь решением, проверенным многими разработчиками. Ваше решение хорошо документировано, к тому же оно будет понятно другим разработчикам (помните: единая номенклатура и все такое).

Однако у паттернов проектирования существует и оборотная сторона. Паттерны часто вводят в решение дополнительные классы и объекты, а иногда и увеличивают сложность ваших архитектур. Кроме того, в решении могут появиться дополнительные логические уровни, которые отражаются не только на сложности, но и на эффективности решения.

Наконец, применение паттернов иногда оказывается явным «перебором». Руководствуясь принципами проектирования, можно найти более простое решение исходной задачи. Если вы оказались в такой ситуации — не сопротивляйтесь. Используйте более простое решение.

Но пусть все сказанное не погасит ваш энтузиазм. Паттерн проектирования, правильно выбранный для конкретной задачи, обладает множеством преимуществ.

И не забудьте о единстве номенклатуры

В этой книге так долго обсуждались технические тонкости ОО-проектирования, что за этим занятием можно было легко упустить «человеческую» сторону паттернов: они не только предоставляют готовые решения, но и формируют единую номенклатуру, понятную другим разработчикам. Не стоит недооценивать силу единой номенклатуры, это одно из *величайших преимуществ* паттернов проектирования.

С того момента, когда мы в последний раз говорили о единстве номенклатуры, произошло нечто важное: у вас появилась своя номенклатура. Кроме того, изученный вами полный набор паттернов ОО-проектирования поможет легко понять мотивацию и принципы работы любых новых паттернов, с которыми вы столкнетесь.

Теперь, когда вы овладели основами паттернов проектирования, начинайте распространять свои знания среди коллег. Почему? Потому что, когда ваши коллеги будут знать паттерны и владеть той же единой номенклатурой, это улучшит качество проектирования, повысит эффективность общения и, что самое ценное, сэкономит немало времени, которое можно потратить с большей пользой.

И тогда я создала класс, который ведет список всех объектов-слушателей, и при поступлении новых данных отправляет сообщение каждому слушателю. Причем слушатели могут в любой момент присоединяться к рассылке или отсоединяться от нее. А в качестве слушателя может зарегистрироваться любой объект, реализующий нужный интерфейс.

Неполно

Невразумительно

Пространно

Пять способов использования единой номенклатуры

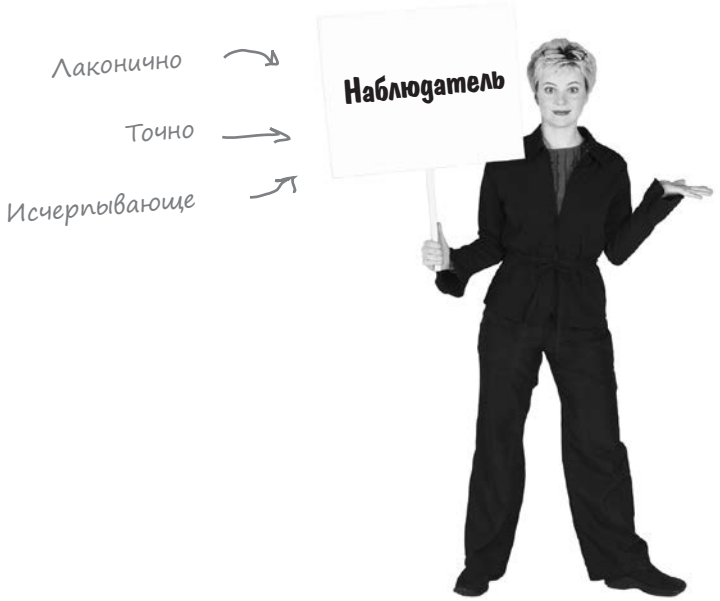
1. **На обсуждениях архитектуры:** Когда группа встречается для обсуждения архитектуры продукта, паттерны проектирования помогут вам оставаться в курсе происходящего. Обсуждение архитектуры с точки зрения паттернов и ОО-принципов поможет вам не увязнуть в подробностях реализации, а также предотвратит многие недоразумения.

2. **С другими разработчиками:** Использование паттернов в общении с другими разработчиками способствует формированию сообщества и распространению информации о паттернах. И конечно, самое приятное, когда вы учите чему-то других — услышать от собеседника «Ага, я понял!»

3. **В документации:** Когда вы описываете свою архитектуру в документации, упоминание паттернов уменьшает объем текста и дает читателю более четкое понимание архитектуры.

4. **В комментариях и именах:** При написании кода следует упоминать использованные паттерны в комментариях. Также паттерны должны отражаться в именах классов и методов. Другие разработчики, читающие ваш код, будут благодарны вам за то, что вы помогли им так быстро разобраться в вашей реализации.

5. **Среди заинтересованных разработчиков:** Делитесь знаниями. Многие разработчики что-то слышали о паттернах, но плохо понимают, о чем идет речь. Вызовитесь провести семинар, посвященный паттернам, или выступите с докладом на собрании.



Прогулка по Объективилю с «Бандой Четырех»

«Банда Четырех» породила движение паттернов программирования, но значительный вклад в него внесли и другие люди, в числе которых Уорд Каннингем, Кент Бек, Джим Коплин, Грэди Буч, Брюс Андерсон, Ричард Гэбриел, Дуг Ли, Питер Коуд и Дуг Шмидт.

В Объективиле нет рокеров и уличных хулиганов, но зато в нем есть «Банда Четырех». Как вы, вероятно, уже заметили, в мире паттернов трудно далеко уйти, не встретившись с ними. Кто же эти таинственные люди?

В двух словах, «Банда Четырех» — Эрик Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес — это люди, которые составили первый каталог паттернов, а попутно породили новое движение в области программирования!

Откуда взялось это название? Точно не известно, но оно прижилось. Впрочем, участники этой «Банды» — на редкость славные люди. Они даже согласились встретиться с нами и поделиться полезными советами...

* Джона Влиссидеса не стало в 2005 г. Большая потеря для профессионального сообщества.

Наше путешествие только начинается...

Что ж, ну а теперь, когда вы изучили эту книгу и готовы развиваться дальше, мы можем порекомендовать еще три издания, которые просто необходимо иметь на книжной полке любому программисту, занимающемуся разработкой паттернов.

Классическая книга о паттернах проектирования

Эта книга, изданная в далеком 1995 году, послужила основным толчком роста популярности паттернов проектирования. Здесь вы найдете все базовые и классические шаблоны. И на самом деле, именно она является основой для набора паттернов, использованных в этой книге.

Книгу «Банды Четырех» вряд ли можно назвать последним словом в области паттернов проектирования, поскольку эта сфера существенно изменилась с момента ее первого издания. Но она, безусловно, является основополагающей.

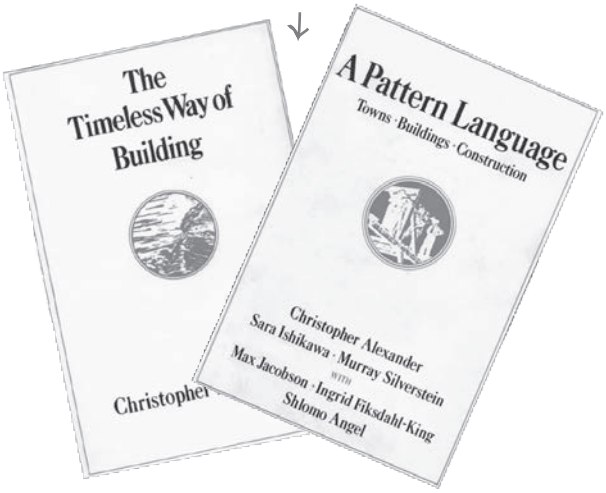
Авторы книги «Приемы объектно-ориентированного проектирования (Паттерны проектирования)» также известны как «Банда Четырех».

Базовые книги о паттернах

Паттерны начались не с «Банды Четырех», начало им положил Кристофер Александер, профессор архитектуры университета Беркли. Совершенно верно, Александр был *архитектором*, а не программистом, и использовал паттерны для проектирования жилых пространств (домов, городов и мегаполисов).

Если вы хотите более углубленно изучить паттерны, обратите внимание на две его книги: *The Timeless Way of Building* и *A Pattern Language*. Здесь вы узнаете об истоках паттернов и найдете много общего между строительством жилых зданий и разработкой программного обеспечения. Так что налейте чашечку кофе Starbuzz, устройтесь поудобнее и получайте удовольствие...

Кристофер Александр изобрел паттерны, которые можно использовать и для разработки программного обеспечения.



Другие ресурсы, посвященные паттернам

Вы готовы присоединиться к дружественному сообществу разработчиков паттернов проектирования? Вот несколько ресурсов, с которых вам стоит начать.

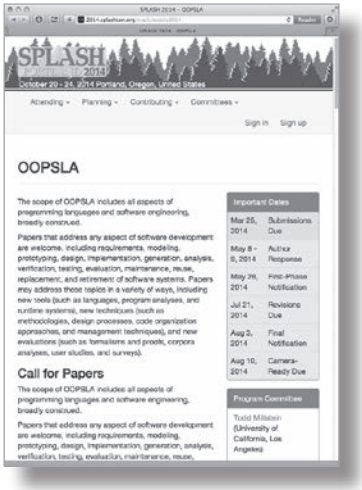
Интернет-сайты

The Portland Patterns Repository, проект, который ведет Уорд Каннингем, представляет собой WIKI-портал, касающийся всех аспектов разработки и использования паттернов. Каждый может стать автором сайта и принять участие в дискуссиях о паттернах и объектно-ориентированном программировании.

`http://c2.com/cgi/wiki?WelcomeVisitors`

The Hillside Group специализируется на практике программирования и проектирования ПО и является одним из основных ресурсов, посвященных использованию паттернов. На этом сайте вы найдете массу статей, книг и средств для работы с паттернами.

`http://hillside.net`



Конференции и семинары

А если вы предпочитаете живое общение с людьми, посетите любую из множества конференций или семинаров, посвященных паттернам. На сайте Hillside Group вы найдете полный список мероприятий. Кроме того, вы можете принять участие в OOPSLA, ежегодной американской конференции, посвященной вопросам объектно-ориентированного программирования.

Разновидности паттернов

Первые паттерны появились не в архитектуре программных продуктов, а в архитектуре зданий и городов. Более того, сама концепция паттернов может быть применена в самых разных областях. Давайте прогуляемся по зоопарку Объективиля и познакомимся с некоторыми из паттернов...

Архитектурные паттерны используются для создания живой, современной архитектуры зданий, городков и целых городов. Здесь зародилась сама концепция паттернов.

Область обитания: здания, в которых вам хотелось бы жить. Смотрите и заходите.

Область обитания: трехуровневые архитектуры, системы «клиент-сервер», Веб.

Прикладные паттерны используются при создании архитектур системного уровня. Многие многоуровневые архитектуры относятся к этой категории.

Примечание: MVC иногда относят к категории прикладных паттернов.

Узкоспециализированные паттерны предназначены для решения задач в конкретных областях (параллельное программирование, системы реального времени и т. д.).

Помогите найти область обитания

J2EE

Паттерны бизнес-процессов описывают взаимодействия между предприятиями, клиентами и данными; могут применяться для решения таких проблем, как эффективное принятие и передача решений.

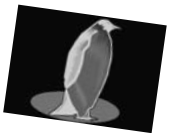
Встречается в залах заседаний крупных корпораций и на собраниях, посвященных управлению проектами.

Помогите найти область обитания

Группа разработки

Группа технической поддержки

Организационные паттерны описывают структуру и порядок деятельности организаций. Большинство современных разработок относится к организациям, занимающимся созданием и (или) поддержкой программных продуктов.



Паттерны пользовательского интерфейса решают проблемы проектирования интерактивных программ.

Область обитания: проектирование видеоигр, графические интерфейсы.

Запишите здесь свои наблюдения по поводу разновидностей паттернов.

Антипаттерны и борьба со злом

Вселенная попросту была бы неполной, если бы существовали одни паттерны без антипаттернов.

Если паттерн проектирования предоставляет общее решение распространенной задачи в конкретном контексте, то что же делает антипаттерн?

> **Антипаттерн** описывает ПЛОХОЕ решение задачи.

Резонно спросить: зачем тратить время на документирование плохих решений?

Если у типичной задачи имеется распространенное плохое решение, то его документирование поможет другим разработчикам избежать повторения ошибки. И если на то пошло, предотвращение плохих решений может принести не меньше пользы, чем поиск хороших!

Рассмотрим основные свойства антипаттерна:

Антипаттерн показывает, почему плохое решение выглядит привлекательно. Согласитесь, никто не станет выбирать плохое решение, если бы оно изначально не казалось привлекательным. Одна из самых важных задач антипаттернов — обратить ваше внимание на соблазны антипаттернов.

Антипаттерн объясняет, почему данное решение плохо в долгосрочной перспективе. Чтобы понять, почему решение относится к антипаттернам, необходимо видеть все отрицательные последствия его применения.

Антипаттерн предлагает паттерны, которые могут привести к хорошему решению. Чтобы антипаттерн был действительно полезным, он должен направить вас в верном направлении, то есть предложить другие возможности, которые ведут к хорошим решениям.

Рассмотрим пример антипаттерна.

Антипаттерн всегда выглядит, как хорошее решение, но после применения оказывается плохим.

Документирование антипаттернов поможет другим разработчикам распознавать плохие решения до того, как они будут реализованы.

У антипаттернов, как и у паттернов, существует много разновидностей: антипаттерны разработки, ОО-проектирования, организационные, узкоспециализированные и т. д.

Пример антипаттерна разработки.

Антипаттерн

Антипаттерн тоже обладает именем — а следовательно, закладывает основу для единой номенклатуры.

Название: Золотой молоток

Задача и контекст, как в описании паттерна.

Проблема: Требуется выбрать технологию для разработки. Вы полагаете, что вся архитектура должна быть построена на фундаменте ровно одной технологии.

Контекст: Технология, с которой хорошо знакома группа, плохо соответствует требованиям разрабатываемой системы или программного продукта.

Факторы, способствующие выбору данного решения.

Факторы:

- Группа разработки стремится использовать знакомую технологию.
- Группа разработки не знакома с другими технологиями.
- Переход на незнакомую технологию сопряжен с определенным риском.
- Знакомая технология упрощает планирование и оценку разработки.

Плохое, но внешне привлекательное решение.

Предлагаемое решение: Использовать знакомую технологию, несмотря ни на что. Технология упорно применяется для решения любых задач — даже там, где это совершенно очевидно неуместно.

Пути перехода к хорошему решению.

Рекомендуемое решение: Повышение квалификации разработчиков посредством обучения и участия в семинарах, которые знакомят разработчиков с новыми решениями.

Ситуации, в которых был замечен антипаттерн.

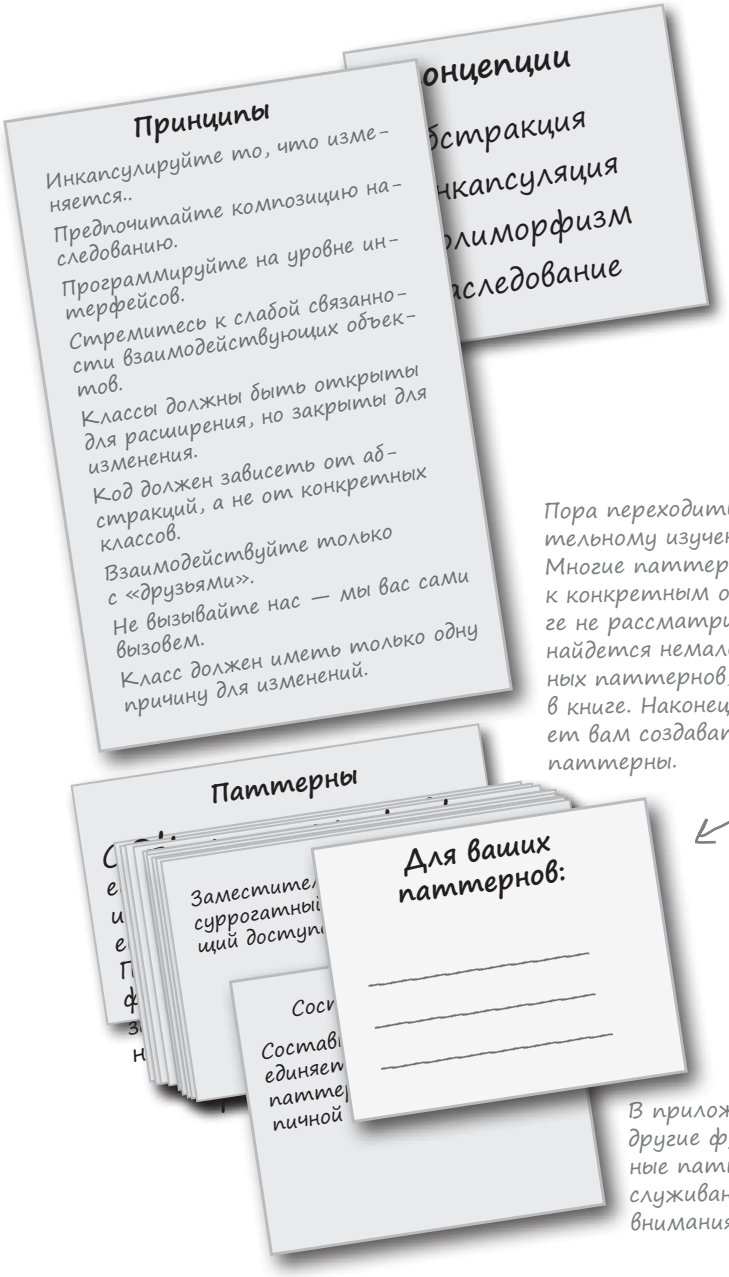
Примеры:

Некоторые веб-компании продолжают использовать и поддерживать самостоятельно разработанные системы кэширования, несмотря на наличие альтернативных решений с открытым кодом.

По материалам Portland Pattern Repository WIKI (http://c2.com/).

Новые инструменты

Сейчас вы знаете столько же, сколько знаем мы. Пора выбираться в реальный мир и заняться самостоятельным изучением паттернов...



Пора переходить к самостоятельному изучению паттернов. Многие паттерны, относящиеся к конкретным областям, в книге не рассматривались. Также найдется немало фундаментальных паттернов, не упоминаемых в книге. Наконец, ничто не мешает вам создавать собственные паттерны.

В приложении описаны другие фундаментальные паттерны, заслуживающие вашего внимания.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Решение об использовании паттернов должно естественно следовать из архитектуры. Не используйте паттерны только ради паттернов.
- Паттерны проектирования — не догма; адаптируйте и подстраивайте их для своих потребностей.
- Всегда выбирайте самое простое решение, соответствующее вашим потребностям, даже если в нем не используются паттерны.
- Изучайте каталоги паттернов, чтобы поближе познакомиться с паттернами и отношениями между ними.
- Классификация паттернов обеспечивает их логическую группировку.
- Для создания новых паттернов необходимо время и терпение. Приготовьтесь вносить многократные изменения в свою работу.
- Большинство «новых» паттернов, с которыми вы столкнетесь, будет представлять собой модификации существующих паттернов.
- Формируйте общую терминологию вашей рабочей группы. Это одно из важнейших преимуществ паттернов.
- В сообществе паттернов, как и в любом другом сообществе, существует свой жаргон. Пусть он вас не пугает — после прочтения этой книги вы знаете большую часть терминов.

Покидая Объективиль...

Как хорошо, что вы к нам заехали...

Конечно, мы будем по вам скучать. Но не огорчайтесь — вы и оглянуться не успеете, как в серии Head First выйдет новая книга и мы встретимся снова. О чем будет следующая книга? Хмм, хороший вопрос! Может, хотите поделиться своим мнением? Отправляйте пожелания по адресу booksuggestions@wickedlysmart.com

Кто и что делает?

Решение

Соедините каждый паттерн с его описанием:

Паттерн	Описание
Декоратор	Упаковывает объект и предоставляет другой интерфейс к нему.
Состояние	Субклассы решают, как реализовать шаги в алгоритме.
Итератор	Субклассы решают, какие конкретные классы должны создаваться.
Фасад	Обеспечивает создание одного (и только одного!) экземпляра класса.
Стратегия	Инкапсулирует взаимозаменяемые варианты поведения и выбирает один из них посредством делегирования.
Заместитель	Клиент выполняет однородные операции с объектами и коллекциями.
Фабричный Метод	Инкапсулирует поведение, связанное с состоянием, с делегированием поведения объекту текущего состояния.
Адаптер	Обеспечивает механизм перебора коллекций объектов без раскрытия реализации.
Наблюдатель	Упрощает интерфейс группы классов.
Шаблонный Метод	Упаковывает объект для реализации нового поведения.
Компоновщик	Позволяет клиенту создавать семейства объектов без указания их конкретных классов.
Одиночка	Обеспечивает оповещение объектов об изменении состояния.
Абстрактная Фабрика	Упаковывает объект для управления доступом к нему.
Команда	Инкапсулирует запрос в виде объекта.

14 Приложение

Другие паттерны

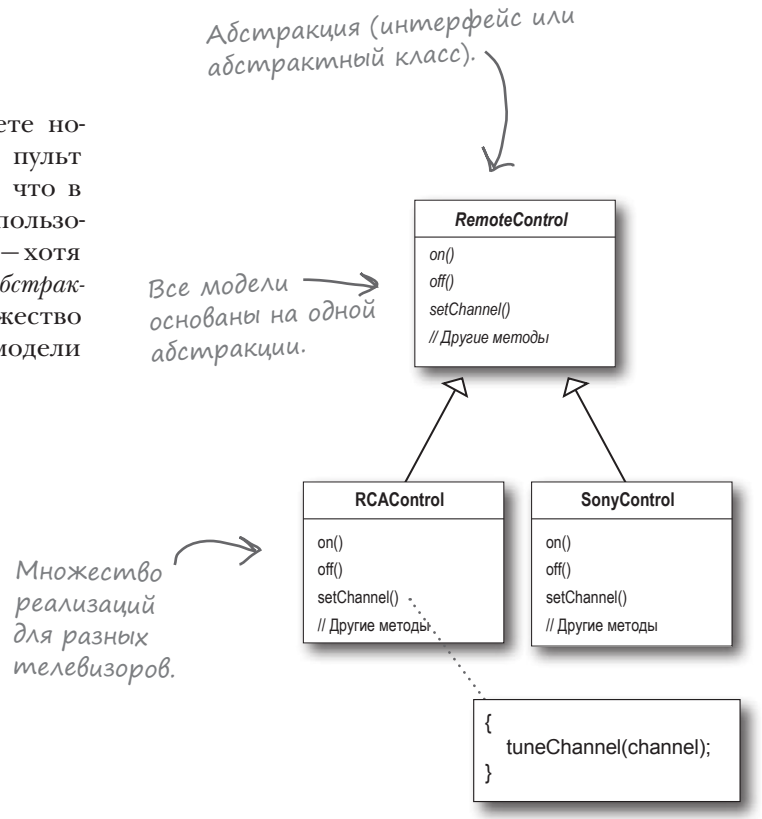
Не каждому суждено оставаться на пике популярности. За последние 10 лет многое изменилось. С момента выхода первого издания книги «Банды Четырех» разработчики тысячи раз применяли эти паттерны в своих проектах. В этом приложении представлены полноценные, первосортные паттерны от «Банды Четырех» — они используются реже других паттернов, которые рассматривались ранее. Однако эти паттерны ничем не плохи, и если они уместны в вашей ситуации — применяйте их без малейших сомнений. В этом приложении мы постараемся дать общее представление о сути этих паттернов.

Мост

Используйте паттерн Мост, если изменяться может не только реализация, но и абстракции.

Сценарий

Представьте, что вы программируете новый эргономичный универсальный пульт ДУ для телевизора. Вы уже знаете, что в программировании необходимо использовать принципы ОО-проектирования – хотя архитектура базируется на одной *абстракции*, в ней будет задействовано множество *реализаций* – по одному для каждой модели телевизора.



Дилемма

Вы знаете, что интерфейс пульта не будет определен в окончательном виде с первого раза. Более того, предполагается, что продукт будет совершенствоваться с получением данных обратной связи от пользователей.

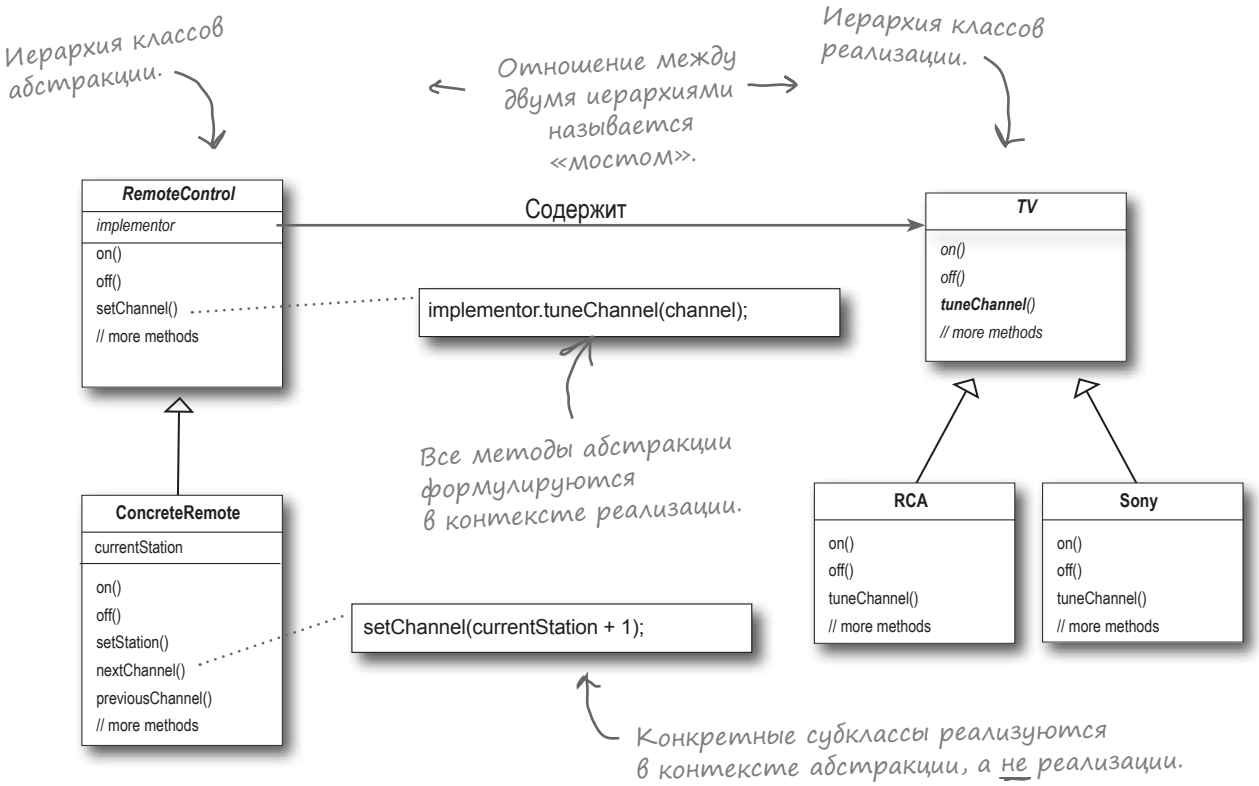
Таким образом, меняться будут как пульты, так и телевизоры. Вы уже *абстрагировали* пользовательский интерфейс, чтобы иметь возможность изменения *реализации* для многочисленных моделей телевизоров, принадлежащих пользователям. Но, как ожидается, абстракция *тоже* будет изменяться со временем на основании полученных отзывов пользователей.

Как же создать ОО-архитектуру, которая позволяла бы изменять как реализацию, *так и* абстракцию?

В этой архитектуре изменяться может только реализация, но не интерфейс.

Как использовать паттерн Мост

Паттерн Мост позволяет изменять реализацию *и* абстракцию, для чего они размещаются в двух разных иерархиях классов.



Архитектура состоит из двух иерархий: для пультов и для конкретных телевизоров (реализаций). Паттерн Мост позволяет изменять каждую из двух иерархий независимо от другой иерархии.

Преимущества Моста

- Логическое отделение реализации от интерфейса.
- Абстракция и реализация могут расширяться независимо друг от друга.
- Изменения в конкретных классах абстракции не отражаются на стороне клиента.

Область применения и недостатки

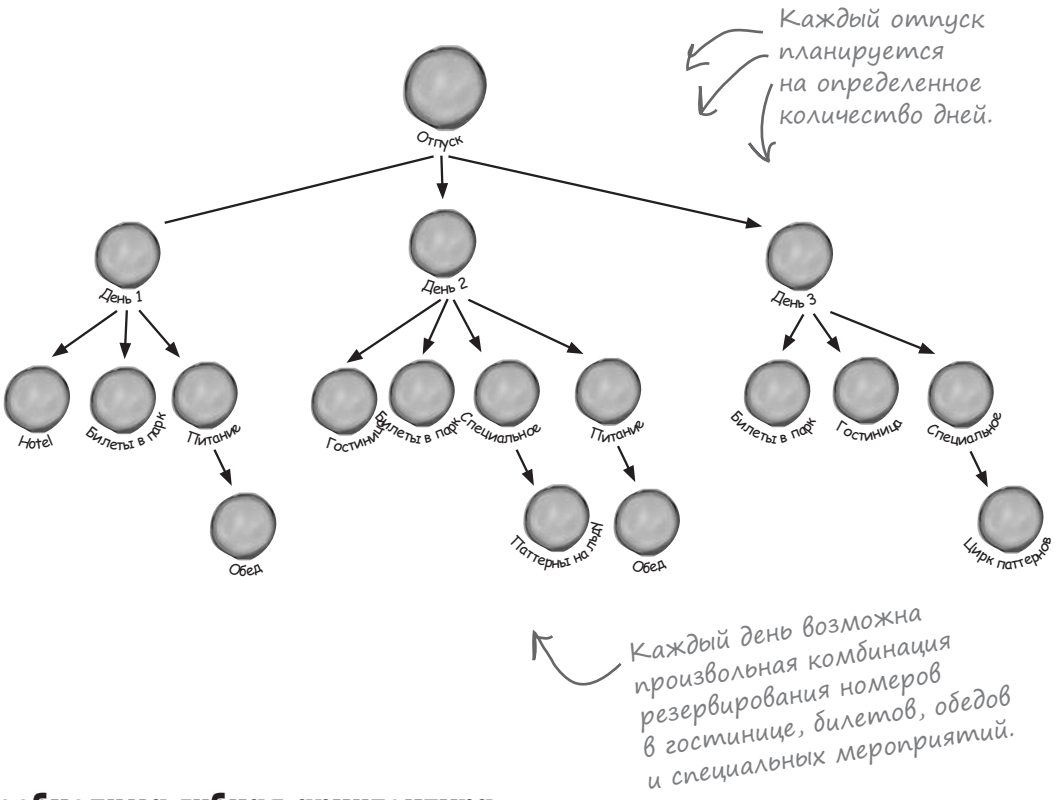
- Используется в графических и оконных системах, которые должны работать на разных платформах.
- Полезен в ситуациях с возможностью независимого изменения интерфейса и реализации.
- Повышает сложность.

Строитель

Паттерн Строитель инкапсулирует конструирование продукта и позволяет разделить его на этапы.

Сценарий

Вам предложено создать систему планирования экскурсий по Паттернленду — новому парку развлечений неподалеку от Объективиля. Гости парка могут выбрать гостиницу, заказать билеты на аттракционы, зарезервировать места в ресторане и даже заказать специальные мероприятия. Запланированная экскурсия выглядит примерно так:



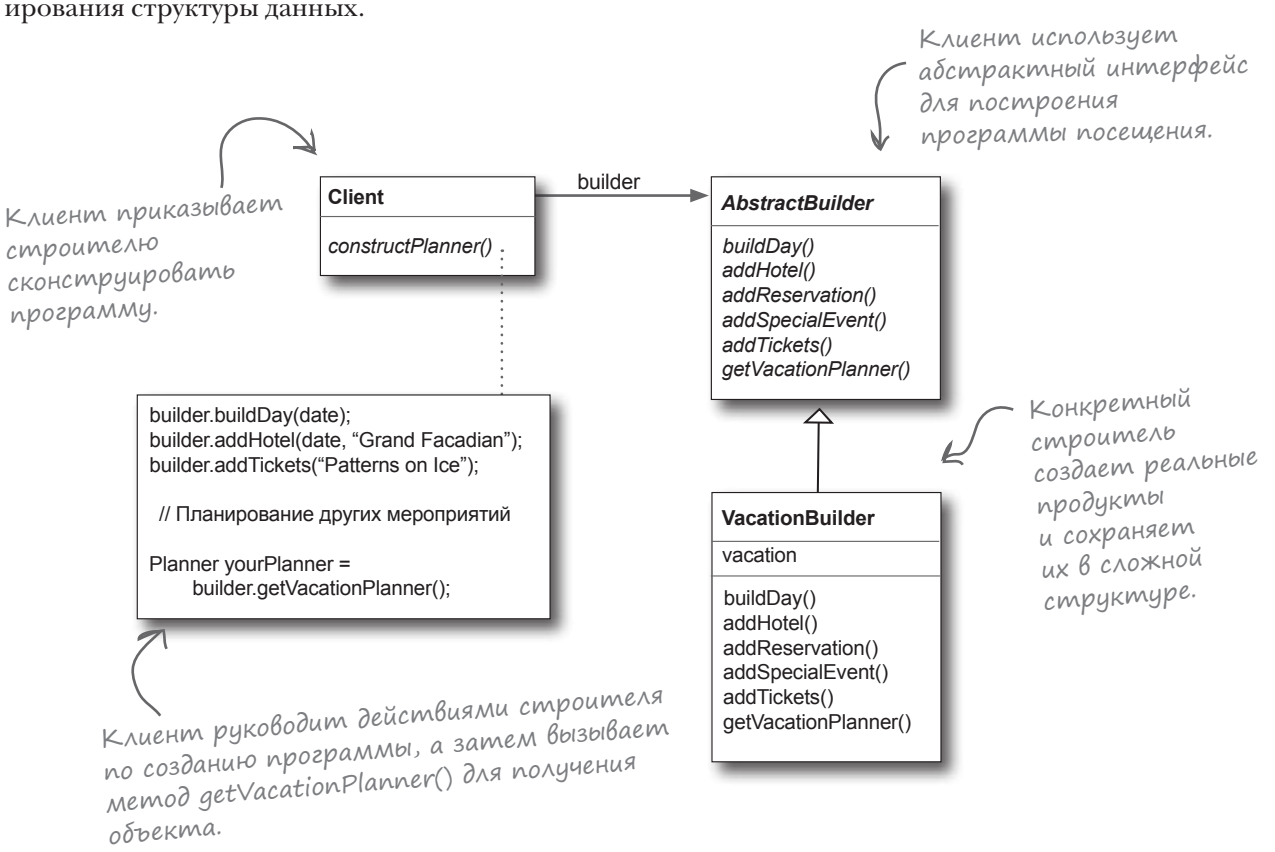
Необходима гибкая архитектура

Поездки могут различаться по количеству дней и составу развлекательной программы. Например, местный житель не нуждается в гостинице, но хочет заказать обед и специальные мероприятия. Другой гость прилетает в Объективиль самолетом, и ему необходимо забронировать номер в гостинице, столик в ресторане и билеты на мероприятия.

Таким образом, нам нужна гибкая структура данных, которая может представлять программу посещения со всеми возможными вариациями; кроме того, построение программы может состоять из нескольких (возможно, нетривиальных) шагов. Как предоставить способ построения сложной структуры данных, не смешивая его с отдельными этапами ее создания?

Как использовать паттерн Строитель

Помните паттерн Итератор? Мы инкапсулировали перебор в отдельном объекте и скрыли внутреннее представление коллекции от клиента. Здесь используется та же идея: создание программы посещения инкапсулируется в объекте (назовем его «строителем»), а клиент использует этот объект для конструирования структуры данных.



Преимущества Строителя

- Инкапсуляция процесса создания сложного объекта.
- Возможность поэтапного конструирования объекта с переменным набором этапов (в отличие от «одноэтапных» фабрик).
- Сокрытие внутреннего представления продукта от клиента.
- Реализации продуктов могут свободно изменяться, потому что клиент имеет дело только с абстрактным интерфейсом.

Область применения и недостатки

- Часто используется для создания составных структур.
- Конструирование объектов с использованием паттерна Фабрика требует от клиента дополнительных знаний предметной области.

Цепочка Обязанностей

Паттерн Цепочка Обязанностей используется, когда вы хотите предоставить нескольким объектам возможность обработать запрос.

Сценарий

Фирма Mighty Gumball не может справиться с сообщениями, поступающими после выпуска торгового автомата с поддержкой Java. Анализ показывает, что сообщения делятся на четыре вида: выражения восхищения от поклонников, которым нравится новая игра «1 из 10»; жалобы от родителей, дети которых не на шутку увлеклись игрой, и просьбы об установке автоматов в новых местах. К четвертому виду, естественно, относится спам.

Все сообщения поклонников должны поступать руководству, все жалобы — в юридический отдел, а запросы на установку — в отдел коммерческого развития. Спам просто удаляется.

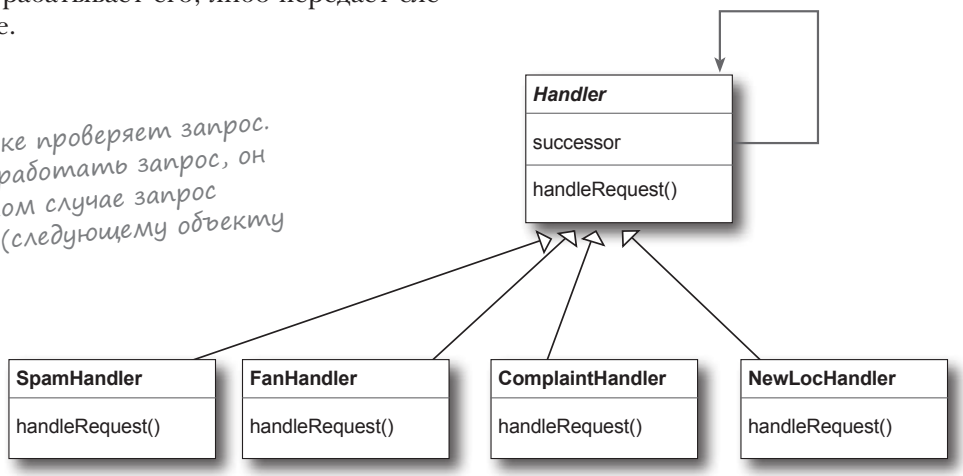
Ваша задача

Фирма Mighty Gumball уже разработала эвристические детекторы, которые определяют, к какому виду относится сообщение. Теперь вы должны создать архитектуру, использующую эти детекторы для обработки входящей почты.

Как использовать паттерн Цепочка Обязанностей

В паттерне Цепочка Обязанностей создается цепочка объектов, последовательно анализирующих запрос. Каждый объект получает запрос и либо обрабатывает его, либо передает следующему объекту в цепочке.

Каждый объект в цепочке проверяет запрос. Если объект может обработать запрос, он это делает; в противном случае запрос передается преемнику (следующему объекту в цепочке).



Полученное сообщение передается первому обработчику, SpamHandler. Если SpamHandler не может обработать запрос, то последний передается FanHandler. И так далее...

Сообщение, добравшееся до конца цепочки, остается необработанным — хотя вы всегда можете реализовать обработчик по умолчанию.

Каждое сообщение передается первому обработчику.

Преимущества Цепочки Обязанностей

- Логическая изоляция отправителя запроса от получателей.
- Объект упрощается, поскольку ему не нужно ни знать структуру цепочки, ни хранить прямые ссылки на ее элементы.
- Возможность динамического добавления или удаления обязанностей посредством изменения элементов цепочки или их порядка.

Область применения и недостатки

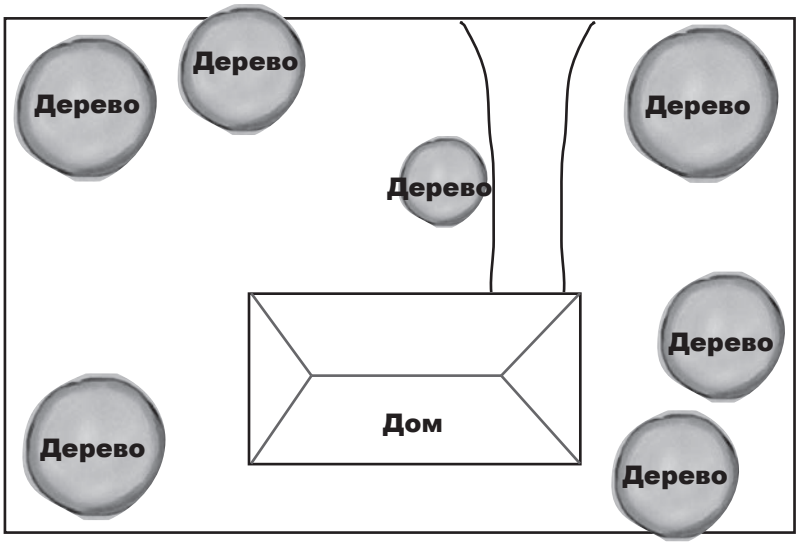
- Часто используется в графических средах для обработки событий (щелчки мышью, события клавиатуры и т. д.).
- Обработка запроса не гарантирована; если событие не будет обработано ни одним объектом, оно просто выходит с конца цепочки (это может быть как достоинством, так и недостатком).
- Цепочки могут усложнять отслеживание запросов и отладку.

Приспособленец

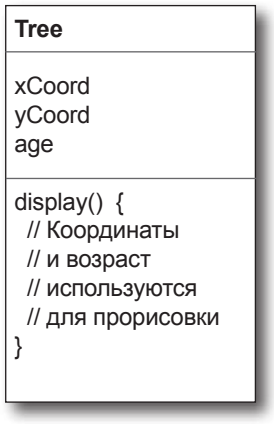
Используйте паттерн Приспособленец, если один экземпляр класса может предоставлять много «виртуальных экземпляров».

Сценарий

Вы хотите включить деревья в свое новое приложение ландшафтного дизайна. В вашем приложении деревья обладают минимальной функциональностью; они имеют координаты X-Y и могут динамически прорисовывать себя в зависимости от возраста. Проблема в том, что рано или поздно клиент захочет включить в свой дизайн множество деревьев. Это будет выглядеть примерно так:



Каждый экземпляр Tree поддерживает свое состояние.

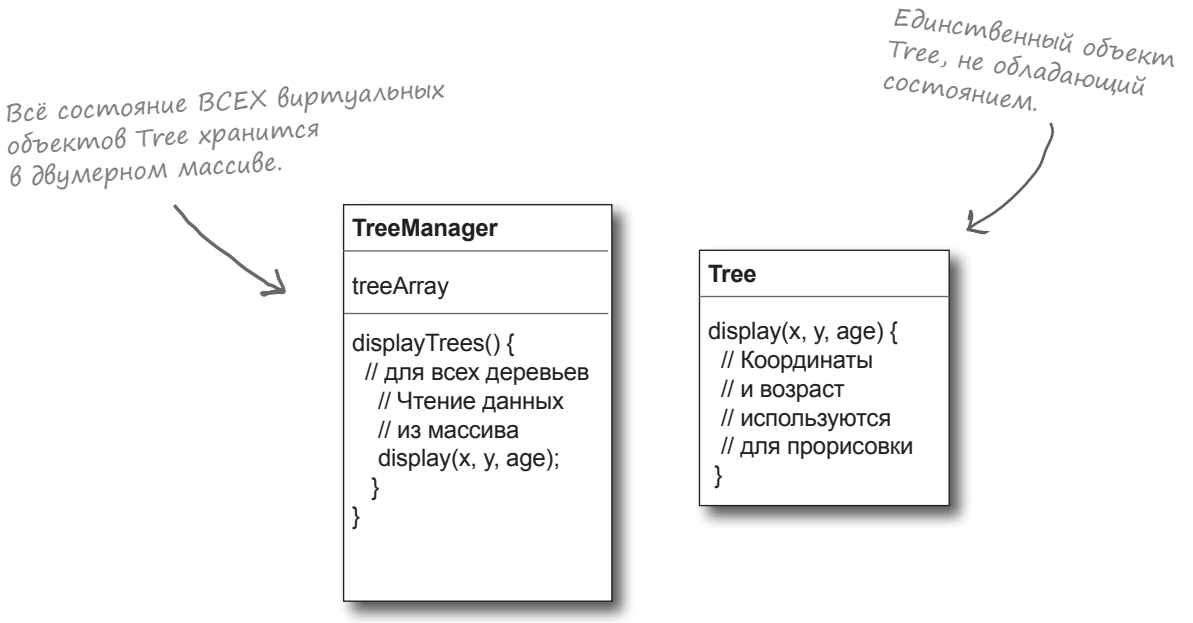


Дилемма

Важный клиент, которого вы обхаживали в течение многих месяцев, собирается купить 1000 комплектов вашего приложения для планирования ландшафтов в крупных сообществах. Поработав с приложением в течение недели, клиент жалуется, что при создании большого количества деревьев приложение начинает «тормозить»...

Как использовать паттерн Приспособленец

А если вместо создания тысяч объектов Tree переработать систему, чтобы в ней был только один экземпляр Tree и клиентский объект, хранящий состояние ВСЕХ деревьев? Это и есть паттерн Приспособленец!



Преимущества Приспособленца

- Сокращение количества экземпляров класса во время выполнения (экономия памяти).
- Централизованное хранение состояния многих «виртуальных» объектов.

Область применения и недостатки

- Приспособленец используется в том случае, когда класс имеет много экземпляров, которыми можно управлять одинаково.
- Недостаток паттерна Приспособленец заключается в том, что после его реализации отдельные экземпляры класса уже не смогут обладать собственным поведением, независимым от других экземпляров.

Интерпретатор

Паттерн Интерпретатор используется для создания языковых интерпретаторов.

РАССЛАБЬТЕСЬ

Паттерн Интерпретатор требует некоторого знания теории формальной грамматики. Впрочем, если вы никогда не изучали формальную грамматику, все равно прочитайте описание паттерна; суть будет понятна.

Сценарий

Помните имитатор утиного пруда? На его основе можно создать программу, которая будет обучать детишек программированию. Ребенок получает в свое распоряжение одну утку и управляет ею, отдавая команды на простом языке. Пример:

```
right;
while (daylight) fly;
quack;
```

Повернуть направо.

Лететь весь день...

...потом крякнуть.

Припомнив теорию создания грамматик, которую вам излагали на одной из вводных лекций по программированию, вы записываете определение грамматики:

Программа представляет собой выражение, состояние из серии команд и повторений («while»).

Серия представляет собой последовательность выражений, разделенных символом «;».

Три команды: right, quack и fly.

Конструкция while состоит из управляющей переменной и выражения.

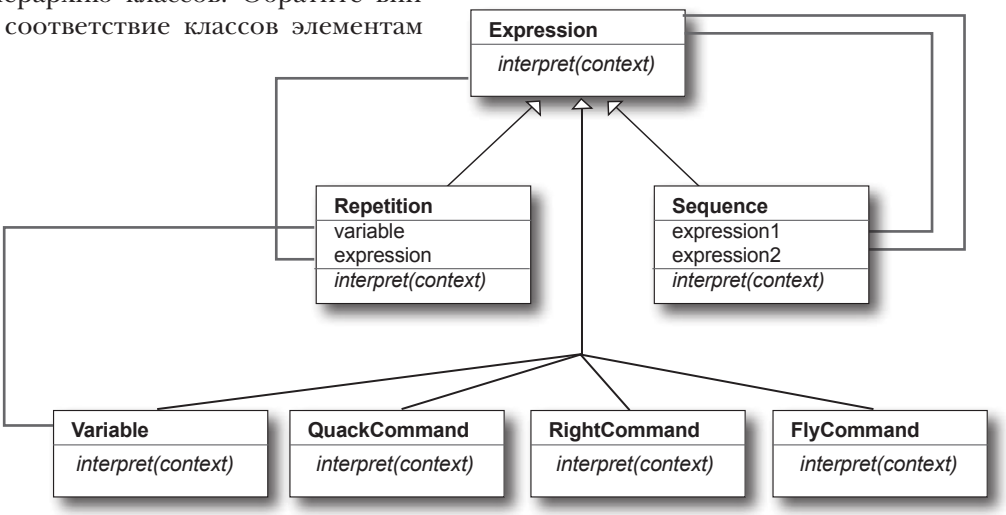
```
expression ::=  <command> | <sequence> | <repetition>
sequence ::= <expression> ';' <expression>
command ::= right | quack | fly
repetition ::= while '(' <variable> ')'<expresion>
variable ::= [A-Z,a-z]+
```

Что дальше?

Грамматика готова; теперь необходим механизм представления и интерпретации серий, чтобы ученики видели, как введенные ими команды действуют на виртуальных уток.

Как реализовать интерпретатор

В паттерне Интерпретатор определяется представление своей грамматики в виде класса вместе с интерпретатором для обработки серий. В архитектуру вводится класс, представляющий каждое правило языка. На диаграмме изображен «утиный» язык, преобразованный в иерархию классов. Обратите внимание на прямое соответствие классов элементам грамматики:



Интерпретация языка осуществляется вызовом метода interpret() для каждого типа выражения. Метод получает контекст (содержащий входной поток разбираемой программы), идентифицирует входные данные и обрабатывает их.

Преимущества Интерпретатора

- Представление правил грамматики в виде классов упрощает реализацию языка.
- Так как грамматика представлена классами, вы можете легко изменять и расширять язык.
- Включение дополнительных методов в структуру классов позволяет добавлять новое поведение, не связанное с интерпретацией. Скажем, форматированный вывод или проверку корректности интерпретируемого кода.

Область применения и недостатки

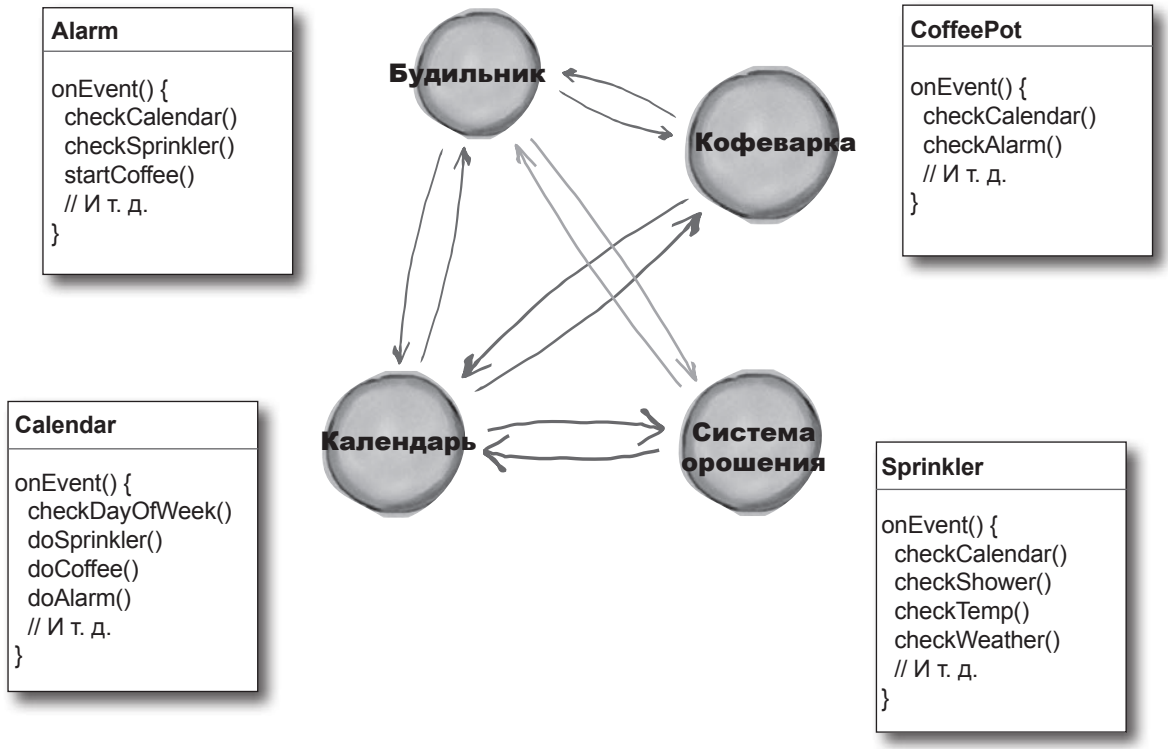
- Используйте интерпретатор для реализации простого языка.
- Паттерн уместен для языков с простой грамматикой, когда простота важнее эффективности.
- Может использоваться как со сценарными языками, так и с языками программирования.
- При большом количестве грамматических правил реализация становится громоздкой и неуклюжей. В таких случаях лучше воспользоваться парсером/компилятором.

Посредник

Паттерн Посредник используется для централизации сложных взаимодействий и управляющих операций между объектами.

Сценарий

Боб живет в «умном доме» из недалекого будущего: все домашние приборы и устройства сконструированы так, чтобы сделать его жизнь как можно более комфортной. Когда Боб отключает сигнал будильника, то будильник приказывает кофеварке приступить к приготовлению кофе. Но несмотря на все удобства, Боб и другие клиенты всегда хотят чего-то нового: не варить кофе по выходным... Отключить оросительную систему за 15 минут до принятия душа... Ставить будильник на более раннее время в дни вывоза мусора...



Дилемма

Создателям дома становится все труднее следить за тем, в каких объектах находятся те или иные правила и как разные объекты связаны друг с другом.

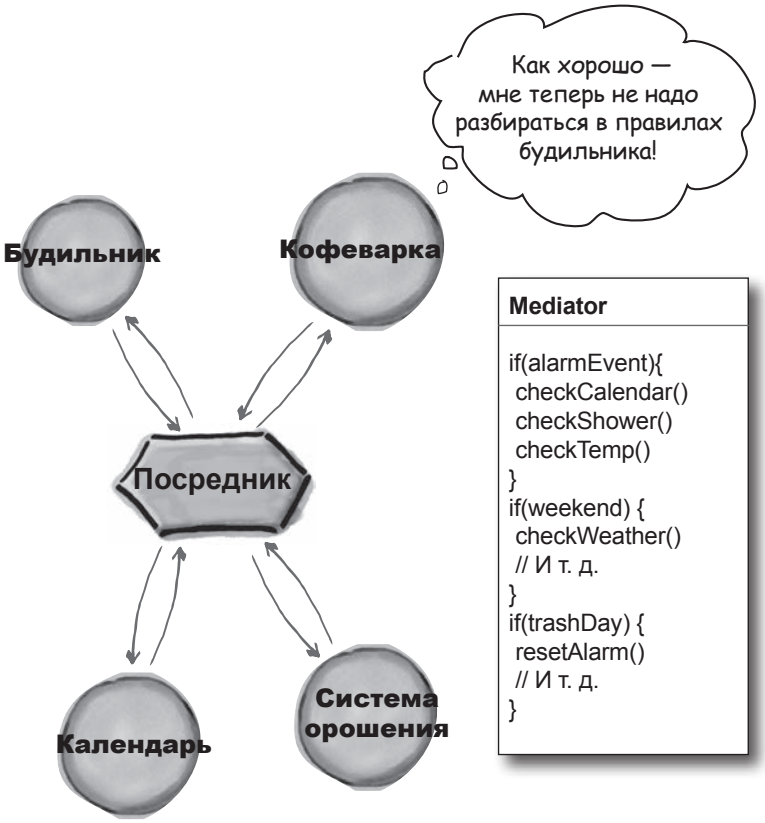
Посредник в действии...

Включение Посредника в систему значительно упрощает все объекты устройств:

- Объекты устройств оповещают Посредника об изменении своего состояния.
- Объекты устройств отвечают на запросы посредника.

Перед добавлением Посредника все объекты устройств должны были знать о существовании друг друга... Между ними существовала жесткая привязка. В архитектуре с Посредником объекты устройств *полностью отделены* друг от друга.

Посредник содержит всю управляющую логику системы. Когда существующему устройству потребуется добавить новое правило или в систему добавляется новое устройство, вся необходимая логика будет размещаться в Посреднике.



Преимущества Посредника

- Расширение возможностей повторного использования кода объектов, находящихся под управлением посредника.
- Упрощение сопровождения системы за счет централизации управляющей логики.
- Упрощение сообщений, передаваемых между объектами в системе.

Область применения и недостатки

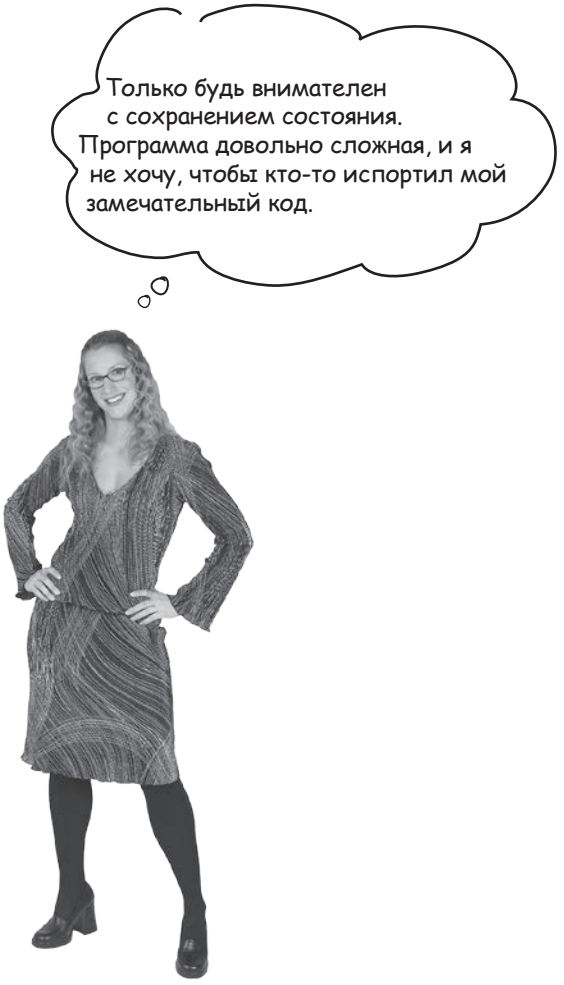
- Посредник часто используется для координации работы взаимосвязанных компонентов графического интерфейса.
- Недостаток паттерна Посредник состоит в том, что без должного проектирования объект Посредник может стать излишне сложным.

Хранитель

Паттерн Хранитель используется для реализации возврата к одному из предыдущих состояний (например, если пользователь выполнил команду «Отменить»).

Сценарий

Ваша ролевая игра пользуется огромным успехом, и у нее появился целый легион поклонников. Все они пытаются добраться до знаменитого «13-го уровня». По мере того, как пользователи переходят на более сложные уровни, вероятность внезапного завершения игры растет. Человек потратил несколько дней на проход к сложным уровням, а его персонаж вдруг пропадает, и все приходится начинать заново... Естественно, это никого не обрадует. Игроки хором требуют ввести команду сохранения, чтобы в случае гибели они могли восстановить хотя бы бóльшую часть своих игровых достижений. Команда сохранения должна возвращать воскресшего персонажа на последний успешно пройденный уровень.

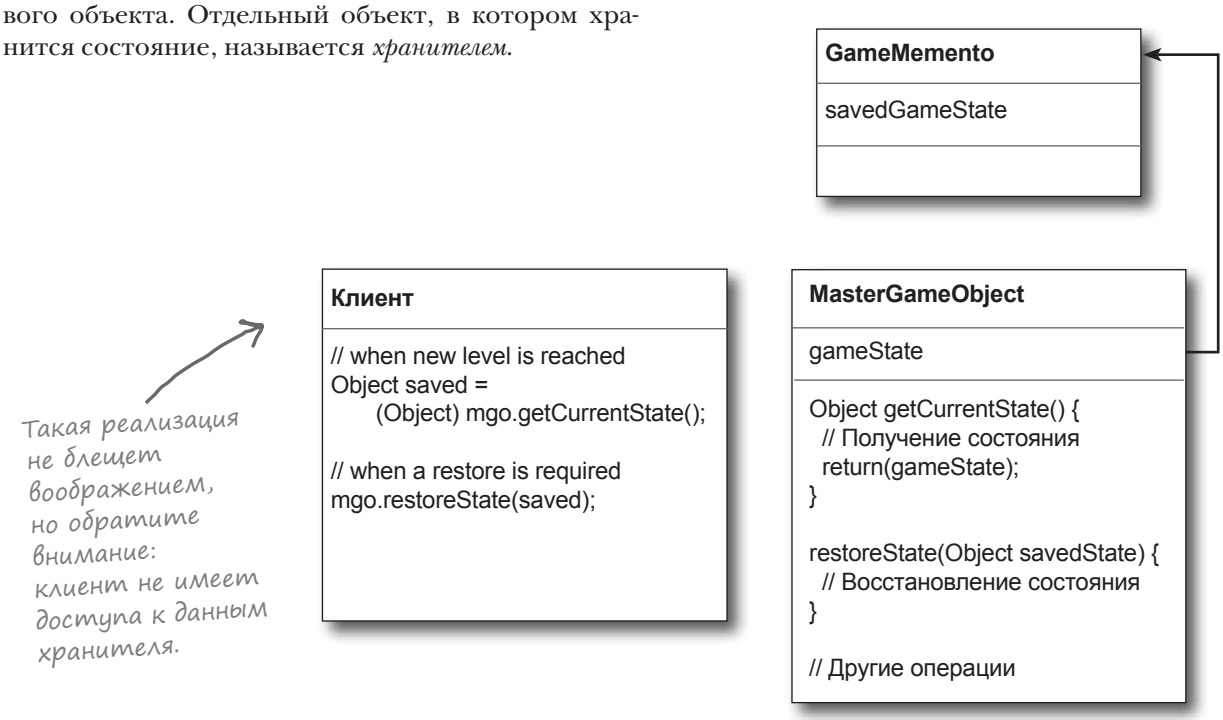


Хранитель за работой

Паттерн Хранитель имеет две цели:

- Сохранение важного состояния ключевого объекта системы.
- Должная инкапсуляция ключевого объекта.

В соответствии с принципом единой обязанности состояние желательно хранить отдельно от ключевого объекта. Отдельный объект, в котором хранится состояние, называется *хранителем*.



Преимущества Хранителя

- Хранение состояния отдельно от ключевого объекта улучшает связность системы.
- Инкапсуляция данных ключевого объекта.
- Простая реализация восстановления.

Область применения и недостатки

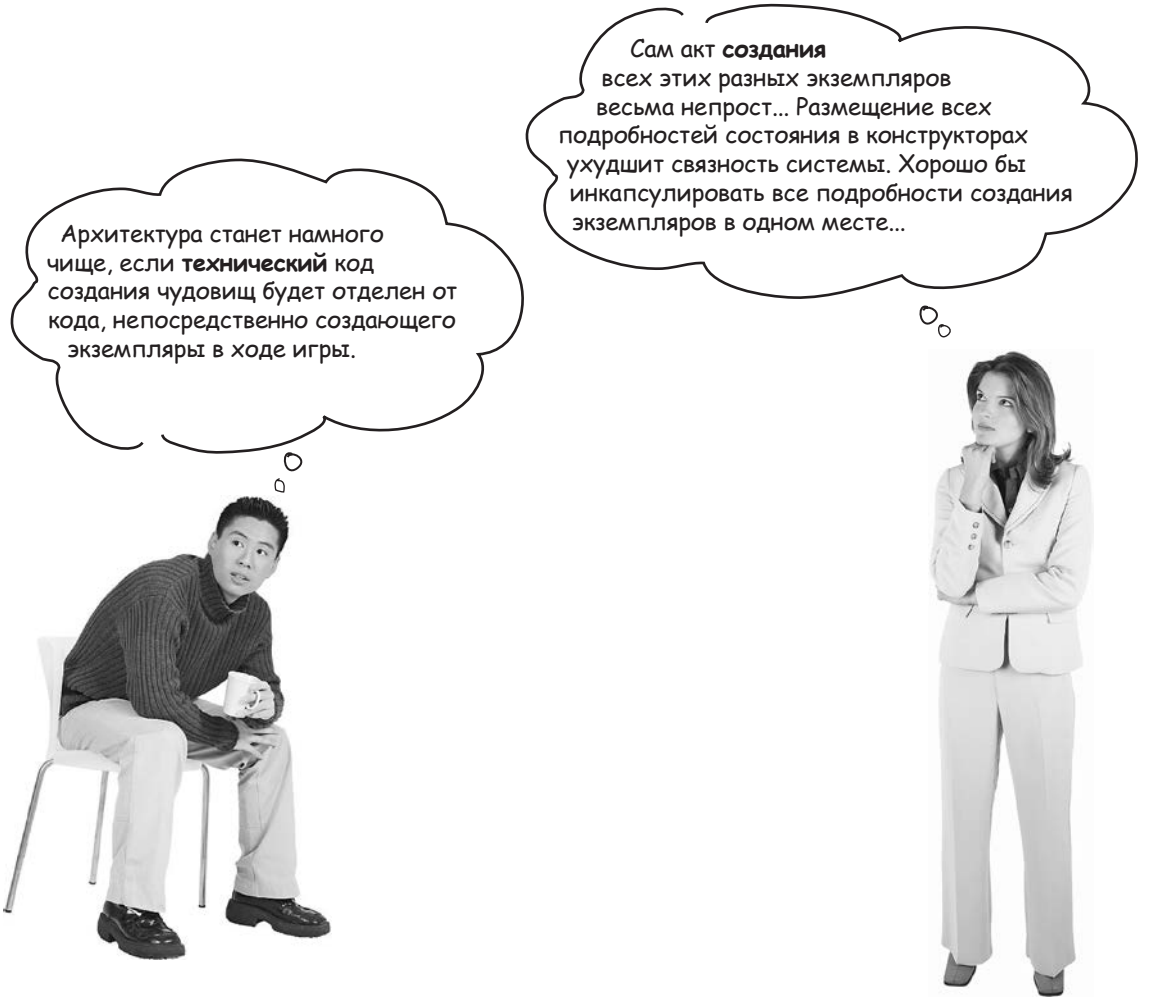
- Хранитель используется для сохранения состояния.
- Сохранение и восстановление состояния может быть довольно продолжительной операцией.
- В системах на базе Java для сохранения текущего состояния иногда можно воспользоваться сериализацией.

Прототип

Паттерн Прототип используется в тех случаях, когда создание экземпляра класса требует больших затрат ресурсов или занимает много времени.

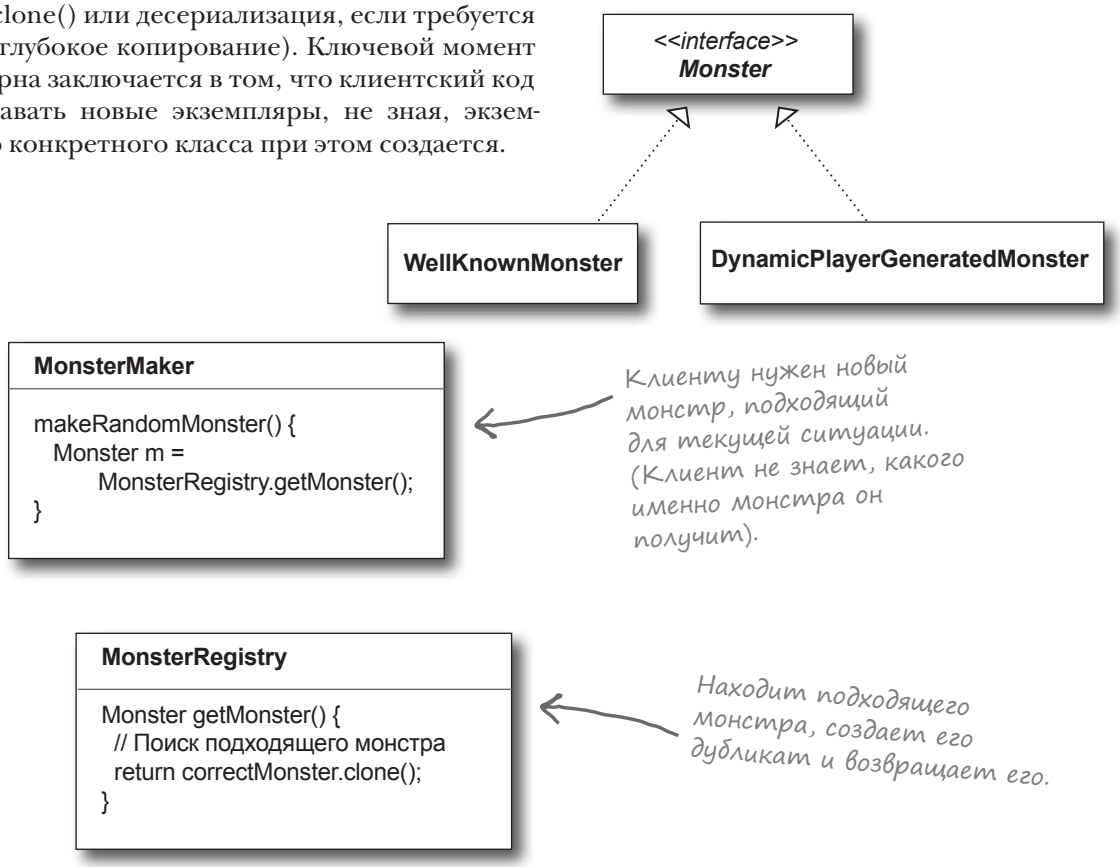
Сценарий

В вашей ролевой игре герои, странствующие по динамически генерируемому ландшафту, встречают бесконечную череду врагов. Характеристики монстров должны зависеть от изменяющегося ландшафта — согласитесь, крылатая тварь не очень уместна в подводном царстве. Наконец, вы хотите, чтобы опытные игроки могли создавать собственных монстров.



На помощь приходит Прототип

Паттерн Прототип позволяет создавать новые экземпляры посредством копирования существующих экземпляров. (В языке Java для этой цели обычно применяется метод clone() или десериализация, если требуется выполнить глубокое копирование). Ключевой момент этого паттерна заключается в том, что клиентский код может создавать новые экземпляры, не зная, экземпляр какого конкретного класса при этом создается.



Преимущества Прототипа

- Сложности создания новых экземпляров остаются скрытыми от клиента.
- У клиента появляется возможность генерировать объекты, тип которых ему неизвестен.
- В некоторых обстоятельствах копирование эффективнее создания нового объекта.

Область применения и недостатки

- Прототип обычно применяется в системах, создающих новые объекты разных типов из сложной иерархии классов.
- Создание копии объекта иногда бывает нетривиальной операцией.

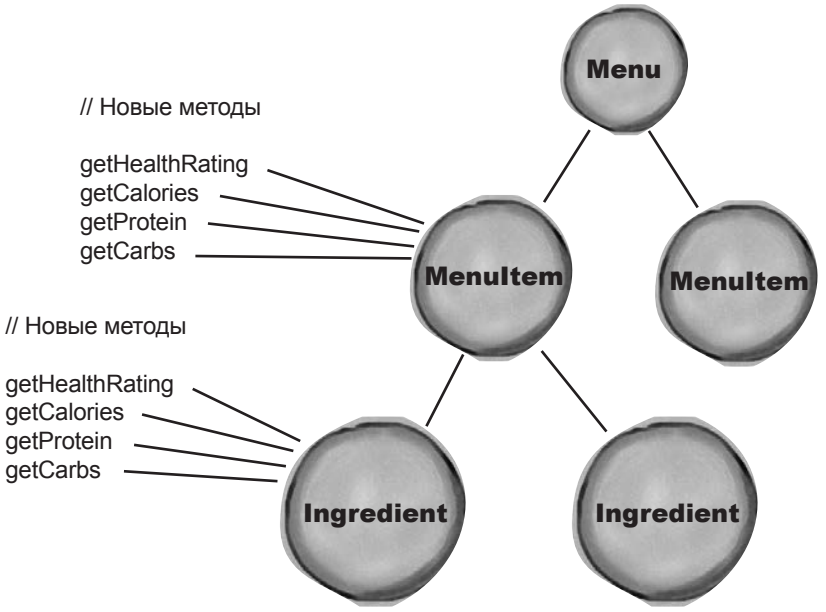
Посетитель

Паттерн Посетитель используется для расширения возможностей комбинации объектов в том случае, если инкапсуляция не существенна.

Сценарий

Посетители бистро и блинной из Объектвиля стали следить за своим здоровьем. Теперь они желают знать калорийность блюд перед оформлением заказа. Поскольку оба заведения предлагают блюда на заказ, некоторые посетители даже хотят получить информацию о калорийности отдельных ингредиентов.

Решение, предложенное Лу:

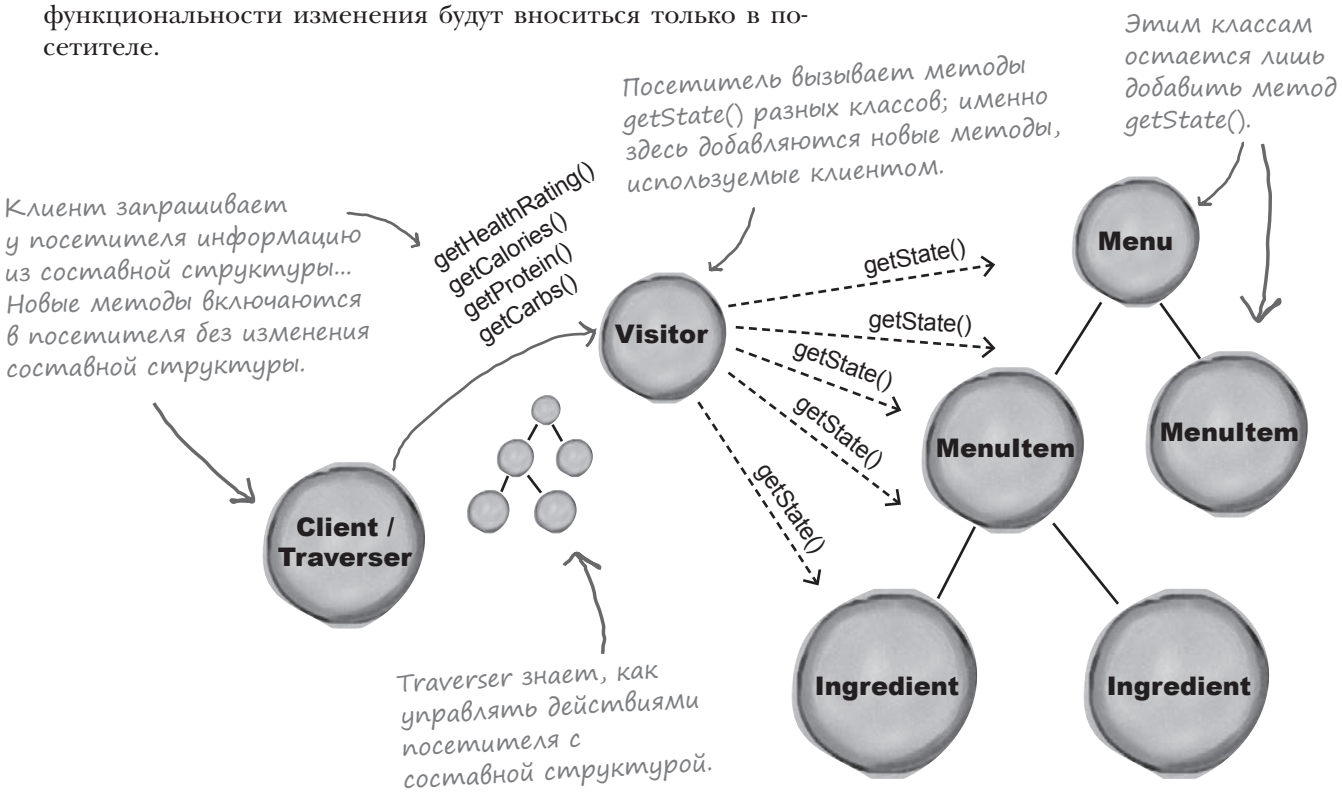


Мэл обеспокоен...

«Кажется, мы открываем ящик Пандоры. Кто знает, какие новые методы нам еще придется добавить — и каждый раз изменения будут вноситься в двух местах. А если мы захотим дополнить базовое приложение классом рецептов? Тогда изменения придется вносить уже в трех разных местах...»

Заходит Посетитель

Посетитель должен посетить каждый элемент составной структуры; эта функциональность сосредоточена в объекте Traverser. Последний управляет перемещениями Посетителя, чтобы тот мог получить данные состояния от всех объектов комбинации. После того как данные состояния будут получены, посетитель может по поручению клиента выполнить различные операции с состоянием. При добавлении новой функциональности изменения будут вноситься только в посетителе.



Преимущества Посетителя

- Возможность добавления операций в составную стурктуру без изменения самой структуры.
- Добавление новых операций выполняется относительно просто.
- Централизация кода операций, выполняемых посетителем.

Недостатки Посетителя

- Использование паттерна Посетитель нарушает инкапсуляцию составной структуры.
- Усложнение возможных изменений составной структуры.

Э. Фримен, Э. Робсон, К. Сьерра, Б. Бейтс

Head First. Паттерны проектирования. Обновленное юбилейное издание

Перевел с английского Е. Матвеев

Заведующая редакцией	*Ю. Сергиенко*
Художественный редактор	*С. Заматевская*
Корректор	*И. Тимофеева*
Верстка	*Н. Лукьянова*

Изготовлено в России. Изготовитель: ООО «Питер Пресс».
Место нахождения и фактический адрес: 192102, Россия, город Санкт-Петербург,
улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.

Дата изготовления: 01.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 —
Книги печатные профессиональные, технические и научные.

Подписано в печать 16.01.18. Формат 84×108/16. Бумага писчая. Усл. п. л. 68,880. Тираж 2000. Заказ 0000.

Отпечатано в соответствии с предоставленными материалами в ООО «ИПК Парето-Принт».
170546, Тверская область, Промышленная зона Боровлево-1, комплекс № 3А, www.pareto-print.

Э. Фримен, Э. Робсон

ИЗУЧАЕМ ПРОГРАММИРОВАНИЕ НА JAVASCRIPT

Вы готовы сделать шаг вперед в веб-программировании и перейти от верстки в HTML и CSS к созданию полноценных динамических страниц? Тогда пришло время познакомиться с самым «горячим» языком программирования — JavaScript! С помощью этой книги вы узнаете все о языке JavaScript — от переменных до циклов. Вы поймете, почему разные браузеры по-разному реагируют на код и как написать универсальный код, поддерживаемый всеми браузерами. Вам станет ясно, почему с кодом JavaScript никогда не придется беспокоиться о перегруженности страниц и ошибках передачи данных. Не пугайтесь, даже если ранее вы не написали ни одной строчки кода, — благодаря уникальному формату подачи материала эта книга с легкостью проведет вас по всему пути обучения: от написания простейшего скрипта до создания сложных веб-проектов, которые будут работать во всех современных браузерах. Особенностью этого издания является уникальный способ подачи материала, выделяющий серию «Head First» издательства O'Reilly в ряду множества скучных книг, посвященных программированию.

ПИТЕР®

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает
профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216;*
e-mail: books@piter.com

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217;*
e-mail: kuznetsov@piter.com

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Kiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**