Отзывы о книге

«В этой книге вы найдете новый, свежий взгляд на тему ООАП. Она отличается от других книг прежде всего своей учебной направленностью. На рынке слишком много изданий, которые подолгу объясняют, почему и как, но, по сути, не помогают программисту-практику начать работу над проектом. Эти книги очень интересны, но непрактичны. Я твердо уверен, что в будущем разработки программного обеспечения центральное место занимает практик. Авторы сделали содержимое ООАП доступным и полезным для практиков»

— Айвер Джейкобсон, Ivar Jacobson Consulting

«Я только что закончил читать книгу — и в полном восторге от нее! В книге излагаются основы объектно-ориентированного проектирования и анализа с использованием UML и сценариев использования и даже приводятся уроки качественного проектирования — и все это делается доступно и в быстром темпе. Больше всего мне понравилось то, что книга доступно объясняет, почему объектно-ориентированное проектирование способствует созданию качественных программных продуктов! Определяя, что такое качественная программа, и демонстрируя, как каждый шаг процесса ООАП ведет вас к этой цели, книга может объяснить даже самому неопытному Java-программисту значение ООАП. Это замечательная "первая книга" по проектированию для любого новичка, а также для программистов, которые уже довольно давно работают на Java, но не находят смелости взяться за толстые тома по ООАП».

— Кайл Браун, старший инженер, IBM

«Наконец-то я увидел книгу по ООАП, которая указывает, что UML — всего лишь способ записи, а самое главное при разработке программного продукта — тщательно обдумать все его аспекты».

— Пит Макбрин, автор, *Software Craftsmanship*

«В книге отлично передан увлекательный, визуально-ориентированный стиль серии *Head First.* Но за смешными картинками и безумными шрифтами скрывается серьезное, разумное, в высшей степени качественно сработанное введение в объектно-ориентированный анализ и проектирование. Авторы знают, как проектировать программные продукты, и эффективно передают свои знания читателю. Мне нравится, как последовательная проработка примеров ведет читателя по разным стадиям процесса проектирования. Во время чтения книги мне казалось, что я стою за спиной опытного проектировщика, который объясняет, какие проблемы особенно важны на каждом шаге и почему.»

— Эдвард Сциоре, адъюнкт-профессор, кафедра компьютерных технологий Бостонского колледжа

«Это хорошо написанная книга, которая учит своих читателей, как анализировать, проектировать и писать серьезные объектно-ориентированные программы. Ее содержимое без малейших усилий перетекает от применения сценариев использования для определения требований к анализу, проектированию, реализации, тестированию и итеративной разработке. Каждый шаг разработки объектно-ориентированного продукта представлен в свете проверенных принципов проектирования ПО. Примеры книги понятны и поучительны. Это основательная книга по объектно-ориентированной разработке, отличающаяся от других новым подходом к изложению материала"

— Дан Зун Нгуен, лектор, Университет Райса

О других книгах серии *Head First*

«HTML5 – это „будущее Интернета“. Наверняка вы уже слышали эту фразу. Если вы действительно хотите разобраться в семействе технологий, лежащих в основе HTML5, – данная книга именно то, что вам нужно! Она рассчитана на широкий круг читателей – от новичков до опытных разработчиков».

— Аарон Лаберже, исполнительный директор Fanzter Inc.

«Эта книга – захватывающее путешествие в неизведанный мир HTML5, который мы будем осваивать еще долгие годы. В ней раскрываются базовые концепции, благодаря которым вы сможете понять назначение HTML5-дизайна, а также исследуются все аспекты, необходимые для того, чтобы вы могли стать грамотным специалистом. Как и в других книгах серии *Head First*, в данном издании вместо сухого изложения материала читателю предлагаются яркие всплески информации, которая основана на фактических примерах и легко запоминается. Конечно, можно посетить официальный сайт со спецификациями HTML5, но я предпочитаю изучать этот язык посредством подобных ярких изданий».

— Кен Арнольд, дизайнер/разработчик в Peak Impact, Inc.

«Настольная книга о HTML5, продолжающая традицию серии *Head First*. В ней остроумное, веселое преподнесение материала потрясающе ловко сочетается с массой наглядных примеров!»

— Дэнни Мавроматис, старший разработчик ПО в ABC Television Group

«*„Изучаем программирование на HTML5“* замечательно преподносит читателю суть многих ключевых аспектов HTML5 в веселой, легкой для восприятия форме. Благодаря обилию наглядных примеров и многочисленным образцам кода сложные концепции (например, элемент canvas или асинхронное программирование) становятся простыми и понятными, что делает их в глазах читателя легкими для восприятия и притягательными».

— Майкл С. Шероттер, ведущий специалист по продвижению архитектур в Microsoft

«HTML5 – это торт с множеством технологических слоев. Книга *„Изучаем программирование на HTML5“* выпекает этот торт и бросает его вам в лицо. И вы ощутите, какой у него восхитительный вкус».

— Джош Роадс, сооснователь BrightHalf

«В книге *„Изучаем программирование на HTML5“* используется умеренно сложный подход к описанию разнообразных аспектов HTML5, благодаря чему тяжелый процесс изучения превращается в веселое занятие».

— Уорд Каннингэм, изобретатель технологии wiki

О других книгах серии *Head First*

«Будьте осторожны. Если кто-то из вас любит читать перед сном, то лучше отложите чтение книги „*Изучаем HTML, XHTML и CSS*“ (Head First HTML with CSS & XHTML) на дневное время. Эта книга будоражит мозг».

— Поулин Макнамара, Центр новых технологий и образования, Фрибурский университет, Швейцария

«Книга „*Изучаем HTML, XHTML и CSS*“ представляет собой тщательно проработанное современное руководство по дальновидным практикам в области разметки и представления веб-страниц. Авторы предвидят, какие моменты могут вызвать у читателя замешательство, и своевременно разъясняют их. Использованный подход, в основе которого лежат обилие наглядных примеров и последовательность изложения, является оптимальным для читателя: он будет вносить небольшие изменения и наблюдать итоговый эффект в браузере, что позволит разобраться в назначении каждого нового элемента».

— Дэнни Гудмен, автор книги «Динамический HTML: подробное руководство» (Dynamic HTML: The Definitive Guide)

«Книга „*Изучаем HTML, XHTML и CSS*“ с самого начала создает у читателя ощущение, что весь процесс обучения окажется простым и увлекательным. Освоение HTML при правильном объяснении не сложнее изучения основ родного языка, при этом авторы проделали отличную работу и приводят наглядные примеры по каждой концепции».

— Майк Дэвидсон, президент и исполнительный директор Newsvine, Inc

«Вместо изложения материала в стиле традиционных учебников „*Программируем для iPhone и iPad*“ предлагает читателю живую, увлекательную и даже приятную методику обучения программированию для iOS. Материал подобран умело и качественно: в книге рассматриваются многие ключевые технологии, включая Core Data, и даже такие важные аспекты, как проектирование интерфейса. И где еще можно прочитать, как UIWebView и UITextField беседуют у камина?»

— Шон Мерфи, проектировщик и разработчик приложений для iOS

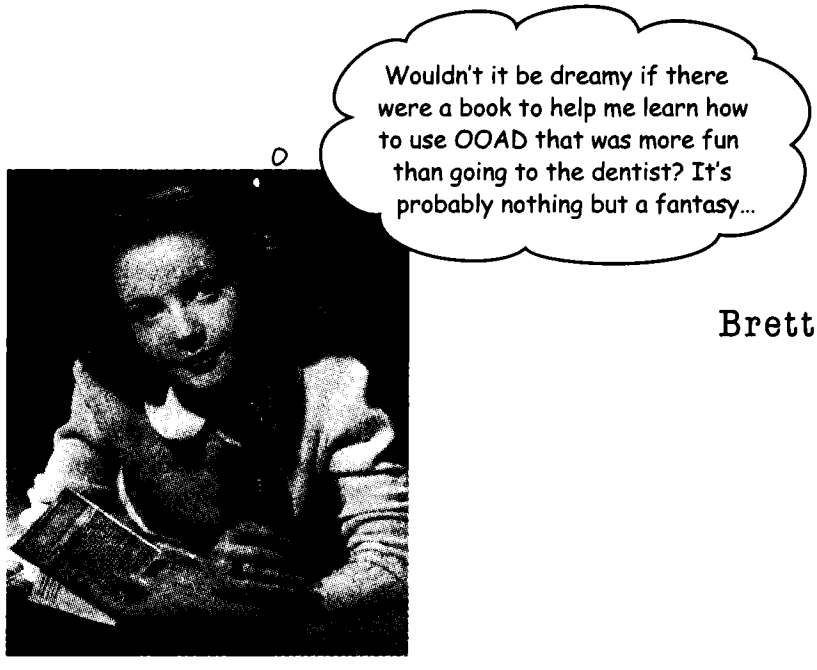
«Книга „*Программируем для iPhone и iPad*“ объясняет принципы разработки приложений iOS с самого начала. Основные изменения по сравнению с первым изданием относятся к iOS 4, Xcode 4 и написанием приложений для iPad. Благодаря пошаговым описаниям с визуальным стилем изложения материала эта книга становится отличным средством изучения программирования для iPhone и iPad во всех аспектах, от простейших до нетривиальных».

— Рич Розен, программист и соавтор книги Mac OS X for Unix Geeks

«В отличие от многих невразумительных книг по программированию, насыщенных техническим жаргоном, руководства серии *Head First jQuery* помогают новичкам создавать их первые страницы jQuery на простом и доступном уровне».

— Линдси Скурас, юрист и программист-самоучка

Head First Object-Oriented Analysis and Design



Brett D. McLaughlin
Gary Pollice
David West

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

Б. Маклафлин
Г. Поллайс
Д. Уэст

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск

2013

ББК 32.973.2-018.1
УДК 004.43
М15

Маклафлин Б., Поллайс Г., Уэст Д.
М15 Объектно-ориентированный анализ и проектирование. — СПб.: Питер, 2013. — 608 с.: ил.

ISBN 978-5-496-00144-1

Надоело читать книги по объектно-ориентированному анализу и проектированию, которые понятны только специалистам? Возможно, вы слышали, что ООАП помогает писать хорошие программы — программы, которыми будет довольно ваше начальство и заказчики. Но как это сделать?

Книга покажет вам, как организованы анализ, проектирование и написание серьезных объектно-ориентированных программ; программ, которые просты в повторном использовании, сопровождении и расширении; программ, от которых не болит голова; программ, в которые можно добавлять новые возможности, не нарушая работу старых.

Вы узнаете, какое место занимают ОО-принципы, паттерны проектирования и различные методы разработки в жизненном цикле ООАП-проектов. За счет использования особенностей работы мозга эта книга сокращает время усвоения и запоминания сложной информации. К тому времени, когда будет перевернута последняя страница, вы повеселитесь, узнаете много нового и научитесь писать хорошие программы!

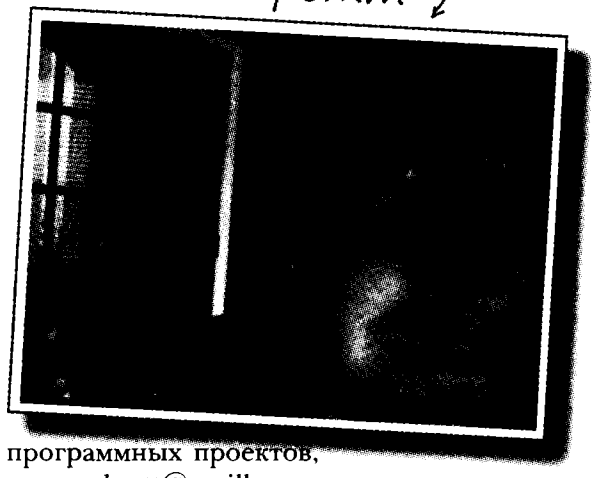
ББК 32.973.2-018.1
УДК 004.43

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

ISBN 978-0596008673 англ.
© 2013, Piter Inter Ltd.
Authorized Russian translation of the English edition of titled Head First Object-Oriented Analysis and Design, 1st Edition (ISBN 9780596008673) © 2007, O'Reilly Media, Inc.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.
ISBN 978-5-496-00144-1
© Перевод на русский язык ООО Издательство «Питер», 2013
© Издание на русском языке, оформление ООО Издательство «Питер», 2013

Всем замечательным людям, которые находили разные пути сбора требований, анализа и проектирования программного кода...

...Спасибо, что ваши находки достаточно ценны для написания хороших программ, но при этом достаточно трудны, чтобы для их объяснения понадобилась отдельная книга.



Бретт Маклафлин – гитарист, который все еще не смирился с неприятным открытием: трудно платить по счетам, играя блюз и джаз на акустической гитаре. Недавно он обнаружил, что написание книг для начинающих программистов помогает с оплатой счетов. Это очень обрадовало его самого, его жену Ли и их детей Дина и Робби.

Прежде чем войти в мир Head First, Бретт занимался разработкой корпоративных Java-приложений для Nextel Communications и Allegiance Telecom. Когда эта работа стала слишком обыденной, Бретт взялся за серверы приложений; в частности, он работал над внутренней реализацией механизма сервлетов Lutris Enhydra и контейнером EJB. Попутно Бретт «подсел» на программы с открытым кодом и помог запуску нескольких полезных программных проектов, таких как Jakarta Turbine и JDOM. С ним можно связаться по адресу brett@oreilly.com.

Гэри Поллайс, по его собственному признанию, – старый брюзга, который работает в нашей отрасли уже 35 лет, пытаясь понять, кем бы он хотел стать, когда вырастет. И хотя он до сих пор не вырос, в 2003 году он переместился в учебные аудитории, где развращает неокрепшие молодые умы радикальными идеями типа «разрабатывать программы для заказчика, учиться работать в команде, ценить качество кода, архитектуру и элегантность и не бояться выглядеть занудой – если при этом быть великим».

Гэри является практикующим профессором (то есть у него была нормальная работа, прежде чем он стал профессором) в Уорчестерском политехническом институте. Он живет в Центральном Массачусетсе с женой Викки и двумя собаками. Его домашняя страница находится по адресу http://web.cs.wpi.edu/~gpollice/. Там вы можете оставить ему весточку, обругать или похвалить книгу.

Дэйв Уэст любит говорить о передовых методах разработки ПО со страстью и энергией проповедника. Недавно Дэйв перешел на управление американским филиалом фирмы Ivar Jacobson Consulting, где он может объединить свое желание говорить о разработке и одновременно распространять информацию о регби и футболе, а также спорить о том, что крикет интереснее бейсбола.

Прежде чем взяться за американские дела Ivar Jacobson Consulting, Дэйв многие годы проработал в Rational Software (сейчас является частью IBM). Дэйв занимал разные должности в Rational и затем в IBM, вплоть до руководителя проекта RUP, где он внес в RUP идею процессных плагинов и гибких методологий. С Дэйвом можно связаться по адресу dwest@ivarjacobson.com.

Содержание (сводка)

	Введение	23
1	С чего начинаются хорошие программы. *О пользе качественного проектирования*	35
2	Дайте им то, что они хотят. *Сбор требований*	87
3	Я тебя люблю, ты мой идеал... Теперь изменись. *Изменение требований*	141
4	Программы для реального мира. *Анализ*	173
5	Часть 1. Все течет, все меняется. *Качественное проектирование*	225
	Вставка: ОО-КАТАСТРОФА	249
	Часть 2. Зарядка для программ. *Гибкость программы*	261
6	«Меня зовут Арт... И я архитектор». *Решение больших задач*	305
7	Навести порядок в хаосе. *Архитектура*	347
8	Не стремитесь к оригинальности. *Принципы проектирования*	397
9	Программы пишутся для заказчика. *Итерации и тестирование*	443
10	Все вместе. *Жизненный цикл ООАП*	501
	Приложение I. *Остатки*	571
	Приложение II. *Добро пожаловать в Объектвиль*	589

Содержание (настоящее)

Введение

Ваш мозг думает об ООАП. Вы сидите за книгой и пытаетесь что-нибудь выучить, но ваш мозг считает, что вся эта писанина не нужна. Ваш мозг говорит: «Выгляни в окно! На свете есть более важные вещи, например сноуборд». Как заставить мозг думать, что ваша жизнь действительно зависит от объектно-ориентированного анализа и проектирования?

Для кого написана эта книга?	24
Мы знаем, о чем вы думаете	25
Метапознание: наука о мышлении	27
Вот что сделали МЫ	28
Что можете сделать ВЫ	29
Примите к сведению	30
Благодарности	33

1 О пользе качественного проектирования

С чего начинаются хорошие программы

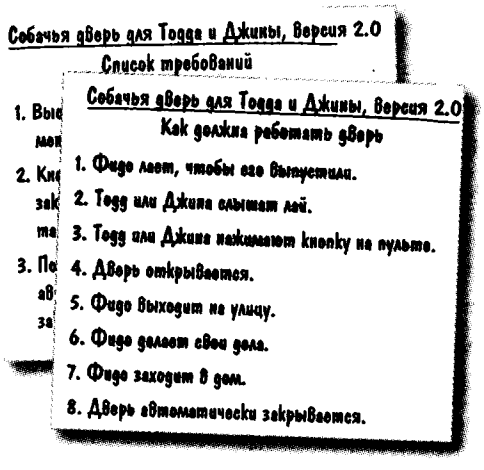
Так как же на самом деле пишутся хорошие программы? Всегда сложно понять, с чего следует начинать. Делает ли приложение то, что ему положено делать? И как насчет таких тонкостей, как дублирование кода, — ведь это всегда плохо, верно? Да, определить, **над чем следует работать в первую очередь**, бывает нелегко, к тому же еще нужно не запороть все остальное по ходу дела. Но пусть вас это не тревожит! К тому моменту, когда вы дочитаете эту главу, вы **будете знать, как пишутся хорошие программы**, а ваш подход к разработке приложений навсегда изменится. И наконец-то вы поймете, почему ООАП — это такое слово, которое не стыдно произносить в приличном обществе.

Рок-н-ролл навсегда!	36
Новенькое приложение Рика...	37
Что бы вы изменили В ПЕРВУЮ ОЧЕРЕДЬ?	42
Хорошая программа... сочетание нескольких факторов	46
Хорошая программа за 3 простых шага	47
Пробный запуск	57
Ищем проблемы	59
Анализируем метод search()	60
Теперь обновите свой код	64
Обновление класса Inventory	66
Готовимся к следующему пробному запуску	67
Возвращаемся к приложению Рика...	69
Продолжаем совершенствовать структуру кода	70
Убедимся в том, что класс Inventory хорошо спроектирован	71
Последнее тестирование (проверка готовности кода к повторному использованию)	80
Цель ООАП — написание хороших программ, а не формализация рабочего процесса!	83
Ключевые моменты	64

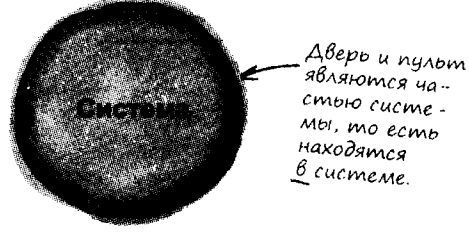
2 Сбор требований

Дайте им то, что они хотят

Хорошо, когда заказчик доволен. Вы ведь уже знаете, что написание хороших программ начинается с того, что вы проверяете, работает ли программа так, как хочет заказчик. Но как узнать, чего на самом деле хочет заказчик? И как убедиться в том, что заказчик вообще знает, чего он хочет? На помощь приходят хорошо составленные требования. В этой главе вы узнаете, как выполнить требования заказчика и предоставить ему именно то, чего он хотел. После этого на всех ваших проектах можно будет ставить «знак качества», а вы сделаете еще один большой шаг к заветной цели — написанию хороших программ... раз за разом.



Вам поручена новая задача	88
Пробный запуск	91
Создание списка требований	96
Планирование возможных неудач	100
Альтернативные пути для решения проблем	102
(Снова) о вариантах использования	104
Один вариант, три части	106
Проверка требований по варианту использования	110
Все ли на месте?	110
Ну теперь-то можно заняться написанием кода?	113
Автоматическое закрывание двери	114
Нужен новый тест!	115
Пробный запуск, версия 2.0	116
Анализ альтернативного пути	118
Пробный запуск, версия 2.1	121
Приложение работает, заказчики довольны	123
Инструментарий ООАП	138



3 Изменение требований

Я тебя люблю, ты мой идеал... Теперь изменись

Думаете, вы сделали то, что нужно заказчику? Как бы не так... Вы поговорили с заказчиком, собрали требования, записали варианты использования и выдали убойное приложение. Можно расслабиться, верно? Верно... Пока заказчик не решит, что ему нужно **что-то отличное от того, о чем он говорил вам**. То, что вы сделали, ему понравилось, честно, но **теперь это не совсем то, что нужно**. В реальном мире **требования всегда изменяются**; вы должны адаптироваться к этим изменениям и сделать так, чтобы заказчик был доволен.

Вы — герой!	142
А потом был телефонный звонок...	142
Снова беремся за карандаш	144
Единственный постоянный фактор в области анализа и проектирования программного обеспечения	145
Варианты использования должны быть понятными для вас	152
Готовимся к написанию кода...	159
Доработка списка требований	160
А теперь можно снова заняться программированием	161
Кто сказал «гав»?	162
Проверка новой двери	164
Обновление класса двери	169
Упрощение класса пульта	169
Последний пробный запуск	170
Новые инструменты ООАП	172


```
public void pressButton() {
  System.out.println("Pressing the remote control button...");
  if (door.isOpen()) {
    door.close();
  } else {
    door.open();

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      public void run() {
        door.close();
        timer.cancel();
      }
    }, 5000);
  }
}
```

Remote.java

4 Анализ

Программы для реального мира

Пора переходить к реальным приложениям. Даже если ваше приложение идеально работает на машине разработки, этого недостаточно: приложения должны работать тогда, когда их используют реальные люди. В этой главе говорится о том, как заставить вашу программу работать в реальном контексте. Вы узнаете, как текстологический анализ преобразовывает вариант использования, над которым вы работали, в классы и методы, которые делают то, что нужно вашим заказчикам. А когда работа будет завершена, вы сможете сказать: «Да, у меня получилось! Моя программа готова к жизни в реальном мире!»

Одна собака, две собаки, три...	174
Программа работает в определенном контексте	175
Выявление проблемы	176
Планирование решения	177
История двух программистов	184
Делегирование в версии Сэма	188
Сила приложений с низкой связностью	190
Обращайте внимание на существительные в варианте использования	195
От качественного анализа к хорошим классам...	208
Подробнее о диаграммах классов	210
Диаграммы классов — это еще не все	215
И как теперь работает метод recognize()?	217

Реальный мир

Качественное проектирование = гибкость программного продукта

5 (часть 1)

Все течет, все меняется

Изменения неизбежны. Какой бы замечательной ни была ваша программа сегодня, завтра ее, скорее всего, придется изменять. И чем труднее вносить поправки в программу, тем сложнее реагировать на изменяющиеся потребности заказчика. В этой главе мы вернемся к старому знакомому, попробуем улучшить существующую программу и увидим, как маленькие изменения создают большие трудности. А при этом мы обнаружим проблему настолько значительную, что для ее решения потребуется глава ИЗ ДВУХ ЧАСТЕЙ!

Струнные инструменты	226
Давайте проверим структуру кода на прочность	226
Вы обратили внимание на абстрактный базовый класс?	229
Как устроены диаграммы классов (снова)	234
Напишем код новой поисковой программы Рика	236
Создание абстрактного класса для спецификаций инструментов	237
Завершение работы над поисковой программой	240

5 (вставка)

5 (часть 2)

Хорошая структура = гибкость программы

Зарядка для программ

А вам когда-нибудь хотелось обрести большую гибкость? Если внесение изменений в приложение создает проблемы, скорее всего, это означает, что вашей программе не хватает гибкости и пластичности. Чтобы помочь приложению прийти в хорошую форму, необходимо провести анализ, основательно проработать структуру кода приложения и узнать, как ОО-принципы способствуют ослаблению связности. А в завершение вы увидите, как высокое сцепление способствует ослаблению связности. Звучит заманчиво? Переверните страницу, и мы продолжим наводить порядок в нашем негибком приложении.

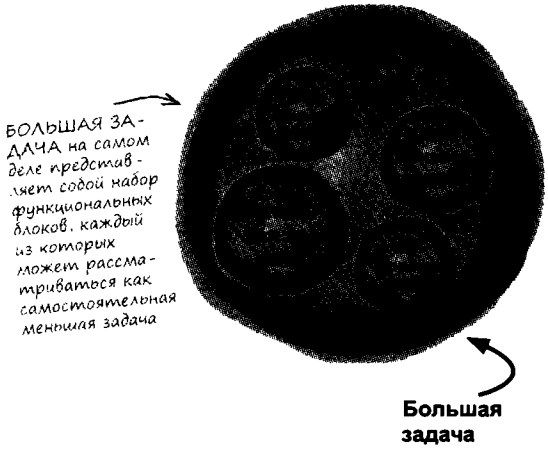
Возвращаемся к приложению Рика	262
Присмотримся повнимательнее к методу search()	265
Результаты анализа	266
Проблемы с классами инструментов	269
Но ведь классы создаются ради поведения!	269
Смерть проектировочных решений	274
Субклассы конкретных инструментов	274
Преобразуем плохие структурные решения в хорошие	275
«Двойная инкапсуляция» в программе Рика	277
Динамические свойства инструментов	278
Смотрите: гибкое приложение Рика!	286
А приложение действительно работает?	287
Пробный запуск улучшенной программы Рика	289
Высокое сцепление и одна причина для изменения	298
Когда следует сказать: «Достаточно!»	303
Инструментарий ООАП	304

6 Решение больших задач

«Меня зовут Арт... И я архитектор»

Пора построить нечто ДЕЙСТВИТЕЛЬНО БОЛЬШОЕ. Готовы? В вашем инструментарии ООАП скопилось множество инструментов, но как использовать эти инструменты при построении большого проекта? Возможно, вы этого и не осознали, но у вас имеется все необходимое для решения больших задач. В этой главе мы рассмотрим новые инструменты — анализ предметной области и диаграммы вариантов использования. Но даже эти инструменты основаны на том, что вам уже известно, — на необходимости прислушиваться к мнению заказчика и необходимости понимания того, что вы строите, до начала работы над кодом. Приготовьтесь... Пора поиграть в архитектора.

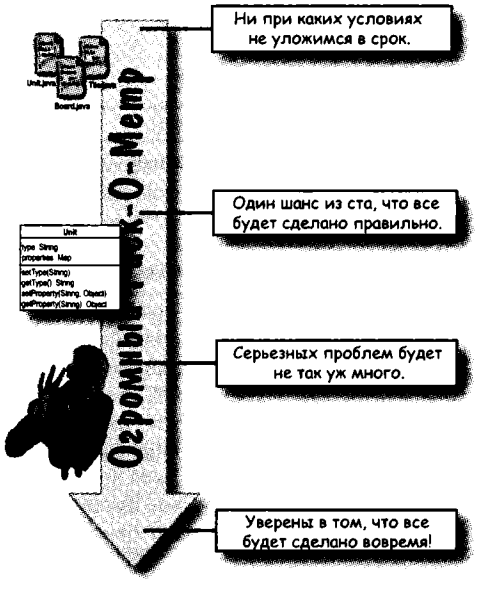
Все дело в подходе к решению большой задачи	307
Так давайте решим БОЛЬШУЮ задачу!	309
Необходимо больше информации	313
Определение функциональных возможностей	316
Используем диаграммы вариантов	322
Маленький субъект	327
Субъекты тоже люди (но не всегда)	328
Займемся анализом предметной области!	333
Разделяй и властвуй	335
Не забывайте, кто ваш заказчик	339
Что такое «паттерны проектирования»? И как их использовать?	341
Сила ООАП (и немного здравого смысла)	344
Инструментарий ООАП	346



7 Архитектура

Навести порядок в хаосе

Любая работа с чего-то начинается, но вам стоит выбрать *правильное* что-то! Вы уже знаете, как разбить приложение на множество мелких задач, но это означает, что мелких задач будет действительно МНОГО. В этой главе мы поможем вам разобраться, с чего начать и как избежать потерь времени на движение в ошибочном направлении. Пришло время взять все эти мелкие детали, разложенные на вашем рабочем месте, и понять, как собрать из них четко структурированное, хорошо спроектированное приложение. Попутно вы узнаете о трех главных вопросах архитектуры и поймете, что «Риск» — это не только интересная настольная игра из 1980-х годов.

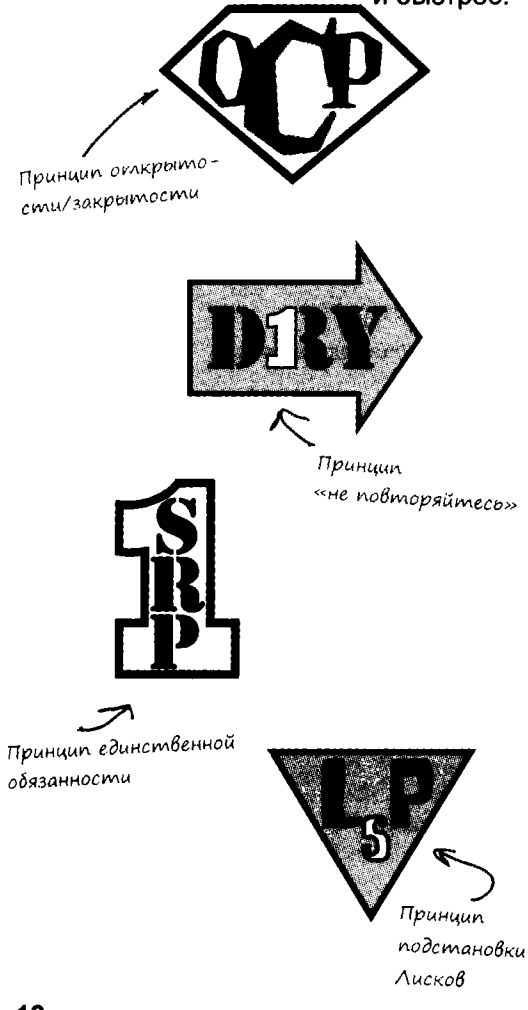


Слегка растерялись? 348
Нам нужна архитектура 350
Начнем с функциональности 353
Но что из этого считать самым важным? 353
Три основных вопроса архитектуры 356
Сценарии способствуют сокращению рисков 365
Классы Tile и Unit 372
Больше порядка, меньше хаоса 374
Чем займемся теперь? 375
Юниты для конкретной игры... что это означает? 376
Возвращаемся к общности 379
Анализ общности: путь к гибкости кода 385
Сокращение рисков помогает писать хорошие программы 395
Ключевык вопросы 396

8 Принципы проектирования

Не стремитесь к оригинальности

Имитация — самая искренняя форма проявления ума. Ничто не доставляет столько удовлетворения, как создание совершенно нового, оригинального решения задачи, которая не давала вам покоя несколько дней. Пока не выяснится, что кто-то уже решил ту же задачу задолго до вас, да еще и сделал это лучше вас! В этой главе рассматриваются принципы проектирования, которые были созданы за прошедшие годы, и то, как они помогают вам стать более квалифицированным программистом. Откажитесь от мысли «сделать по-своему», в этой главе вы узнаете, как сделать умнее и быстрее.



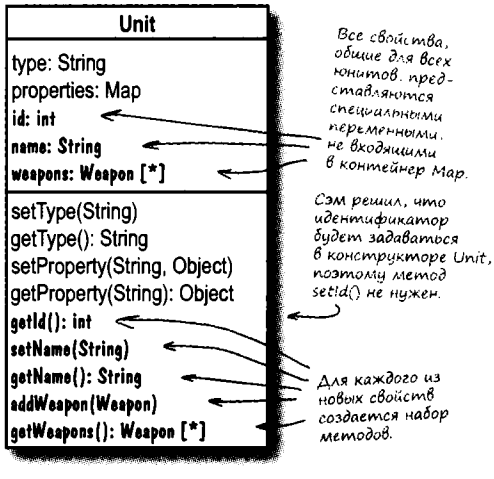
Кратко о принципах проектирования	398
Принцип № 1: принцип открытости/закрытости	399
OCP шаг за шагом	401
Принцип № 2: не повторяйтесь	404
DRY: ОДНО требование в ОДНОМ месте	406
Принцип № 3: принцип единственной обязанности	412
Выявление множественных обязанностей	414
От множественных обязанностей к единственной	417
Принцип № 4: принцип подстановки Лисков	422
Пример неправильного использования наследования	423
Принцип LSP выявляет скрытые проблемы в структуре наследования structure	424
«Замена базового типа субтипом не должна отразиться на работе программы»	425
Нарушение LSP усложняет код	426
Делегирование функциональности другому классу	428
Использование композиции для объединения поведения других классов	430
Агрегирование: композиция с продолжением существования	434
Агрегирование и композиция	435
Наследование — всего лишь одна из возможностей	436
Ключевые моменты	439
Инструментарий ООАП	440

9 Итерации и тестирование

Программы пишутся для заказчика

Пришло время показать заказчику, что вы действительно беспокоитесь о его интересах. Назойливое начальство? Обеспокоенные заказчики? Люди, которые постоянно спрашивают: «Ну что, успеете к сроку?» Никакой хорошо структурированный код не порадует вашего заказчика; вы должны показать ему что-то работающее. А теперь, когда вы овладели солидным инструментарием ОО-программирования, пора узнать, как доказать заказчику, что его программа действительно работает. В этой главе мы рассмотрим два способа углубленного анализа функциональности вашего продукта, после которого заказчик с чувством скажет: *«Да, вы определенно идеально подходите для этой работы!»*

Инструментарий постепенно заполняется 444
Но программу-то вы пишете для ЗАКАЗЧИКА! 445
Углубление итераций: два основных варианта 447
Функционально-ориентированная разработка 448
Сценарно-ориентированная разработка 449
Два метода разработки 450
Использование функционально-ориентированной разработки 453
Анализ функциональной возможности 454
Написание тестовых сценариев 457
Воспользуемся решением, ориентированным на общность 470
Сопоставьте тесты с разработанной структурой 472
Изучаем тестовые примеры... 474
Представьте результат заказчику 480
До настоящего момента мы занимались контрактным программированием 482
Контрактное программирование основано на доверии 483
Перемещение юнитов 492
Ключевые моменты 495
Инструментарий ООАП 498



10 Жизненный цикл ООАП

Все вместе

Далеко ли до цели? Мы изучили множество способов совершенствования программных продуктов, но пришло время собрать все воедино. Да, это именно тот момент, которого вы так долго ждали: мы возьмем все, о чем вы узнали из книги, и покажем, что в действительности это были лишь части общего процесса, который можно использовать вновь и вновь для написания хороших программ.

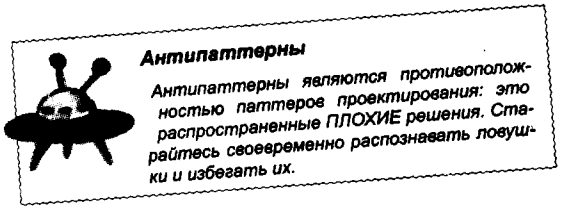
Разработка в стиле ООАП	502
Жизненный цикл проекта в стиле ООАП	503
Задача	506
Начало итераций	521
Подробнее о представлении линий метро	523
Использовать класс Line или не использовать класс Line...	532
Достопримечательности (класса) метро Объективиль	538
Защита ваших классов (и классов клиентов)	541
Возвращаемся к фазе требований...	551
Все внимание коду, потом все внимание заказчикам	553
Итеративный метод упрощает решение	557
Как выглядит маршрут?	562
Увидеть Объективиль собственными глазами	566
Нужна ли итерация №3?	569
Путешествие продолжается...	570

I Приложение I. Остатки

Десять главных тем (не рассмотренных в книге)

Хотите верьте, хотите нет, но это еще не все. Да, позади осталось более 550 страниц, и кое-что в них не уместилось. И хотя каждая из последних днсяти тем заслуживают разве что краткого упоминания, мы решили выдать вам на дорогу чуть больше информации по каждой из них. Теперь вам будет о чем поговорить во время рекламных пауз... да и кто не любит время от времени поговорить о ООАП?

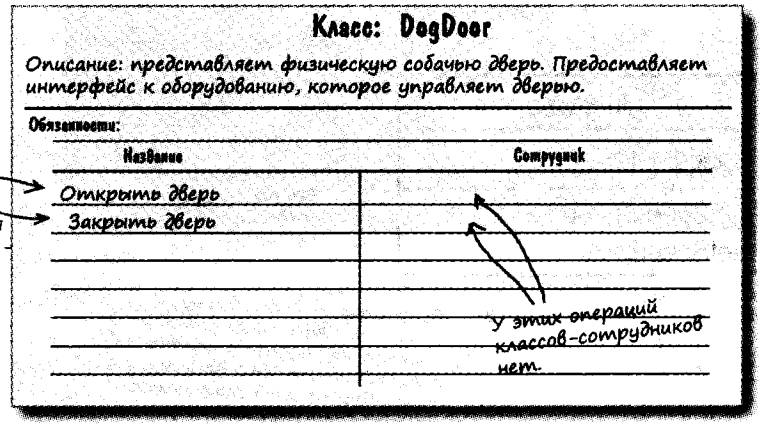
А когда закончите читать это приложение, останется еще одно... И немного рекламы... А потом — действительно всё. Честное слово!



Антипаттерны

Антипаттерны являются противоположностью паттерое проектирования: это распространенные ПЛОХИЕ решения. Старайтесь своевременно распознавать ловушки и избегать их.

1. Отношения «является» и «содержит» 572
2. Форматы вариантов использования 574
3. Антипаттерны 577
4. Карты CRC 578
5. Метрики 580
6. Диаграммы последовательности 581
7. Диаграммы состояния 582
8. Модульное тестирование 584
9. Стандарты программирования и удобочитаемость кода 586
10. Рефакторинг 588

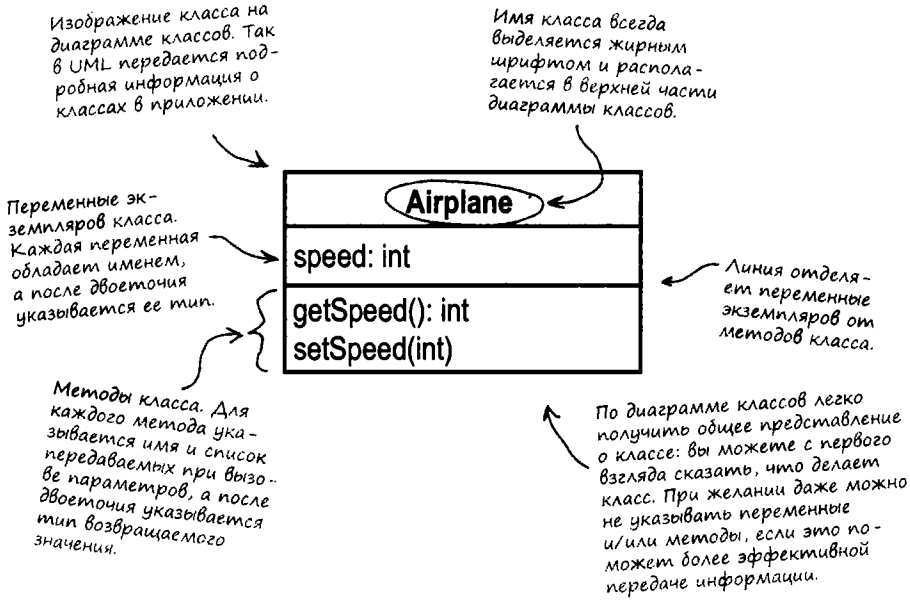


II Приложение II. Добро пожаловать в Объектвиль

Говорим на языке ООП

Приготовьтесь к путешествию в другую страну. Пришло время навестить Объектвиль — место, где объекты делают то, что им положено, все приложения хорошо инкапсулированы (вскоре вы узнаете, что это означает), а программные структуры обеспечивают простоту расширения и повторного использования. Но прежде чем браться за дело, необходимо кое-что знать заранее — в частности, овладеть кое-какими языковыми навыками. Не беспокойтесь, это не займет много времени. Вы и не заметите, как начнете говорить на языке ООП так, словно вы уже давно живете в элитном районе Объектвиля.

Добро пожаловать в Объектвиль	590
UML и диаграммы классов	591
Наследование	593
И еще полиморфизм...	595
И наконец, инкапсуляция	596
Теперь кто угодно может задать значение speed напрямую	596
И из-за чего столько шума?	597



Введение

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

Для кого написана эта книга?

Если вы ответите «да» на все следующие вопросы:

1. Вы знаете **Java**? (Быть гуру необязательно).

Наверное, вместо Java сойдет и знание C#.

2. Вы хотите **изучить, понять, запомнить и применять** объектно-ориентированный анализ и проектирование в реальных проектах и улучшить качество своих программных продуктов?
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 разбираетесь в Java? (Быть знатоком необязательно, и даже если вы не владеете Java, но знаете C#, вероятно, вы поймете большинство приводимых примеров кода. А может, будет достаточно даже C++).
2. Вы — опытный ОО-разработчик/проектировщик и вам нужен ***справочник***?
3. Вы **боитесь попробовать что-нибудь новое**? Скорее пойдете к зубному врачу, чем наденете полосатое с клетчатым? Считаете, что техническая книга с изображениями человечков серьезной быть не может?

...эта книга не для вас.

[Примечание 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.

Как 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 вышли на прогулку и вдруг прямо перед вами появляется тигр. Что происходит в вашей голове и в теле?

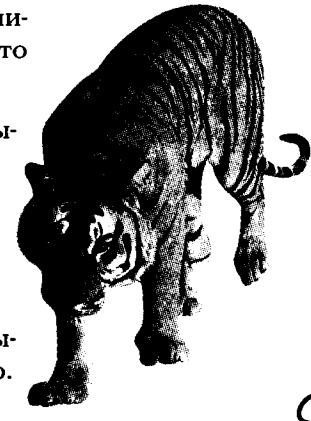
Активизируются нейроны. Вспыхивают эмоции. Происходят химические реакции. И тогда ваш мозг понимает...

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».

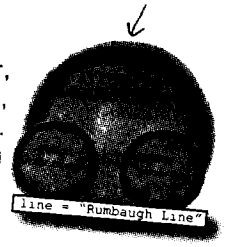


Эта книга для тех, кто хочет учиться.

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:

Все это представляется одним объектом Connection.



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 изложении материала в разговорном стиле (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. Так и живешь без тела.

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.

void roam();

У метода нет тела! Он завершается точкой с запятой.

Хорошие программы раз за разом? Я и представить не могу, что такое возможно!

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.

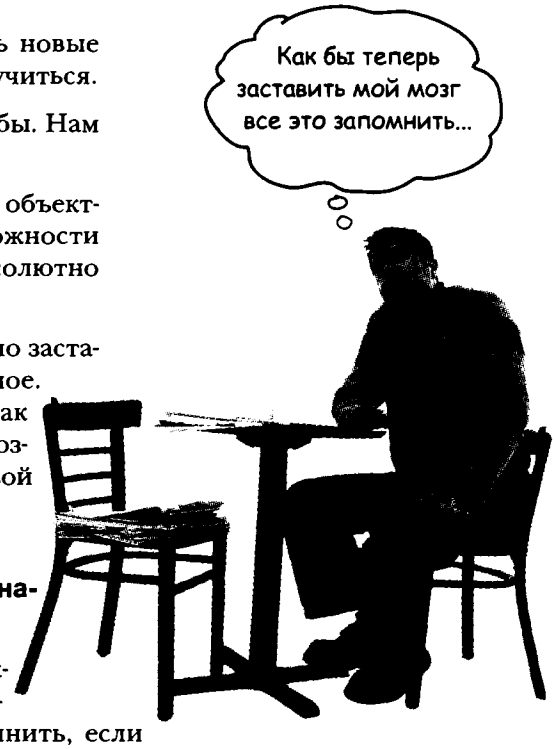
Обращение к эмоциям. Известно, что наша способность запоминать в значительной мере зависит от эмоционального сопереживания. Мы запоминаем то, что нам небезразлично. Мы запоминаем, когда что-то чувствуем. Нет, сентименты здесь ни при чем: речь идет о таких эмоциях, как удивление, любопытство, интерес 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 освоить объектно-ориентированный анализ и проектирование, и по возможности быстрее. Вы хотите *запомнить* прочитанное, а для этого абсолютно необходимо сначала *понять* прочитанное.



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 объектно-ориентированный анализ и проектирование не менее важно, чем 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».

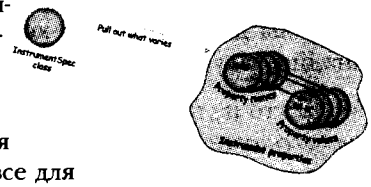
Быстрый способ основан на **повышении активности мозга** и особенно на сочетании разных ее *видов*. Доказано, что все факторы, перечисленные на предыдущей странице, помогают вашему мозгу работать на вас. Например, исследования показали, что размещение слов *внутри* рисунков (а не в подписях, в основном 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 *действительно* стоит тысячи 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.

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 что-то делаете. Мы постарались сделать их непростыми, но интересными, — то, что предпочитают большинство читателей.

КЛЮЧЕВЫЕ МОМЕНТЫ

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

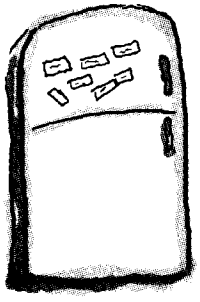
Мы постарались задействовать *оба полушария* вашего мозга; это повышает вероятность усвоения материала. Пока одна сторона мозга работает, другая часто имеет возможность отдохнуть; это повышает эффективность обучения в течение продолжительного времени.

А еще в книгу включены ***истории*** и упражнения, отражающие другие точки зрения. Мозг глубже 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.

В историях, примерах, на картинках присутствуют ***люди*** — потому что вы тоже ***человек***. И ваш мозг обращает на людей больше внимания, чем на неодушевленные ***предметы***.

А еще мы использовали принцип ***80/20***. Предполагается, что если вы собираетесь получать ученую степень в области проектирования программных продуктов, эта книга у вас будет не *единственной*. Поэтому мы не стали в ней говорить обо *всем*, а только о том, что вам действительно *необходимо*.



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.

Вырежьте и прикрепите на холодильник.

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

Просто читать недостаточно. Когда в книге вам встречается вопрос, не переходите сразу к готовому 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* обучения повышают его эффективность.

3 Читайте врезки.

Это значит: читайте все. ***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 стали поверхностно воспринимать материал или забываете только что прочитанное – пора сделать перерыв. С определенного момента попытки «затолкать» в мозг дополнительную информацию не только не ускоряют обучение, а скорее идут во вред ему.

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 Творите!

Попробуйте применить новые знания в своей повседневной работе. Просто сделайте хоть что-нибудь, чтобы приобрести практический опыт за рамками упражнений. Все, что для этого нужно, – карандаш и подходящая задача... задача, в которой изучаемые методы и инструменты могут принести пользу.

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.

Предполагается, что вы уже знакомы с Java.

Чтобы научить вас языку Java, потребовалась бы целая книга (собственно, такая книга уже есть: *Head First Java*). Мы решили сосредоточиться на анализе и проектировании, так что в книге предполагается, что читатель уже владеет основами Java. Но там, где используются какие-то нетривиальные концепции, они излагаются так, словно они полностью неизвестны читателю.

Если вы совершенно не владеете Java или пришли к этой книге с опытом программирования на C# или C++, мы настоятельно рекомендуем заглянуть в конец книги и прочитать приложение II. В нем содержится вводный материал, который поможет вам на первых порах.

Java 5 используется только там, где это необходимо.

В Java 5.0 язык был дополнен рядом новых возможностей, от параметризации до перечисляемых типов и циклов **`foreach`**. Так как многие профессиональные программисты еще только переходят на Java 5, мы решили не усложнять вам жизнь новым синтаксисом во время изучения ООАП. Как правило, в книге используется синтаксис предыдущих версий Java. Единственным исключением стала глава 1, в которой нам понадобился перечисляемый тип, — однако в этом разделе перечисляемые типы описаны достаточно подробно.

Даже если вы еще не сталкивались с Java 5, у вас не должно быть проблем с примерами кода. Если у вас уже имеется опыт программирования на Java 5, то при компилировании вы получите предупреждения о непроверенных и небезопасных операциях из-за использования нетипизованных коллекций. Впрочем, при желании вы сможете легко адаптировать код для Java 5 самостоятельно.

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 нескольку 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.

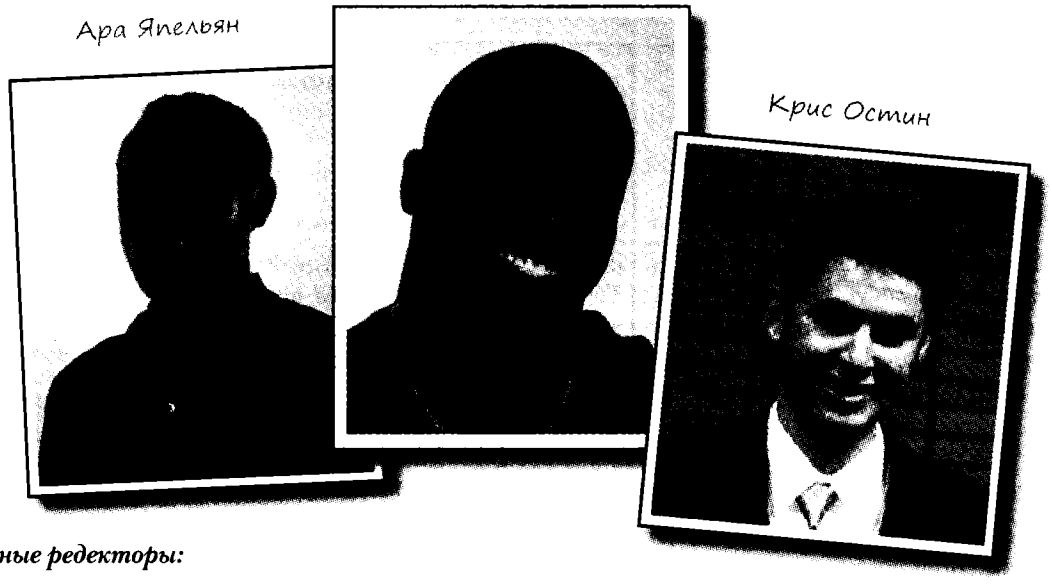
Иногда в примеры не включаются все необходимые директивы импорта; любой Java-программист должен знать, что **`ArrayList`** находится в пакете **`java.util`**. Если директивы импорта не являются частью базового J2SE API, мы упоминаем об этом в тексте. Кроме того, весь исходный код примеров размещен на веб-сайте. Вы можете загрузить его по адресу `http://www.headfirstlabs.com/books/hfoo/`.

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.

Научные редакторы



Научные редекторы:

Огромное спасибо нашему выдающемуся трио научных редакторов. Эти парни находили ошибки, пропущенные нами, указывали нам, когда мы двигались слишком быстро (или медленно), и даже сообщали о неудачных шутках. Несколько раз они справлялись с целыми главами за считанные часы... Не знаем, о чем это говорит: то ли они настолько компетентны, то ли им стоит больше внимания уделять другим делам, не связанным с программированием. Спасибо, ребята, без вашей упорной работы эта книга не была бы и наполовину так хороша.

Кэти Сьерра и Берт Бэйтс:

Мы не устаем поражаться познаниям **Берта Бэйтса** в альпинизме, а **Кэти Сьерра** – в собачьих дверях. Если вы пока не понимаете, о чем речь, не удивляйтесь – когда вы встречаетесь с этой парой, почти все встает с ног на голову. Тем не менее с их помощью все каким-то образом обращается к лучшему.

Берт и Кэти провели огромную работу по рецензированию в конце проекта, и мы благодарны им за это. Их помощь и руководство, как и прежде, занимают центральное место в серии «Head First».

Благодарности

Моим соавторам:

У меня появилась возможность ненадолго выйти из режима «мы» и поблагодарить моих соавторов, **Дэйва Уэста** и **Гэри Поллиса**. Ни один из них в начале работы не знал, на что мы подписываемся, но меня просто потрясли эти два парня, которые желали объяснять, защищать и даже изменять свое мнение по поводу проектирования ПО, требований и анализа. Они были просто бесподобны — готовили материалы до последнего дня и даже иногда давали мне возможность отдохнуть и посмеяться.

Мэри О'Брайен

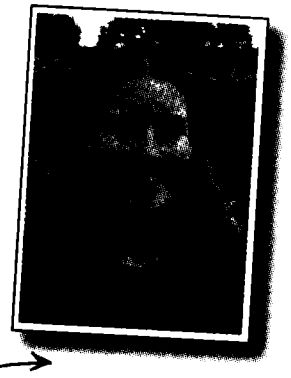
Нашему редактору:

Вы бы сейчас не держали эту книгу в руках, если бы не **Мэри О'Брайен**. Думаю, будет справедливо сказать, что она сражалась с препятствиями и прокладывала нам путь гораздо чаще, чем подозревал кто-либо из нас. Но что еще важнее, она сделала этот проект самым приятным за всю нашу карьеру. Честно говоря, она отвешивала нам пинки в нужный момент, и это сильно меняло ход работы. Она не подозревает, как сильно она влияет на людей, с которыми работает, потому что мы недостаточно часто говорим, как мы уважаем ее и ценим ее мнение. Так что теперь ты это знаешь, Мэри.

Сотрудникам O'Reilly:

Книги всегда создаются коллективными усилиями, и в данном случае это особенно справедливо. **Майк Хендриксон** и **Лори Петрики** осуществляли сопровождение проекта в разное время, а также неоднократно отвечали на срочные телефонные звонки. **Сандерс Клейнфельд** впервые опробовал свои навыки работы с серией «Head First» на этом проекте и вышел из испытания живым; и самое замечательное, что он прекрасно справился со своей работой, улучшил книгу, и мы все очень рады тому, что это была только первая из многих книг Head First, над которыми он будет работать. **Майк Лукидес** когда-то нашел Берта и Кэти, а у **Тима О'Рейли** хватило прозорливости, чтобы превратить их безумную идею в серию книг. Как обычно, **Кайл Харт** готовил книгу к печати, а прекрасный дизайн обложки **Эди Фридмана** продолжает поражать нас всех.

Мы хотим особо поблагодарить **Луизу Барр**, редактора по оформлению серии «Head First». Лу работала по 12 и 14 часов, чтобы помочь нам с графикой для книги, и нарисовала потрясающую карту метро Объектвиля для главы 10. Лу, твоя работа повысила учебную ценность этой книги, и мы не можем выразить нашу благодарность в той мере, в какой ты ее заслуживаешь.



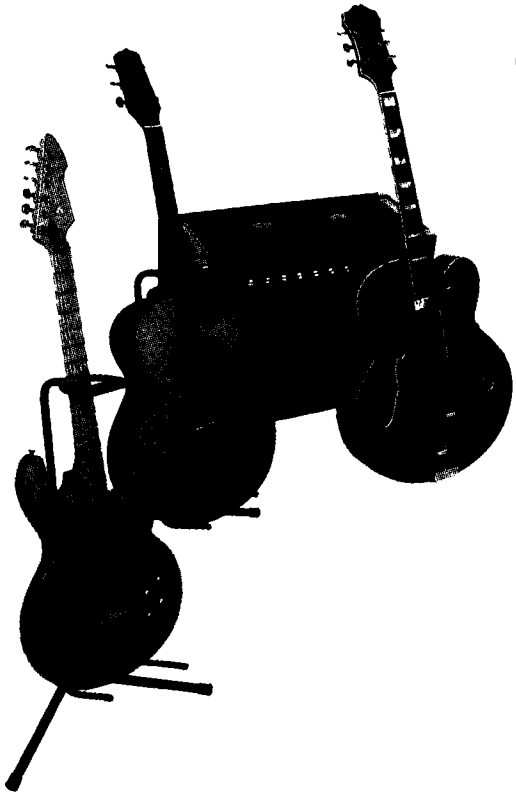
Лу Барр

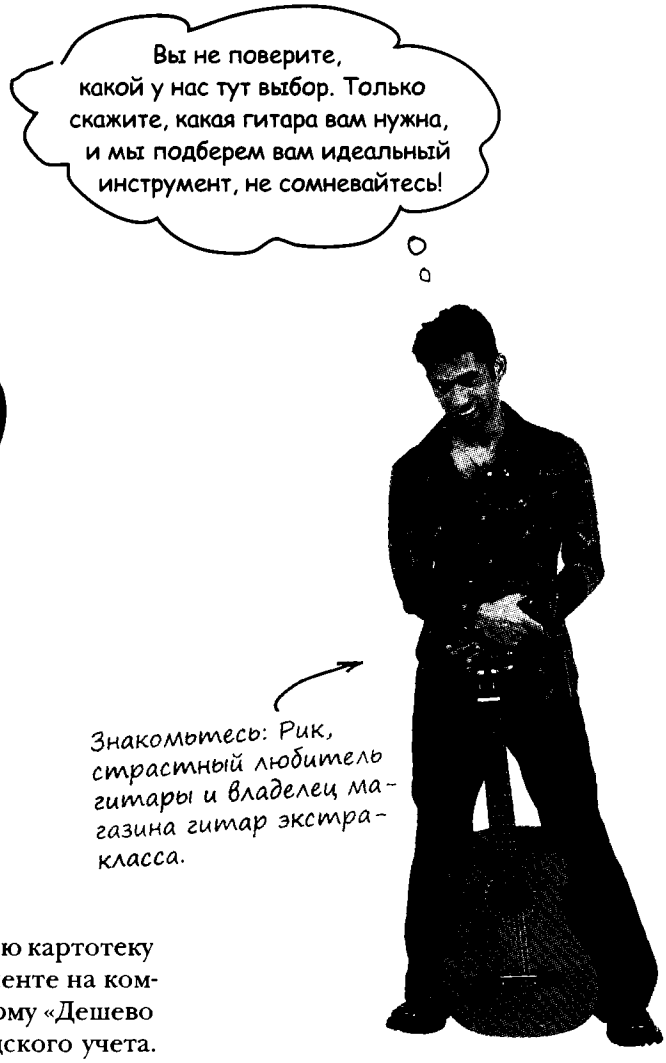
С чего начинаются хорошие программы

Так как же на самом деле пишутся хорошие программы? Всегда сложно понять, с чего следует начинать. Делает ли приложение то, что ему положено делать? И как насчет таких тонкостей, как дублирование кода, — ведь это всегда плохо, верно? Да, определить, **над чем следует работать в первую очередь**, бывает нелегко, к тому же еще нужно не запороть все остальное по ходу дела. Но пусть вас это не тревожит! К тому моменту, когда вы дочитаете эту главу, вы **будете знать, как пишутся хорошие программы**, а ваш подход к разработке приложений навсегда изменится. И наконец-то вы поймете, почему ООАП — это такое слово, которое не стыдно произносить в приличном обществе.

Рок-н-ролл навсегда!

Нет ничего лучше звука классной гитары в руках великого музыканта. Фирма Рика специализируется на поиске идеальных инструментов для своих требовательных покупателей.





Несколько месяцев назад Рик решил выбросить свою картотеку и начал хранить информацию о текущем ассортименте на компьютере. Он обратился с заказом в популярную фирму «Дешево и сердито», и там ему уже написали систему складского учета. Рик даже вытребовал средство поиска, которое поможет ему подобрать для каждого клиента «инструмент мечты».

Новенькое приложение Рика...

А вот и приложение, которое Рик получил от разработчиков из фирмы «Дешево и сердито». Построенная ими система полностью заменяет все рукописные заметки Рика и помогает ему подобрать идеальную гитару для каждого клиента. Чтобы показать, что было сделано, разработчики вручили Рику следующую диаграмму классов UML:

Каждая гитара на складе Рика представлена экземпляром этого класса.

Весь товарный запас Рика, а также средство поиска гитар.

Список всех гитар, которые в настоящее время находятся на складе.

Guitar

serialNumber: String
price: double
builder: String
model: String
type: String
backWood: String
topWood: String

getSerialNumber(): String
getPrice(): double
setPrice(float)
getBuilder(): String
getModel(): String
getType(): String
getBackWood(): String
getTopWood(): String

Inventory

guitars: List

addGuitar(String, double, String, String, String, String, String)
getGuitar(String): Guitar
search(Guitar): Guitar

Переменные класса Guitar.

Методы класса Guitar.

Метод получает серийный номер гитары и возвращает объект, представляющий эту гитару.

Метод поиска: получает описание идеальной гитары клиента и возвращает гитару из запасов Рика, соответствующую спецификации клиента.

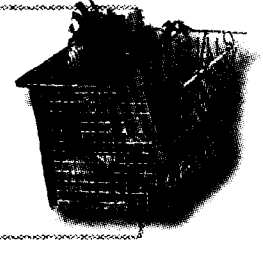
Метод получает всю информацию о гитаре, создает объект Guitar и включает его в каталог Рика.

Рик выбрал определяющие характеристики гитары: серийный номер, стоимость, производитель и модель, тип (акустическая или электрическая) и материалы.

В приложении II приведена полезная информация для тех читателей, которые еще не сталкивались с UML или ОО-программированием.

Недавно в Объектвиле?

Если вы не разбираетесь в объектно-ориентированном программировании, никогда не слышали о UML или не уверены в том, что означает приведенная диаграмма, — ничего страшного! Мы подготовили специальный краткий курс для новичков. Загляните в конец книги и прочитайте приложение II — обещаем, вы не пожалеете. А потом возвращайтесь сюда, и все происходящее покажется вам куда более осмысленным.



А вот как выглядит код Guitar.java

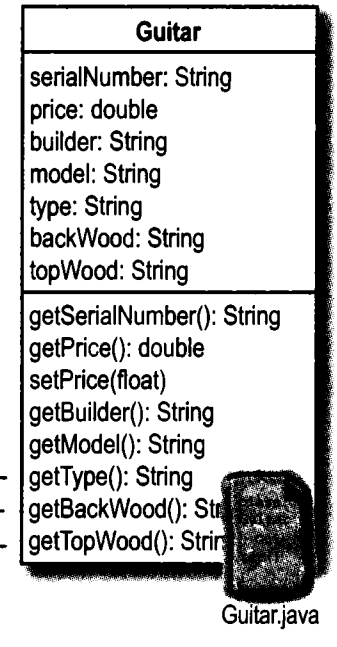
Диаграмма классов приложения Рика была приведена на предыдущей странице; а теперь давайте посмотрим, как выглядит реальный код **Guitar.java** и **Inventory.java**.

Свойства, которые мы видели на диаграмме класса Guitar.

```
public class Guitar {

  private String serialNumber, builder, model, type, backWood, topWood;
  private double price;

  public Guitar(String serialNumber, double price,
                String builder, String model, String type,
                String backWood, String topWood) {
    this.serialNumber = serialNumber;
    this.price = price;
    this.builder = builder;
    this.model = model;
    this.type = type;
    this.backWood = backWood;
    this.topWood = topWood;
  }

  public String getSerialNumber() {
    return serialNumber;
  }

  public double getPrice() {
    return price;
  }
  public void setPrice(float newPrice) {
    this.price = newPrice;
  }
  public String getBuilder() {
    return builder;
  }
  public String getModel() {
    return model;
  }
  public String getType() {
    return type;
  }
  public String getBackWood() {
    return backWood;
  }
  public String getTopWood() {
    return topWood;
  }
}
```

На диаграммах классов UML конструкторы не отображаются. Впрочем, конструктор Guitar делает ровно то, что можно от него ожидать: он задает все свойства нового объекта Guitar.

Методы на диаграмме классов соответствуют методам в коде класса Guitar.



Guitar.java

И Inventory.java...

```
public class Inventory {
  private List guitars;

  public Inventory() {
    guitars = new LinkedList();
  }

  public void addGuitar(String serialNumber, double price,
                        String builder, String model,
                        String type, String backWood, String topWood) {
    Guitar guitar = new Guitar(serialNumber, price, builder,
                               model, type, backWood, topWood);
    guitars.add(guitar);
  }
  public Guitar getGuitar(String serialNumber) {
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
      Guitar guitar = (Guitar)i.next();
      if (guitar.getSerialNumber().equals(serialNumber)) {
        return guitar;
      }
    }
    return null;
  }
  public Guitar search(Guitar searchGuitar) {
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
      Guitar guitar = (Guitar)i.next();
      // Серийный номер игнорируется, так как он уникален
      // Цена игнорируется, так как она уникальна
      String builder = searchGuitar.getBuilder();
      if ((builder != null) && (!builder.equals("")) &&
          (!builder.equals(guitar.getBuilder())))
        continue;
      String model = searchGuitar.getModel();
      if ((model != null) && (!model.equals("")) &&
          (!model.equals(guitar.getModel())))
        continue;
      String type = searchGuitar.getType();
      if ((type != null) && (!searchGuitar.equals("")) &&
          (!type.equals(guitar.getType())))
        continue;
      String backWood = searchGuitar.getBackWood();
      if ((backWood != null) && (!backWood.equals("")) &&
          (!backWood.equals(guitar.getBackWood())))
        continue;
      String topWood = searchGuitar.getTopWood();
      if ((topWood != null) && (!topWood.equals("")) &&
          (!topWood.equals(guitar.getTopWood())))
        continue;
    }
    return null;
  }
}
```

Не забудьте, что директивы импорта были опущены для экономии места.

addGuitar() получает все свойства, необходимые для создания нового экземпляра Guitar, создает его и включает в каталог.

Метод работает довольно примитивно... Каждое свойство полученного объекта Guitar сравнивается с аналогичным свойством каждого объекта Guitar в каталоге Рика.

Inventory
guitars: List
addGuitar(String, double, String, String, String, String, String) getGuitar(String): Guitar search(Guitar): Guitar

Inventory.java

А потом от Рика стали уходить клиенты...

Похоже, независимо от запросов клиента новая поисковая программа Рика всегда выдает пустой результат. Но Рик совершенно точно знает, что у него на складе есть подходящая гитара... Что происходит?

FindGuitarTester.java имитирует типичный рабочий день Рика. Приходит клиент, говорит, что ему нужно, — Рик проводит поиск по каталогу.

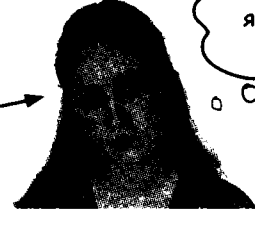
```
public class FindGuitarTester {

  public static void main(String[] args) {
    // Инициализация каталога гитар
    Inventory inventory = new Inventory();
    initializeInventory(inventory);

    Guitar whatErinLikes = new Guitar("", 0, "fender", "Stratocastor",
                                      "electric", "Alder", "Alder");
    Guitar guitar = inventory.search(whatErinLikes);
    if (guitar != null) {
      System.out.println("Erin, you might like this " +
        guitar.getBuilder() + " " + guitar.getModel() + " "
        guitar.getType() + " guitar:\n   " +
        guitar.getBackWood() + " back and sides,\n   " +
        guitar.getTopWood() + " top.\nYou can have it for only $" +
        guitar.getPrice() + "!");
    } else {
      System.out.println("Sorry, Erin, we have nothing for you.");
    }
  }

  private static void initializeInventory(Inventory inventory) {
    // Включение описаний гитар в каталог.
  }
}
```

Эрин ищет гитару фендер «Стратокастер», сделанную из ольхи (Alder).

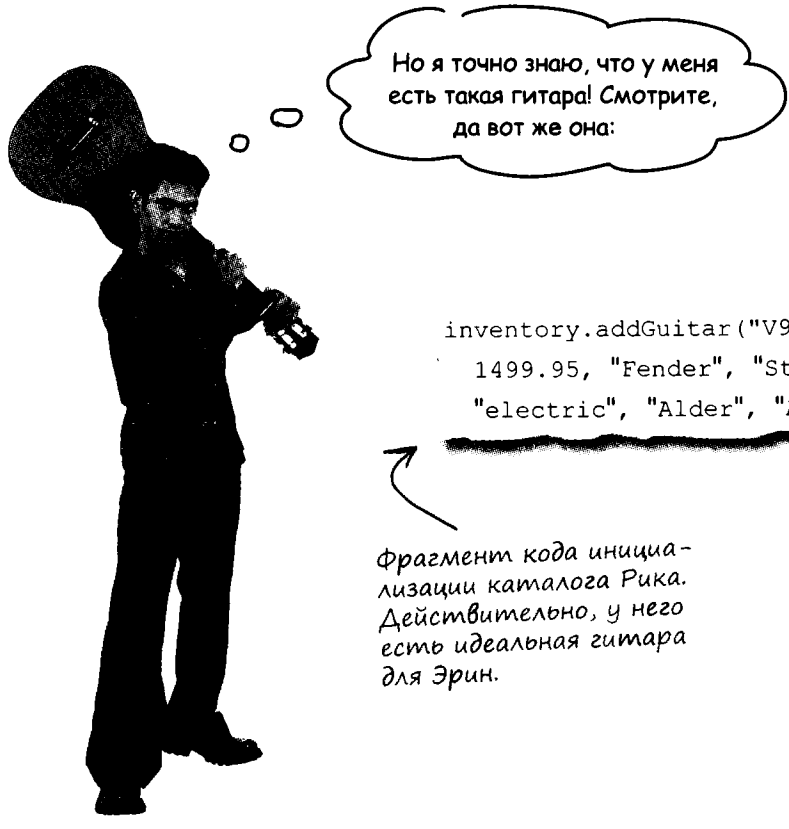
FindGuitarTester.java

```
File Edit Window Help C7#5
%java FindGuitarTester
Sorry, Erin, we have nothing for you.
```

Вот что происходит, когда Эрин приходит в магазин, а Рик пытается подобрать ей гитару.



Извини, Рик, я зайду к твоим конкурентам.




```
inventory.addGuitar("V95693",
  1499.95, "Fender", "Stratocastor",
  "electric", "Alder", "Alder");
```

Фрагмент кода инициализации каталога Рика. Действительно, у него есть идеальная гитара для Эрин.

И все характеристики совпадают с тем, что хочет Эрин... В чем дело?

Возьми в руку карандаш

Как бы вы переработали приложение Рика?

Просмотрите код приложения на трех последних страницах и результаты поиска. Какие недостатки вы заметили? Что бы вы изменили в этом коде? Запишите внизу ПЕРВОЕ, что бы вы сделали для усовершенствования приложения Рика.

Что бы Вы изменили В ПЕРВУЮ ОЧЕРЕДЬ?

Очевидно, в приложении Рика есть проблемы – но пока не совсем понятно, с чего начинать. А выбор широк, прямо глаза разбегаются:

Так... в требованиях заказчика сказано, что поиск должен выдавать все подходящие варианты. Разве метод search() не должен возвращать список совпадений?

Только посмотрите на все эти String! Ужасно... Почему не использовать константы или объекты?

Guitar

serialNumber: String
price: double
builder: String
model: String
type: String
backWood: String
topWood: String

getSerialNumber(): String
getPrice(): double
setPrice(float)
getBuilder(): String
getModel(): String
getType(): String
getBackWood(): String
getTopWood(): String

Inventory

guitars: List

addGuitar(String, double, String, String, Str
String, String)
getGuitar(String): Guitar
search(Guitar): Guitar

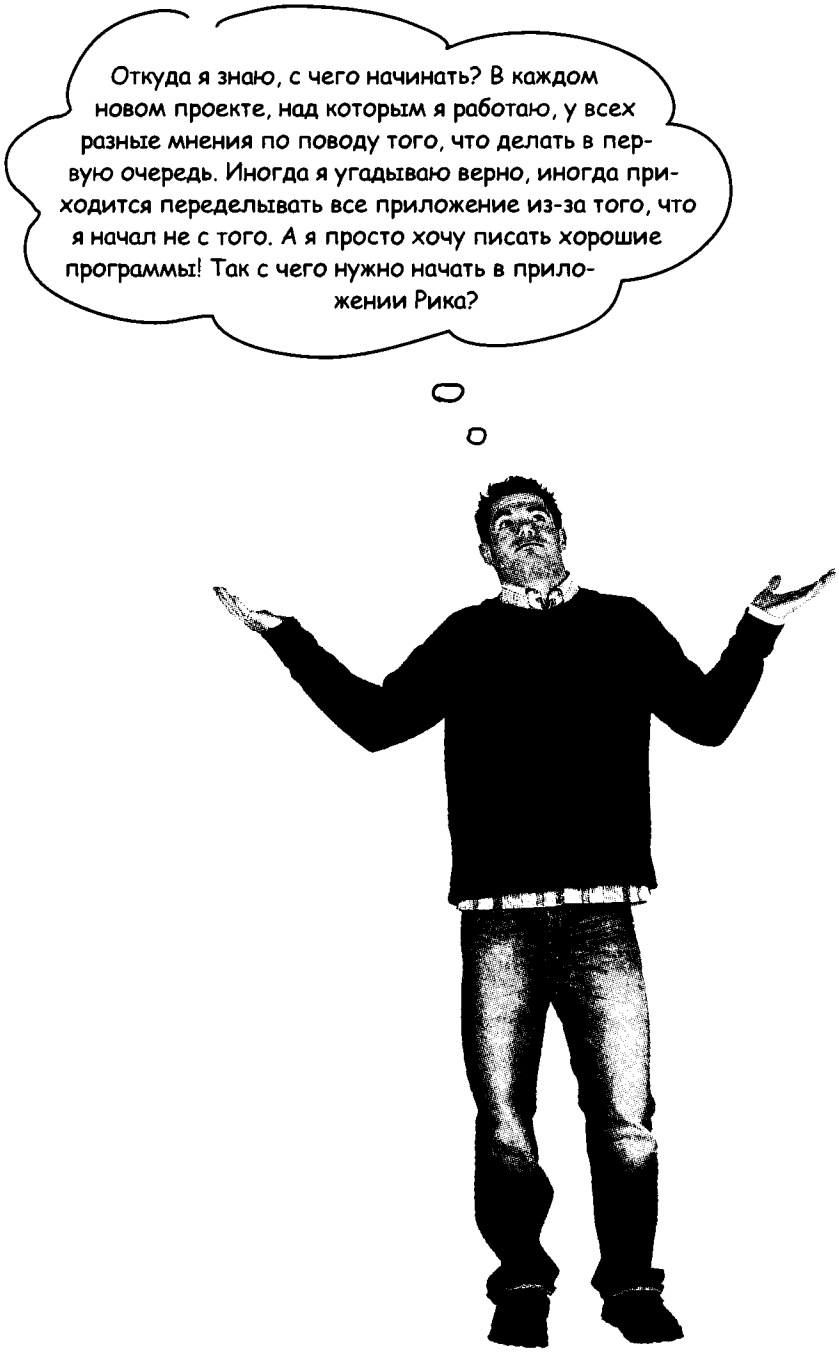
Структура кода просто отвратительна! Классы Inventory и Guitar слишком сильно зависят друг от друга. На такой основе ничего не построишь, необходима реструктуризация.

Джо недавно занимается программированием, но он твердо верит в правильность объектно-ориентированного подхода.

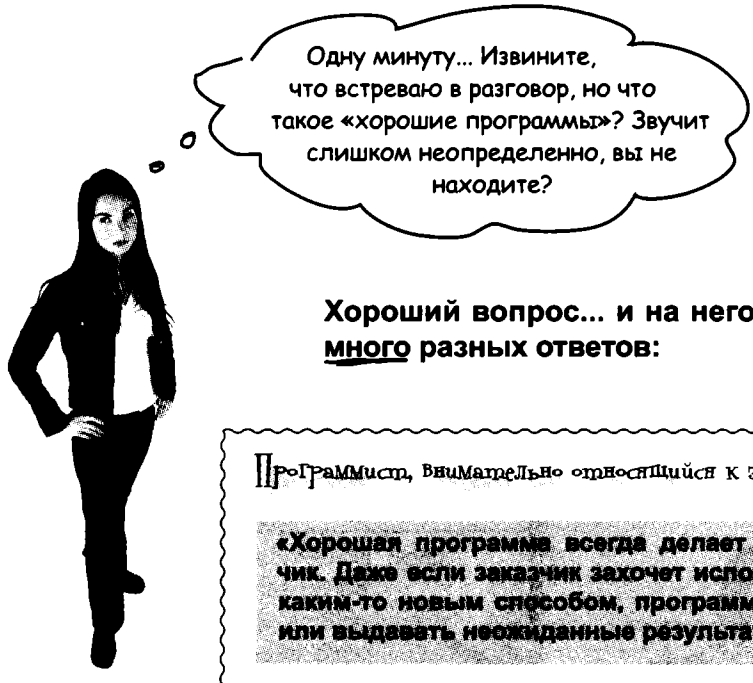
Джилл тщательно следит за точностью выполнения требований заказчика.

Фрэнк — опытный программист, он отлично разбирается в ОО-принципах и паттернах проектирования.

А с чего бы начали Вы?



Как писать хорошие программы каждый раз?



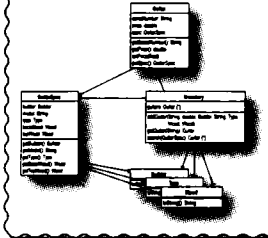
Хороший вопрос... и на него есть много разных ответов:

Программист, внимательно относящийся к заказчику:

«Хорошая программа всегда делает то, что хочет заказчик. Даже если заказчик захочет использовать программу каким-то новым способом, программа не должна сбоить или выдавать неожиданные результаты».

При таком подходе главное — чтобы заказчик был доволен тем, как работает приложение.

Объектно-ориентированный программист:



«Хорошая программа — это объектно-ориентированный код. Он не содержит дубликатов, а каждый объект более или менее полно управляет своим поведением. Такие программы легко расширять, потому что они имеют надежную и гибкую структуру кода».

Хороший ОО-программист всегда старается сделать свой код более гибким.

При структурно-ориентированном подходе код оптимизируется для расширения и повторного использования, в нем используются паттерны проектирования и проверенные ОО-методологии.

Гуру в области проектирования:

«Хорошая программа строится с использованием проверенных паттернов и принципов проектирования. Объекты обладают слабой связностью; код открыт для расширения, но закрыт для изменения. Такая структура кода также упрощает его повторное использование, поскольку вам не приходится переписывать все заново, чтобы использовать готовые части своего приложения».

Пытаетесь понять, что все это значит? Ничего страшного... узнаете в следующих главах.

Возьми в руку карандаш

А что, по *вашему* мнению, следует считать «хорошей программой»?

Вы видели мнения нескольких программистов разных типов... кто из них прав? А может, у вас есть свое определение того, что такое «хорошая программа»? Ваша очередь. Запишите те свойства, которыми, по вашему мнению, должна обладать хорошая программа:

Напишите здесь свое имя...

...а здесь — свое определение хорошей программы.

_____ :

«_____»

Хорошая программа... сочетание нескольких факторов

Чтобы понять, что именно следует считать «хорошей программой», простого определения недостаточно. Каждый из программистов на с. 44 говорил лишь об *одной* из сторон того, что делает программу действительно хорошей.

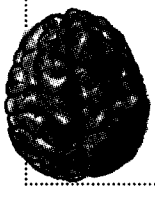
Прежде всего, хорошая программа должна удовлетворять заказчика. Она должна делать то, что от нее хочет заказчик.

Соблюдайте требования заказчика

Заказчик будет считать вашу программу хорошей, если она делает то, что ей положено делать.

Правильно работающая программа — это, конечно, хорошо... но не все. Вы понимаете это, когда потребуется дополнить код новыми возможностями или использовать его в другом приложении. Недостаточно написать программу, которая выполняет все требования заказчика; ваша программа должна пройти проверку временем.

Итак, хорошая программа хорошо спроектирована, хорошо запрограммирована, проста в сопровождении, повторном использовании и расширении.



Ваш код должен быть таким же умным, как вы сами.

Для вас (и ваших коллег) хорошая программа должна быть простой в сопровождении, расширении и повторном использовании.

Хорошая программа за 3 простых шага

Может, сейчас они не кажутся простыми, но, как вы вскоре увидите, применение ООАП и некоторых базовых принципов навсегда изменит качество вашего кода.

1. Убедитесь в том, что поведение программы соответствует требованиям заказчика.

Этот шаг ориентирован на заказчика. ПРЕЖДЕ ВСЕГО убедитесь, что приложение делает то, что положено. В этом вам помогут четко сформулированные требования и анализ.

2. Применяйте базовые ОО-принципы для повышения гибкости.

Когда программа заработает, поищите дубликаты кода, которые могли в ней остаться, и убедитесь в том, что в коде используются проверенные приемы ОО-программирования.

3. Постарайтесь создать структуру кода, упрощающую его сопровождение и повторное использование.

Получилось хорошее объектно-ориентированное приложение, которое делает то, что нужно? Примените паттерны и принципы, чтобы ваше приложение оставалось жизнеспособным в будущем.

Еще не забыли Рика? И его недовольных покупателей?

Давайте проверим наши идеи по поводу того, как пишутся хорошие программы, и посмотрим, найдется ли им место в реальном мире. У Рика не работает система поиска. Ваша задача – исправить ошибки в приложении и превратить его в хорошую программу. Давайте вернемся к приложению и посмотрим, что происходит:

Тестовая программа, которая выявляет ошибку в системе поиска.

```
public class FindGuitarTester {

  public static void main(String[] args) {
    // Инициализация каталога гитар
    Inventory inventory = new Inventory();
    initializeInventory(inventory);

    Guitar whatErinLikes = new Guitar("", 0, "fender", "Stratocastor",
                                      "electric", "Alder", "Alder");
    Guitar guitar = inventory.search(whatErinLikes);
    if (guitar != null) {
```

FindGuitarTester.java

Для этого запроса приложение Рика должно найти совпадение...

...с этой гитарой в каталоге Рика.

```
inventory.addGuitar("V95693",
  1499.95, "Fender", "Stratocastor",
  "electric", "Alder", "Alder");
```

Итак, применяем нашу систему из 3 шагов:

1. Убедитесь в том, что поведение программы соответствует требованиям заказчика.

Итак, сначала нужно убедиться в том, что приложение действительно делает то, чего хочет Рик... Сейчас это явно не так.

2. Применяйте базовые ОО-принципы для повышения гибкости.

Сейчас еще рано думать о применении паттернов и других ОО-средств... Сначала нужно заставить приложение нормально работать.

3. Постарайтесь создать структуру кода, упрощающую его сопровождение и повторное использование.

Если мы начинаем с функциональности, нужно разобраться, что происходит в неработающем методе search(). Похоже, в каталоге Рика название фирмы "Fender" записано с прописной буквы "F", а в запросе покупателя — со строчной ("fender". Метод search() должен выполнять сравнение без учета регистра символов.

Давайте обратимся за помощью к друзьям-программистам.

Фрэнк: Конечно, это решит существующую проблему... Но мне кажется, есть и более удачное решение, чем вызов toLowerCase() для множества строк по всему коду.

Джо: Да, я думал о том же. Все эти сравнения строк вообще выглядят как-то сомнительно. Почему бы не использовать для изготовителей и видов дерева константы или перечисляемые типы?

Джилл: Парни, мне кажется, вы заглядываете слишком далеко в будущее. На шаге 1 нужно исправить приложение так, чтобы оно делало то, что хочет заказчик. Я думала, что о структуре кода беспокоиться рановато.

Фрэнк: Ну да, мы должны сосредоточиться на требованиях заказчика. Но ведь мы же можем по-умному подойти к исправлению ошибок, верно? Зачем создавать проблемы, чтобы потом возвращаться и исправлять их, — не лучше ли сделать все грамотно с самого начала?

Джилл: Хмм... Да, это разумно. Наше решение этой проблемы не должно создавать новых структурных проблем в будущем. Но к другим частям приложениям мы пока не прикасаемся, верно?

Фрэнк: Верно. Мы просто удалим все эти строки и сравнения, чтобы избежать проблем с регистром.

Джо: Точно. А использование перечисляемого типа поможет проследить за тем, чтобы в каталоге хранились только действительные значения полей изготовителя, дерева и типа гитары. И тогда уж клиенты Рика совершенно точно получат гитару, соответствующую их запросу.

Джилл: Исправив ошибку, мы заодно немного усовершенствовали структуру кода... Здорово! Давайте осуществим эту идею.

Не создавайте новые проблемы для решения проблем.

Отказ от сравнений String

Итак, мы приступаем к усовершенствованию системы поиска гитар Рика. Для начала избавимся от этих раздражающих сравнений **String**. И хотя мы могли бы воспользоваться функцией **toLowerCase()**, чтобы избежать проблем с регистром символов, лучше полностью отказаться от сравнения **String**:

Каждое перечисление заменяет одно из стандартных свойств, присущих всем гитарам.

Перечисляемые типы Java отдаленно напоминают наборы констант.

```
public enum Type {

  ACOUSTIC, ELECTRIC;

  public String toString() {
    switc
      cas
      cas
      def
```

Type.java

```
public enum Builder {

  FENDER, MARTIN, GIBSON, COLLINGS,
  OLSON, RYAN, PRS, ANY;

  public String toString() {
    switch(
      case
      case
```

Builder.java

Для ссылки на значения, входящие в перечисление, можно использовать запись вида Wood.SITKA или Builder.GIBSON. Это позволит нам обойтись без сравнений строк.

```
public enum Wood {

  INDIAN_ROSEWOOD, BRAZILIAN_ROSEWOOD, MAHOGANY,
  MAPLE, COCOBOLO, CEDAR, ADIRONDACK, ALDER, SITKA;

  public String toString() {
    switch(this) {
      case INDIAN_ROSEWOOD:
        return "Indian Rosewood";
      case BRAZILIAN_ROSEWOOD:
        return "Brazilian Rosewood";
```

Wood.java

Одно из важных преимуществ перечислений заключается в том, что они ограничивают возможные значения, которые могут передаваться методу... Это решает все проблемы с неправильным написанием или регистром символов.

Часто задаваемые вопросы

В: Впервые вижу перечисления. Что это такое?

О: Перечисления — это сокращенное название для перечисляемых типов. Они доступны в C, C++, Java версии 5.0 и выше и возможно, даже будут поддерживаться в Perl 6.

Перечисляемый тип позволяет определить имя типа (например, Wood) и набор допустимых значений для этого типа (скажем, COCOBOLO, SITKA и MAHOGANY). Для обращения к этим значениям используется запись вида Wood.COCOBOLO.

В: Почему перечисляемые типы так полезны в данном примере?

```
public class FindGuitarTester {

  public static void main(String[] args) {
    // Set up Rick's guitar inventory
    Inventory inventory = new Inventory();
    initializeInventory(inventory);

    Guitar whatErinLikes = new Guitar("", 0, Builder.FENDER,
      "Stratocastor", Type.ELECTRIC, Wood.ALDER, Wood.ALDER);
    Guitar guitar = inventory.search(whatErinLikes);
    if (guitar != null) {
```

Все эти объекты String можно заменить значениями нового перечисляемого типа.

Тип String был сохранен только для модели. Для этого свойства не существует ограниченного набора значений в отличие от изготовителя и материала.

FindGuitarTester.java

Вроде бы ничего не изменилось, но с перечислениями нам не придется беспокоиться о том, что сравнение даст неверный результат из-за опечаток или расхождений в регистре символов.

```
public Guitar search(Guitar searchGuitar) {
  for (Iterator i = guitars.iterator(); i.hasNext(); ) {
    Guitar guitar = (Guitar)i.next();
    // Серийный номер игнорируется, так как он уникален
    // Цена игнорируется, так как она уникальна
    if (searchGuitar.getBuilder() != guitar.getBuilder())
      continue;
    String model = searchGuitar.getModel().toLowerCase();
    if ((model != null) && (!model.equals("")) &&
        (!model.equals(guitar.getModel().toLowerCase())))
      continue;
    if (searchGuitar.getType() != guitar.getType())
      continue;
    if (searchGuitar.getBackWood() != guitar.getBackWood())
      continue;
    if (searchGuitar.getTopWood() != guitar.getTopWood())
      continue;
    return guitar;
  }
  return null;
}
```

Регистр символов важен только для свойства model, относящегося к типу String.

Inventory.java

О: Методы и классы, использующие перечисляемые типы, защищены от передачи значений, не определенных в перечислении. Таким образом, любая опечатка или ошибка в написании приведет к ошибке компилятора. Это замечательный способ обеспечить не только безопасность типов, но и безопасность значений: он предотвращает передачу недействительных данных, не входящих в стандартный набор или множество допустимых значений.

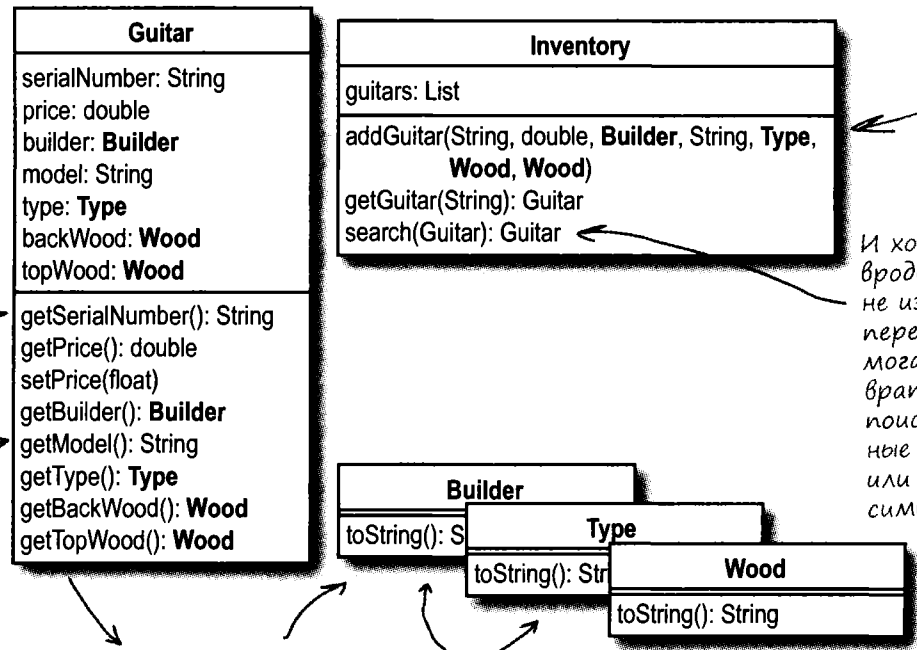
В: Я использую старую версию Java. Я ничего не смогу сделать?

О: Вовсе нет. Зайдите на сайт Head First Labs по адресу **`http://www.headfirstlabs.com`** — на нем опубликована версия приложения, которая не использует перечисления и работает со старыми версиями JDK.

Давайте взглянем на общую картину:

Теперь метод addGuitar() получает несколько перечислений (вместо объектов String или целочисленных констант).

Большую часть строковых свойств заменили перечисляемыми типами.



И хотя в search() вроде бы ничего не изменилось, перечисления помогают предотвратить ошибки поиска, вызванные опечатками или регистром символов.

Серийный номер по-прежнему уникален, а свойство model сохранило тип String, потому что существуют тысячи разных моделей гитар... При таком количестве элементов перечисление уже не приносит пользы.

Класс Guitar использует эти перечисляемые типы для представления данных, чтобы предотвратить возможные ошибки с регистром символов или написанием текста.

Наши перечисляемые типы

Так что же мы сделали?

Мы заметно приблизились к выполнению шага 1 по созданию хорошей программы. Проблема Рика с поиском, когда метод возвращал пустой результат при наличии подходящей гитары в каталоге, остались в прошлом.

Что еще лучше, мы попутно сделали приложение Рика ***менее хрупким***. Оно стало более надежным, потому что вместе с перечислениями в него вошли безопасность типов и безопасность значений. Меньше проблем для Рика, меньше хлопот с сопровождением у нас.

Код, который не является хрупким, обычно называется надежным.

1. Убедитесь в том, что поведение программы соответствует требованиям заказчика.

Возьми в руку карандаш

Применение шага 1 в вашем проекте.

А насколько внимательно вы относитесь к требованиям своих заказчиков? Запишите короткое описание текущего проекта, над которым вы работаете (или проекта, который был недавно закончен):

Теперь запишите первое, что было сделано в начале работы над проектом. Относилось ли оно к проверке соответствия кода требованиям заказчика?

Если вместо заказчика вы сосредоточились на чем-то другом, подумайте, что можно было бы сделать иначе, если бы вы знали 3 шага к построению хороших программ. Что изменилось бы при этом? Стало бы приложение лучше или хуже, чем сейчас?

Часто задаваемые вопросы

В: Значит, немного работы над структурой кода на шаге 1 не повредит, верно?

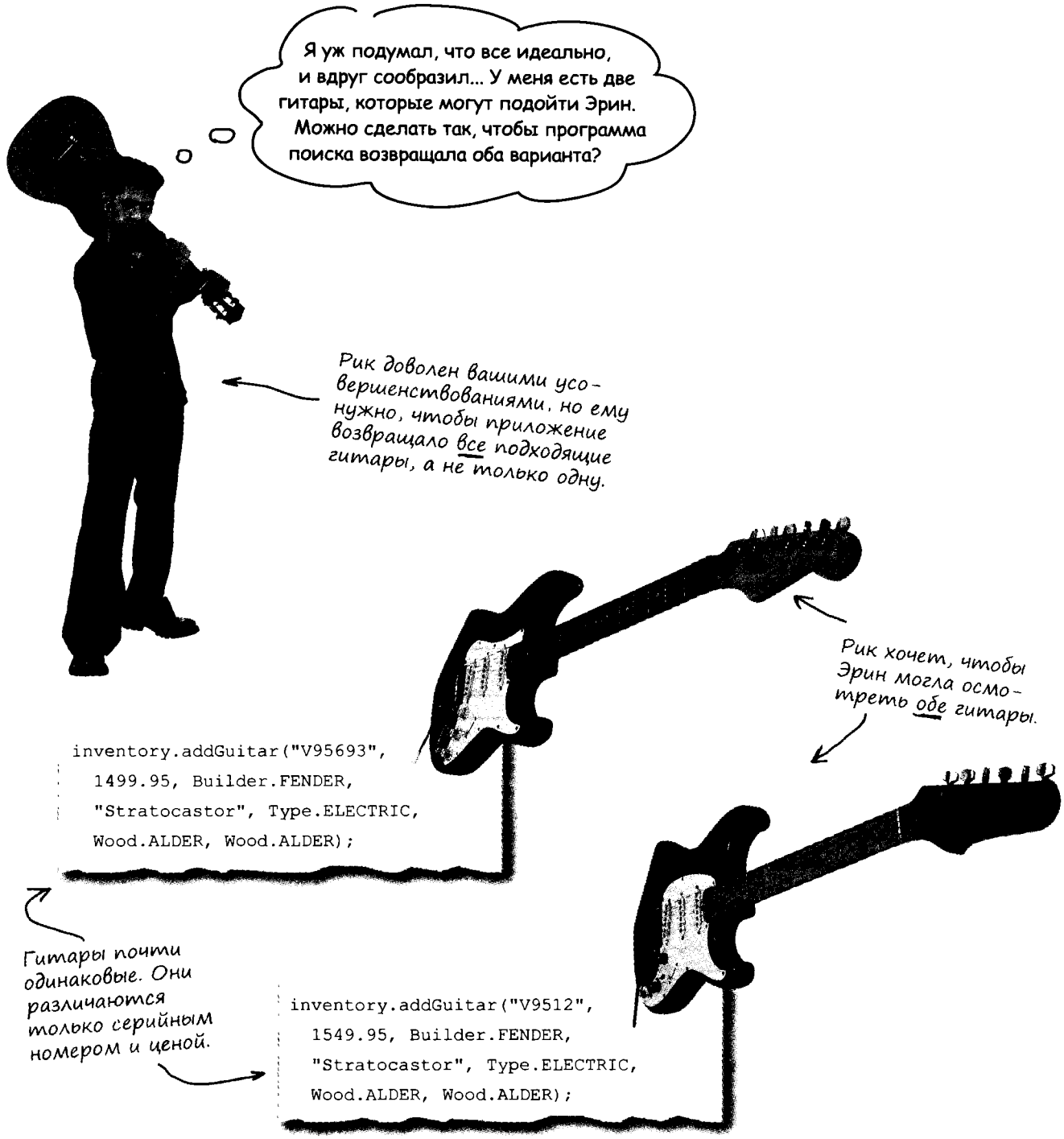
О: Да, если только при этом вы сосредоточены на потребностях заказчика. Основная функциональность приложения должна быть готова до того, как вы начнете вносить серьезные изменения в структуру. Но в ходе работы над функциональностью ничто не мешает вам использовать ОО-принципы, чтобы ваше приложение с самого начала было хорошо спроектировано.

В: Диаграмма на с. 52 называется диаграммой классов, верно? Может, это «диаграммЫ классов», раз на них показано несколько классов?

О: Да, это диаграмма классов, и на одной диаграмме может быть представлено несколько классов. Более того, на диаграммах классов может быть представлено намного больше подробностей, чем мы видели до сих пор. Мы займемся добавлением этой информации в ближайших главах.

В: Итак, мы можем перейти к шагу 2 и начать применение ОО-принципов?

О: Не совсем... Рик хотел бы исправить еще одну недоработку, прежде чем мы начнем анализировать код на предмет его возможных улучшений. Не забудьте: в первую очередь необходимо выполнить требования заказчика, а уже потом можно заняться совершенствованием ОО-структуры.

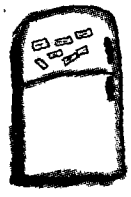

Я уж подумал, что все идеально, и вдруг сообразил... У меня есть две гитары, которые могут подойти Эрин. Можно сделать так, чтобы программа поиска возвращала оба варианта?
Рик доволен вашими усовершенствованиями, но ему нужно, чтобы приложение возвращало все подходящие гитары, а не только одну.
Рик хочет, чтобы Эрин могла осмотреть обе гитары.
inventory.addGuitar("V95693",
1499.95, Builder.FENDER,
"Stratocastor", Type.ELECTRIC,
Wood.ALDER, Wood.ALDER);
Гитары почти одинаковые. Они различаются только серийным номером и ценой.
inventory.addGuitar("V9512",
1549.95, Builder.FENDER,
"Stratocastor", Type.ELECTRIC,
Wood.ALDER, Wood.ALDER);

Покупателям нужна свобода выбора!

У Рика появилось новое требование: он хочет, чтобы программа поиска выдавала *все* гитары, соответствующие требованиям клиента, а не только первый подходящий вариант.

Inventory
guitars: List
addGuitar(String, double, Builder, String, Type, Wood, Wood) getGuitar(String): Guitar search(Guitar): **List**

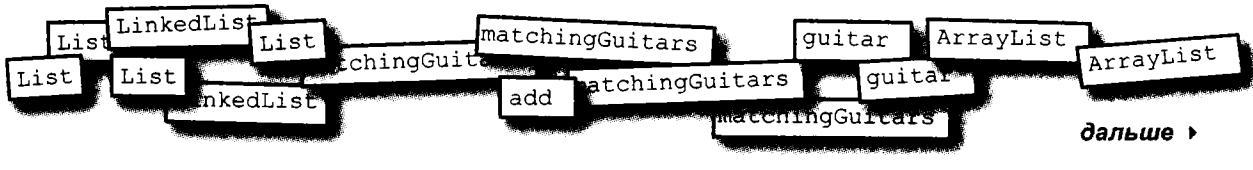
Если у Рика есть несколько гитар, соответствующих требованиям клиента, метод search() должен возвращать несколько объектов Guitar,

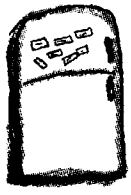


Развлечения с магнитами

Продолжим выполнение шага 1 и убедимся в том, что приложение работает правильно. Ниже приведен код метода search() из программы Рика; ваша задача — заполнить отсутствующие фрагменты. Расставьте магниты в нижней части страницы так, чтобы метод возвращал все подходящие гитары из каталога Рика.

```
public ______ search(Guitar searchGuitar) {
  ______ _______________ = new ___________();
  for (Iterator i = guitars.iterator(); i.hasNext(); ) {
    Guitar guitar = (Guitar)i.next();
    // Серийный номер игнорируется, так как он уникален
    // Цена игнорируется, так как она уникальна
    if (searchGuitar.getBuilder() != guitar.getBuilder())
      continue;
    String model = searchGuitar.getModel();
    if ((model != null) && (!model.equals("")) &&
        (!model.equals(guitar.getModel())))
      continue;
    if (searchGuitar.getType() != guitar.getType())
      continue;
    if (searchGuitar.getBackWood() != guitar.getBackWood())
      continue;
    if (searchGuitar.getTopWood() != guitar.getTopWood())
      continue;
    _______________._____(___________);
  }
  return _______________;
}
```





Развлечения с магнитами. Решение

Продолжим выполнение шага 1 и убедимся в том, что приложение работает правильно. Ниже приведен код метода search() из программы Рика; ваша задача — заполнить отсутствующие фрагменты. Расставьте магниты в нижней части страницы так, чтобы метод возвращал все подходящие гитары из каталога Рика.

```
public List search(Guitar searchGuitar) {
  List matchingGuitars = new LinkedList();
  for (Iterator i = guitars.iterator(); i.hasNext(); ) {
    Guitar guitar = (Guitar)i.next();
    // Серийный номер игнорируется, так как он уникален
    // Цена игнорируется, так как она уникальна
    if (searchGuitar.getBuilder() != guitar.getBuilder())
      continue;
    String model = searchGuitar.getModel();
    if ((model != null) && (!model.equals("")) &&
        (!model.equals(guitar.getModel())))
      continue;
    if (searchGuitar.getType() != guitar.getType())
      continue;
    if (searchGuitar.getBackWood() != guitar.getBackWood())
      continue;
    if (searchGuitar.getTopWood() != guitar.getTopWood())
      continue;
    matchingGuitars.add(guitar);
  }
  return matchingGuitars
}
```

Здесь с таким же успехом можно использовать LinkedList или ArrayList... подойдет любой из вариантов.

Подходящие гитары включаются в список вариантов, который будет выдан покупателю.

List ArrayList guitar ArrayList LinkedList

Неиспользованные магниты.

Часто задаваемые вопросы

В: Выходит, первый шаг закончится только после того, как приложение будет точно соответствовать требованиям покупателя?

О: Совершенно верно. Прежде чем браться за применение паттернов проектирования или реструктуризацию кода, убедитесь в том, чтобы приложение правильно работало.

В: Почему так важно завершить шаг 1 перед переходом к шагу 2?

О: Чтобы обеспечить правильную работу приложения, вам придется внести много изменений в свой код. Попытки совершенствования структуры кода до того, как заработает основная функциональность приложения, могут оказаться напрасными, потому что добавление новой функциональности в классы и методы может привести к масштабным структурным изменениям.

В: Похоже, вы очень сильно привязаны к этой схеме: «Шаг 1», «Шаг 2»... А если я не буду использовать ее при программировании своих приложений?

О: Конечно, никто не заставляет вас точно следовать этой схеме, но это простой путь к созданию приложений, которые делают то, что положено, хорошо спроектированы и пригодны для повторного использования. Если вам известен другой способ достижения этих целей — отлично!

Пробный запуск

Получить правильные требования от заказчика — одно дело, а реализация этих требований в коде — совсем другое. Давайте протестируем программу и посмотрим, делает ли она то, чего хочет Рик.

Тестовая программа, использующая новую версию поисковой системы Рика.

```
public class FindGuitarTester {

  public static void main(String[] args) {
    // Инициализация каталога гитар
    Inventory inventory = new Inventory();
    initializeInventory(inventory);

    Guitar whatErinLikes = new Guitar("", 0, Builder.FENDER,
                                      "Stratocastor", Type.ELECTRIC,
                                      Wood.ALDER, Wood.ALDER);
    List matchingGuitars = inventory.search(whatErinLikes);
    if (!matchingGuitars.isEmpty()) {
      System.out.println("Erin, you might like these guitars:");
      for (Iterator i = matchingGuitars.iterator(); i.hasNext(); ) {
        Guitar guitar = (Guitar)i.next();
        System.out.println("  We have a " +
          guitar.getBuilder() + " " + guitar.getModel() + " " +
          guitar.getType() + " guitar:\n   " +
          guitar.getBackWood() + " back and sides,\n   " +
          guitar.getTopWood() + " top.\n  You can have it for only $" +
          guitar.getPrice() + "!\n  ----");
      }
    } else {
      System.out.println("Sorry, Erin, we have nothing for you.");
    }
  }
```

На этот раз в программе используются перечисляемые типы. Ошибки ввода исключены!

На этот раз покупатель получает целый список гитар, соответствующих его требованиям.

В новой версии необходимо перебрать все варианты, возвращенные программой поиска.

FindGuitarTester.java

```
File Edit Window Help SweetSmell
%java FindGuitarTester
Erin, you might like these guitars:
  We have a Fender Stratocastor electric guitar:
     Alder back and sides,
     Alder top.
  You can have it for only $1499.95!
  ----
  We have a Fender Stratocastor electric guitar:
     Alder back and sides,
     Alder top.
  You can have it for only $1549.95!
  ----
```

Все работает! Эрин получает подборку вариантов гитар, а посетители магазина Рика снова начинают покупать.

Да! Это **именно** то, чего я хотел.

Вернемся к нашей схеме

Теперь, когда Рик полностью удовлетворен работой программы, можно заняться применением ОО-принципов, улучшением гибкости и проработкой структуры кода приложения.

1. Убедитесь в том, что поведение программы соответствует требованиям заказчика.

Приложение делает то, чего хотел Рик; этот шаг завершен.

2. Применяйте базовые ОО-принципы для повышения гибкости.

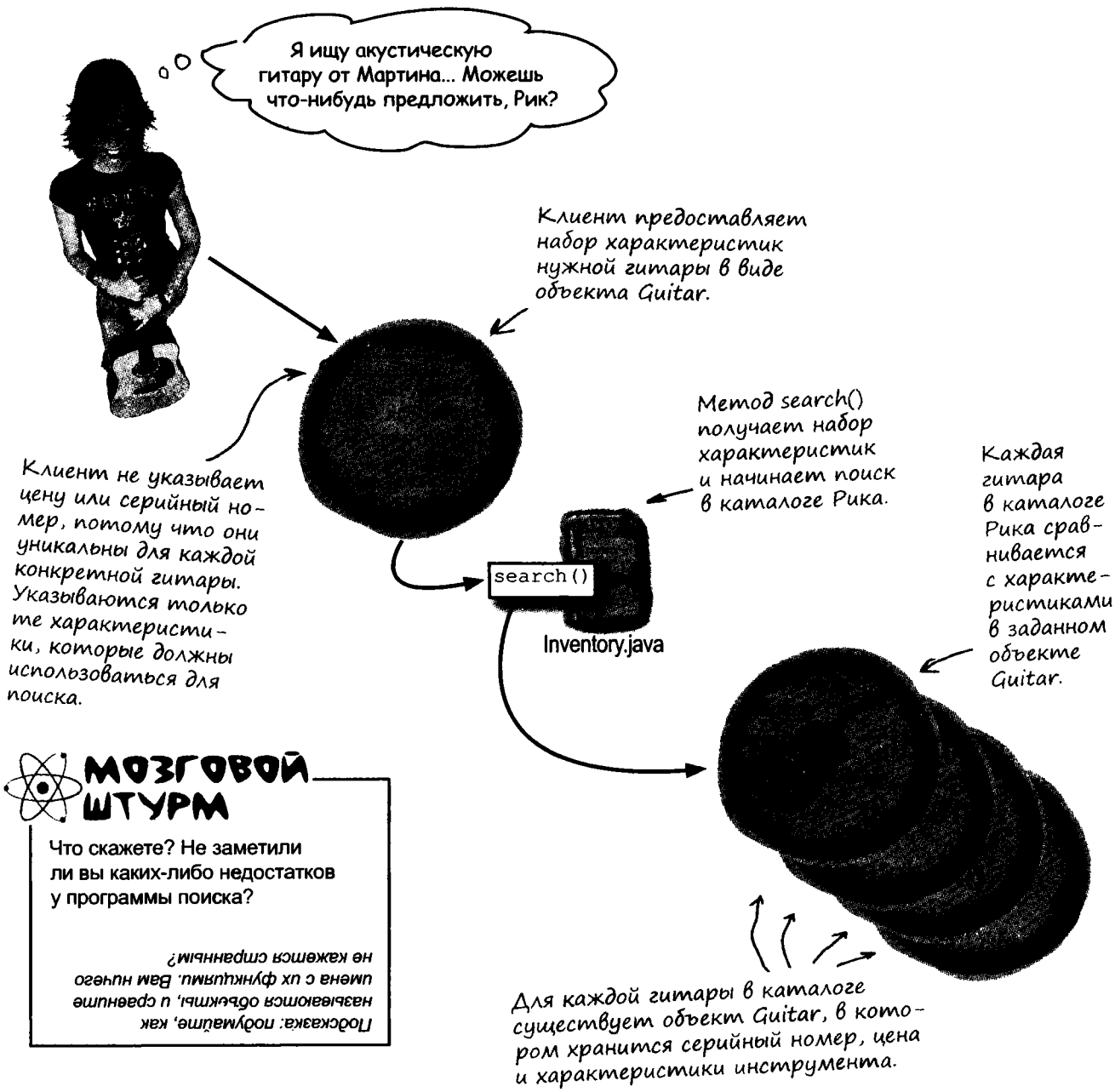
Напоминаю: здесь мы проверяем, что наше приложение не содержит дублирующегося кода, а все объекты хорошо спроектированы.

Мы берем работоспособную программу и убеждаемся в том, что она разумно устроена.

3. Постарайтесь создать структуру кода, упрощающую его сопровождение и повторное использование.

Ищем проблемы

Давайте поподробнее присмотримся к программе поиска. Нет ли в ней каких-либо проблем, которые можно было бы устранить применением простых ОО-принципов? Для начала разберемся с тем, как работает метод **search()** класса **Inventory**:



МОЗГОВОЙ ШТУРМ

Что скажете? Не заметили ли вы каких-либо недостатков у программы поиска?

Подсказка: подумайте, как называются объекты, и сравните имена с их функциями. Вам ничего не кажется странным?

Анализируем метод search()

Давайте разберемся, что же именно происходит в методе **search()** файла **Inventory.java**. Но прежде чем смотреть на код, подумаем, что *должен* делать этот метод.

1. **Клиент указывает характеристики гитары.**

   Каждый из клиентов Рика знает, какими свойствами должна обладать идеальная (в его представлении) гитара: материал, тип гитары, изготовитель или модель. Клиент передает свои пожелания Рику, а тот вводит их в программу поиска по каталогу.

   Клиент может указать только общие свойства инструмента. Он никогда не указывает серийный номер и цену.

2. **Программа поиска просматривает каталог Рика.**

   Программа осуществляет поиск по каждой гитаре в каталоге Рика.

3. **Каждая гитара сравнивается с пожеланиями клиента.**

   Для каждой гитары в каталоге Рика программа поиска проверяет, соответствует ли данная гитара пожеланиям клиента. Если совпадение будет найдено, гитара включается в список вариантов для клиента.

   Все общие свойства (материал, изготовитель) сравниваются со свойствами, указанными в запросе клиента.

4. **Клиент Рика получает список подходящих гитар.**

   Наконец, Рик и его клиент получают список подходящих гитар. Клиент может выбрать один из вариантов, а Рик зарабатывает на продаже.

Используйте текстовое описание проблемы, которую вы пытаетесь решить. Это поможет проследить за тем, чтобы структура кода соответствовала запланированной функциональности вашего приложения.

Тайна несоответствия типов объектов

В хорошо спроектированных районах Объективиля объекты очень ревностно относятся к своей работе. Каждый объект стремится выполнить свою работу — и только свою! — настолько хорошо, насколько это возможно. Для хорошо спроектированного объекта нет ничего противнее, чем применение его в каких-то посторонних целях.

К сожалению, мы видим, что в программе поиска Рика происходит именно это: объекту приходится заниматься тем, чем ему заниматься не положено. Ваша задача — раскрыть тайну и понять, как вернуть приложение Рика на путь истинный.

Чтобы вы поняли, в чем состоит недостаток программы Рика, приведем несколько подсказок, которые помогут вам выйти на правильный путь в поиске несоответствия типа объекта:

1. Объекты должны делать то, на что указывает их имя. Если объект называется Jet (самолет), он может содержать методы взлета и посадки takeOff() и land(), но метода покупки билета takeTicket() в нем быть не должно — это работа для другого объекта, и к Jet она не относится.

2. Каждый объект должен представлять одну концепцию. Объекты не должны иметь две или три разные обязанности. Объект Duck не должен представлять настоящую крякающую утку, желтую пластмассовую уточку и больничную «утку».

3. Неиспользуемые свойства — лишний балласт. Если у вас имеется объект, при использовании которого отдельным свойствам не присваиваются значения, скорее всего, этот объект имеет более одной обязанности. Если свойству редко присваиваются значения, зачем включать его в объект? Не лучше ли определить другой объект, содержащий подмножество исходных свойств?

СТОП! Попробуйте раскрыть тайну самостоятельно, прежде чем переворачивать страницу.

Как вы думаете, какой тип объекта в программе Рика используется не по назначению? Запишите свой ответ:

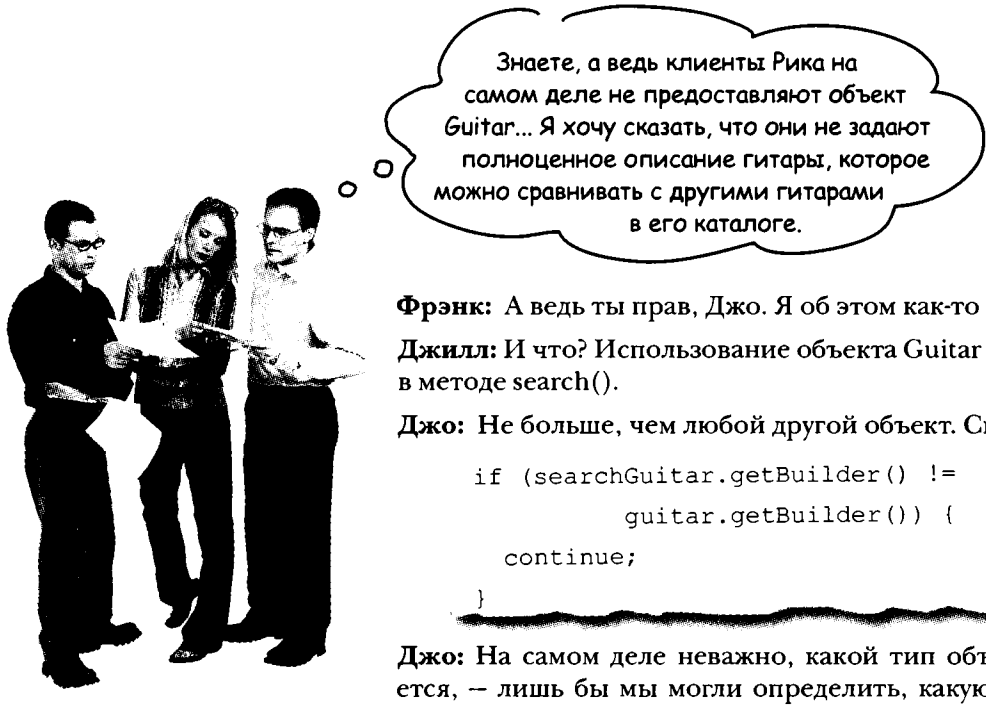
_____

Как бы вы предложили решить проблему? Какие изменения следует внести в программу?

_____

_____

_____



Фрэнк: А ведь ты прав, Джо. Я об этом как-то не задумывался.

Джилл: И что? Использование объекта Guitar упрощает сравнения в методе search().

Джо: Не больше, чем любой другой объект. Смотри:

```
if (searchGuitar.getBuilder() !=
          guitar.getBuilder()) {
  continue;
}
```

Фрагмент метода search() класса Inventory.

Джо: На самом деле неважно, какой тип объекта здесь используется, – лишь бы мы могли определить, какую конкретную гитару ищет клиент.

Фрэнк: Да, я думаю, что нам нужно создать новый объект для хранения только тех свойств, которые клиент хочет передать методу search(). Тогда при вызове не будет передаваться весь объект Guitar; мне и раньше казалось, что это нелогично.

Джилл: Но разве это не приведет к дублированию кода? Если у нас будет один объект для требований клиента и объект Guitar с полным набором свойств, то в программе появятся два метода getBuilder(), два метода getBackWood()... А это нехорошо.

Фрэнк: Тогда почему бы не инкапсулировать эти свойства из Guitar в новый объект?

Джо: Стоп... Все было замечательно, пока ты не сказал «инкапсулировать». Я думал, что инкапсуляция – это когда ты объявляешь все переменные приватными, чтобы никто не мог их использовать некорректно. Какое отношение это имеет к свойствам гитар?

Фрэнк: Под «инкапсуляцией» также понимается разбиение приложения на логические части и их последующее разделение. По аналогии с тем, как данные классов отделяются от поведения приложения, мы можем хранить общие свойства гитар отдельно от самого объекта Guitar.

Джилл: И тогда в Guitar будет храниться указатель на объект нового типа, в котором хранятся все его переменные?

Фрэнк: Точно! Мы выносим свойства гитар из Guitar и размещаем их в отдельном объекте. Смотри, это можно сделать примерно так...

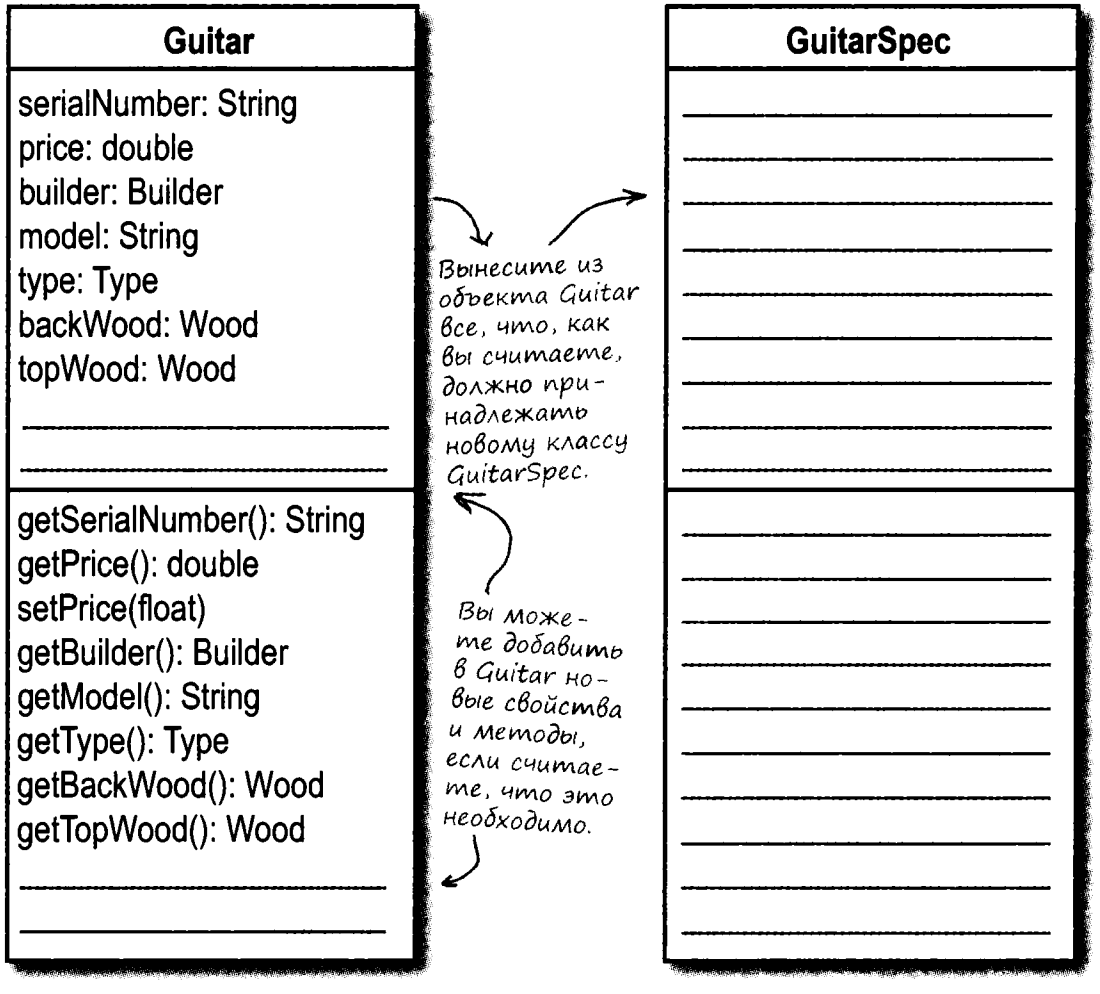
Инкапсуляция поможет скрыть внутреннее строение вашего приложения и при этом четко обозначит смысл каждой составной части.

Никогда не слышали об инкапсуляции? Откройте приложение II, прочитайте краткое введение в объектно-ориентированное программирование, а потом вернитесь сюда и продолжайте читать.

Возьми в руку карандаш

Создайте объект GuitarSpec

Ниже приведена диаграмма классов для Guitar и нового объекта GuitarSpec, о котором говорят Фрэнк, Джилл и Джо. Ваша задача — добавить свойства и методы, которые, на ваш взгляд, следует разместить в GuitarSpec. Затем вычеркните все, что должно исчезнуть из класса Guitar. Также мы добавили немного места в диаграмме класса Guitar на случай, если вам понадобится включить в него новые свойства или методы. Удачи!



* Если у вас возникнут затруднения, подумайте, что общего между объектом Guitar и теми данными, которые клиент передает методу search().

Возьми в руку карандаш
Решение

Создайте объект GuitarSpec.

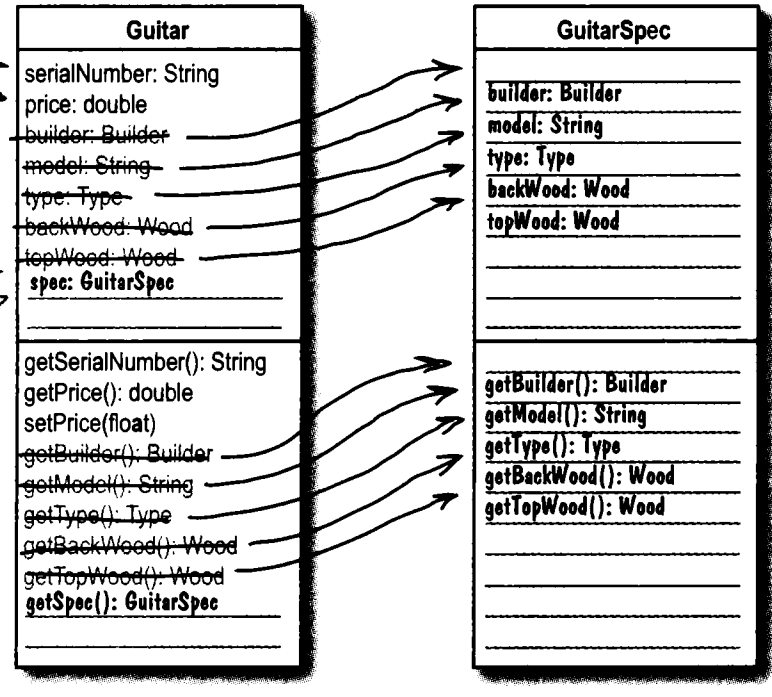
Ниже приведена диаграмма классов для Guitar и нового объекта GuitarSpec, о котором говорят Фрэнк, Джил и Джо. Ваша задача — добавить свойства и методы, которые на ваш взгляд следует разместить в GuitarSpec. Проверьте, совпадают ли ваши предложенные изменения с нашими.

Эти два свойства уникальны для каждой гитары, поэтому они остаются.

Свойства, которые передаются при вызове search(): мы перемещаем их в GuitarSpec.

Также понадобится ссылка на объект GuitarSpec для каждой гитары.

Методы перемещаются по тому же принципу, что и свойства: мы устраняем любое дублирование между спецификацией клиента и объектом Guitar.



Чтобы избежать дублирования кода, мы переместили все общие свойства (и соответствующие методы) в объект, который может использоваться как для поисковых запросов, так и для хранения информации о гитарах.

Теперь обновите свой код

С этой диаграммой классов вы сможете добавить в свое приложение класс **`GuitarSpec`**, а также обновить класс **`Guita`**. Также внесите все необходимые изменения в **`Inventory.java`**, чтобы программа поиска компилировалась без ошибок.

Часто задаваемые вопросы

В: Я понимаю, почему нам нужен объект для передачи клиентского запроса search()... Но зачем использовать этот объект еще и для хранения свойств Guitar?

О: Предположим, вы используете `GuitarSpec` для хранения данных запроса, передаваемого методу `search()`, а класс `Guitar` остался неизменным. Если в каталоге появятся 12-струнные гитары и Рик захочет добавить свойство `numStrings`, это свойство — а также код метода `getNumStrings()` — придется добавлять в оба класса, `GuitarSpec` и `Guitar`. Видите, как появляется дублирование кода?

Вместо этого мы перемещаем весь (теоретически) дублирующийся код в класс GuitarSpec. В объект Guitar включается ссылка на экземпляр класса, и мы благополучно избегаем дублирования кода.

Везде, где вы находите дублирующийся код, ищите возможности для инкапсуляции!

В: Я все еще плохо понимаю смысл этой формы инкапсуляции. Можно объяснить еще раз?

О: В основе инкапсуляции лежит идея защиты информации от других частей вашего приложения. В простейшей форме для защиты данных класса можно объявить их приватными. Но иногда такая информация включает целый набор свойств (как подробная информация о гитаре в нашем примере) и даже аспектов поведения.

Если эти свойства будут выведены за пределы класса, их можно будет изменять без изменения самого класса. Таким образом, при изменении способа хранения свойств вносить изменения в класс Guitar вообще не придется, потому что свойства изолированы от класса Guitar.

В этом проявляется мощь инкапсуляции: логическое разбиение приложения на части позволяет изменить одну часть без изменения остальных. В общем случае те части приложения, которые могут измениться, следует отделить от тех, которые останутся неизменными.

Давайте посмотрим, как мы продвигаемся на пути к созданию хорошей программы.

1. Убедитесь в том, что поведение программы соответствует требованиям заказчика.

То, чем мы занимаемся сейчас: совершенствование структуры кода.

2. Применяйте базовые ОО-принципы для повышения гибкости.

Здесь обычно скрываются проблемы, особенно связанные с дублированием кода или неудачным проектированием классов.

На этом шаге придется еще сильнее поработать над структурой кода, но зато ваше приложение станет простым в сопровождении и повторном использовании.

3. Постарайтесь создать структуру кода, упрощающую его сопровождение и повторное использование.

Обновление класса Inventory

Теперь, когда часть свойств гитары была выделена в отдельный класс, необходимо внести некоторые изменения в код.

Inventory
guitars: List
addGuitar(String, double, Builder, String, Type, Wood, Wood) getGuitar(String): Guitar search(**GuitarSpec**): List

Теперь метод search() получает GuitarSpec вместо объекта Guitar.

```
public class Inventory {

  // переменные, конструктор и другие методы

  public List search(GuitarSpec searchSpec) {
    List matchingGuitars = new LinkedList();
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
      Guitar guitar = (Guitar)i.next();
      GuitarSpec guitarSpec = guitar.getSpec();
      if (searchSpec.getBuilder() != guitarSpec.getBuilder())
        continue;
      String model = searchSpec.getModel().toLowerCase();
      if ((model != null) && (!model.equals("")) &&
          (!model.equals(guitarSpec.getModel().toLowerCase())))
        continue;
      if (searchSpec.getType() != guitarSpec.getType())
        continue;
      if (searchSpec.getBackWood() != guitarSpec.getBackWood())
        continue;
      if (searchSpec.getTopWood() != guitarSpec.getTopWood())
        continue;
      matchingGuitars.add(guitar);
    }
    return matchingGuitars;
  }
}
```

Вся информация, используемая при сравнении гитар, теперь хранится в GuitarSpec, а не в Guitar.

Этот код почти не изменился — если не считать того, что мы используем информацию из объекта GuitarSpec.

Inventory.java

Хотя классы несколько изменились, метод по-прежнему возвращает список гитар, соответствующих запросу клиента.

Готовимся к следующему пробному запуску

Для тестирования всех изменений необходимо обновить класс **FindGuitarTester**:

```
public class FindGuitarTester {

  public static void main(String[] args) {
    // Инициализация каталога гитар
    Inventory inventory = new Inventory();
    initializeInventory(inventory);

    GuitarSpec whatErinLikes =
      new GuitarSpec(Builder.FENDER, "Stratocastor", Type.ELECTRIC,
                     Wood.ALDER, Wood.ALDER);
    List matchingGuitars = inventory.search(whatErinLikes);
    if (!matchingGuitars.isEmpty()) {
      System.out.println("Erin, you might like these guitars:");
      for (Iterator i = matchingGuitars.iterator(); i.hasNext(); ) {
        Guitar guitar = (Guitar)i.next();
        GuitarSpec spec = guitar.getSpec();
        System.out.println("  We have a " +
          spec.getBuilder() + " " + spec.getModel() + " " +
          spec.getType() + " guitar:\n     " +
          spec.getBackWood() + " back and sides,\n     " +
          spec.getTopWood() + " top.\n  You can have it for only $" +
          guitar.getPrice() + "!\n  ----");
      }
    } else {
      System.out.println("Sorry, Erin, we have nothing for you.");
    }
  }

  private static void initializeInventory(Inventory inventory) {
    // Добавление гитар в каталог...
  }
}
```

На этот раз клиент передает методу search() объект GuitarSpec.

Здесь также используется новый класс GuitarSpec.

FindGuitarTester.java

В Сети

Текущую версию поисковой программы Рика можно загрузить на сайте http://www.headfirstlabs.com.

Кто и что делает?

Вы уже много узнали о том, как пишутся хорошие программы, но впереди вас ждет долгий путь! Переведите дух и вспомните некоторые термины и принципы, которые были описаны в этой главе. Соедините термины слева с описаниями этих терминов и принципов в правом столбце.

Гибкость

Инкапсуляция

Функциональность

Паттерн проектирования

Без меня заказчик никогда не будет доволен. Как бы хорошо ни было спроектировано ваше приложение, я — то, что определяет успех приложения у заказчика.

Я упрощаю повторное использование кода, а также слежу за тем, чтобы вы не пытались решать уже решенную ранее задачу.

Я используюсь для отделения неизменных частей вашего кода от частей, которые могут изменяться; благодаря мне вы сможете легко вносить изменения в свой код, не нарушая работу других частей.

Используйте меня, чтобы ваша программа могла изменяться и расти без постоянной переработки. Я борюсь с хрупкостью ваших приложений.

→ Ответы на странице 85.

Часто задаваемые вопросы

В: Инкапсуляция — не единственный ОО-принцип, который может использоваться на этой стадии?

О: Верно. Другие полезные ОО-принципы, которые могут пригодиться на этой стадии, — наследование и полиморфизм. Тем не менее оба принципа имеют отношение к дублированию кода и инкапсуляции, поэтому начинать всегда следует с поиска мест, в которых инкапсуляция могла бы улучшить структуру кода вашего приложения.

Принципы ОО-программирования будут подробно рассмотрены в книге, так же что не беспокойтесь, если сейчас что-то осталось непонятным. Вы еще узнаете много всего об инкапсуляции, проектированию классов и других темах.

В: Но я не понимаю, как инкапсуляция сделает мой код более гибким. Можно объяснить еще раз?

О: Когда программа заработает так, как ей положено, вопрос гибкости начинает играть очень важную роль. Что произойдет, если заказчик захочет добавить в приложение новые свойства или возможности? Если приложение содержит большое количество дублирующегося кода или сложные отношения наследования, внесение изменений породит массу проблем.

Применение таких принципов, как инкапсуляция и качественная структура классов, упростит внесение таких изменений, а приложение станет более гибким.

Возвращаемся к приложению Puka...

Изменения не должны повлиять на работоспособность поисковой программы. Откомпилируйте классы и снова запустите программу **FindGuitarTester**:

Результаты остались прежними, но структура кода приложения улучшилась и стала более гибкой.

```
File Edit Window Help NotQuiteTheSame
%java FindGuitarTester
Erin, you might like these guitars:
  We have a Fender Stratocastor electric guitar:
     Alder back and sides,
     Alder top.
  You can have it for only $1499.95!
  ----
  We have a Fender Stratocastor electric guitar:
     Alder back and sides,
     Alder top.
  You can have it for only $1549.95!
  ----
```

Почему вносить изменения в хорошо спроектированную программу проще, чем в программу с дублирующимся кодом? Предложите три причины.

Продолжаем совершенствовать структуру кода

Мы привели в порядок структуру и применили некоторые базовые ОО-принципы. Пора сделать второй заход. На этот раз ваше внимание должно быть обращено не только на гибкость структуры, но и на простоту повторного использования и расширения.

✓ 1. Убедитесь в том, что поведение программы соответствует требованиям заказчика.

↓

✓ 2. Применяйте базовые ОО-принципы для повышения гибкости.

↓

3. Постарайтесь создать структуру кода, упрощающую его сопровождение и повторное использование.

Пора подумать о повторном использовании и простоте изменения программного продукта. На этом этапе мы берем хорошо спроектированные классы и преобразуем их в программные блоки, хорошо подходящие для повторного использования и расширения.

После применения базовых ОО-принципов можно переходить к применению паттернов и в полной мере сосредоточиться на повторном использовании.

Убедимся в том, что класс Inventory (действительно) хорошо спроектирован

Мы уже использовали инкапсуляцию для улучшения структуры приложения Рика, однако в коде осталось еще несколько мест, в которых стоит устранить потенциальные проблемы. Это упростит расширение программы, когда Рик захочет включить в нее *очередную* полезную возможность. Кроме того, упростится повторное использование кода, то есть использование отдельных частей приложения в других контекстах.

Мы сделали для Рика работающую систему поиска. А это значит, что он непременно обратится к нам, когда ему захочется внести очередное изменение в код.

Метод search() из Inventory.java. Внимательно просмотрите этот код.

```
public List search(GuitarSpec searchSpec) {
  List matchingGuitars = new LinkedList();
  for (Iterator i = guitars.iterator(); i.hasNext(); ) {
    Guitar guitar = (Guitar)i.next();
    GuitarSpec guitarSpec = guitar.getSpec();
    if (searchSpec.getBuilder() != guitarSpec.getBuilder())
      continue;
    String model = searchSpec.getModel().toLowerCase();
    if ((model != null) && (!model.equals("")) &&
        (!model.equals(guitarSpec.getModel().toLowerCase())))
      continue;
    if (searchSpec.getType() != guitarSpec.getType())
      continue;
    if (searchSpec.getBackWood() != guitarSpec.getBackWood())
      continue;
    if (searchSpec.getTopWood() != guitarSpec.getTopWood())
      continue;
    matchingGuitars.add(guitar);
  }
  return matchingGuitars;
}
```

Inventory.java

Возьми в руку карандаш

Что бы вы изменили в этом коде?

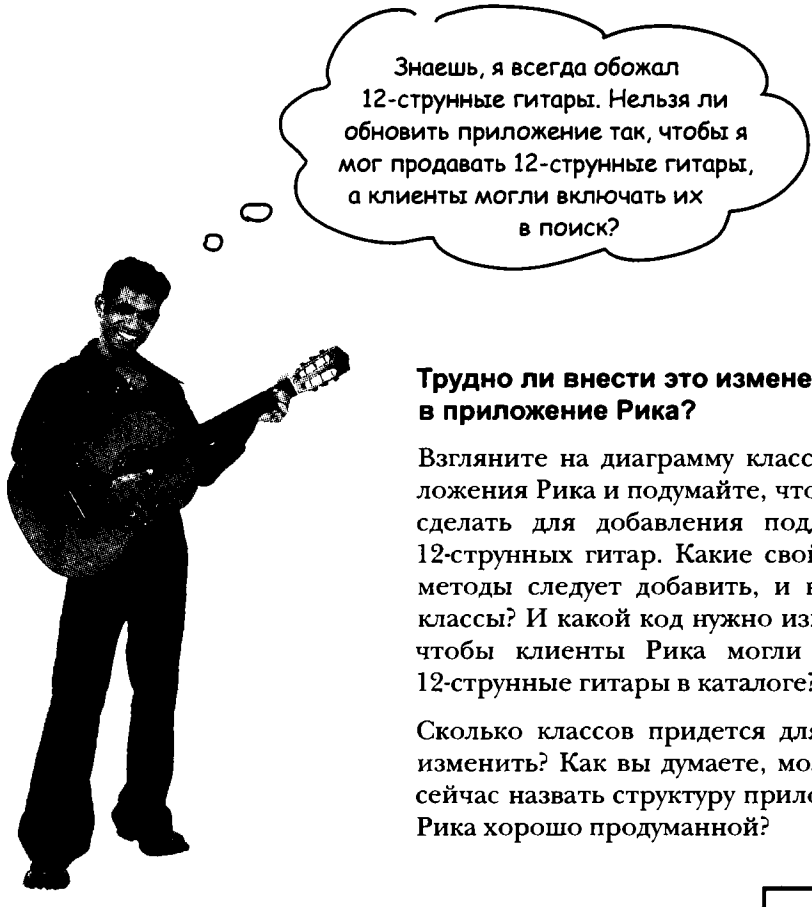
У приведенного кода имеется серьезный недостаток; попробуйте его найти. Запишите внизу, в чем, по-вашему, заключается этот недостаток и как бы вы его исправили.

_____

_____

_____

Сверьтесь с нашим ответом на с. 86.



Трудно ли внести это изменение в приложение Рика?

Взгляните на диаграмму классов приложения Рика и подумайте, что нужно сделать для добавления поддержки 12-струнных гитар. Какие свойства и методы следует добавить, и в какие классы? И какой код нужно изменить, чтобы клиенты Рика могли искать 12-струнные гитары в каталоге?

Сколько классов придется для этого изменить? Как вы думаете, можно ли сейчас назвать структуру приложения Рика хорошо продуманной?

Guitar
serialNumber: String price: double spec: GuitarSpec
getSerialNumber(): String getPrice(): double setPrice(float) getSpec(): GuitarSpec

Возьми в руку карандаш

Сделайте пометки на диаграмме классов приложения Рика.

GuitarSpec
builder: Builder model: String type: Type backWood: Wood topWood: Wood
getBuilder(): Builder getModel(): String getType(): Type getBackWood(): Wood getTopWood(): Wood

Рик хочет продавать 12-струнные гитары. Возьмите карандаш и сделайте пометки на диаграмме классов. Укажите:

1. Где бы вы добавили новое свойство с именем numStrings для хранения количества струн у гитары.

2. Где бы вы добавили новый метод с именем getNumStrings(), возвращающий количество струн у гитары.

3. Какой еще код, по вашему мнению, понадобится для того чтобы клиенты Рика могли указать в своих запросах, что им нужна 12-струнная гитара.

Наконец, запишите внизу все проблемы текущей структуры кода, с которыми вы столкнулись при добавлении поддержки 12-струнных гитар.

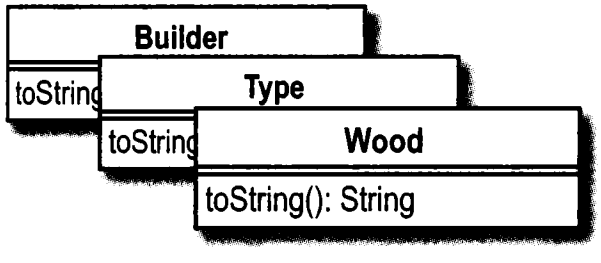
_____

_____

_____

Подсказка: ответ имеет отношение к тому, что вы записали на с. 71.

Inventory
guitars: List
addGuitar(String, double, Builder, String, Type, Wood, Wood) getGuitar(String): Guitar search(GuitarSpec): List



Мозговой штурм

Какими преимуществами обладает свойство numStrings вместо простого добавления логического свойства, указывающего, является гитара 12-струнной или нет?

Возьми в руку карандаш
Решение

Сделайте пометки на диаграмме классов приложения Рика.

Рик хочет продавать 12-струнные гитары. Возьмите карандаш и сделайте пометки на диаграмме классов. Укажите:

1. Где бы вы добавили новое свойство с именем numStrings для хранения количества струн у гитары.

2. Где бы вы добавили новый метод с именем getNumStrings(), возвращающий количество струн у гитары.

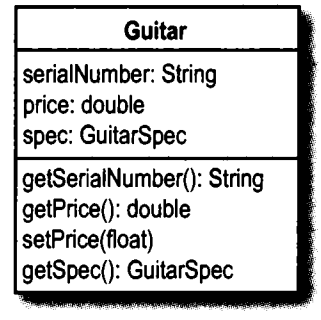
3. Какой еще код, по вашему мнению, понадобится для того чтобы клиенты Рика могли указать в своих запросах, что им нужна 12-струнная гитара.

Наконец, запишите внизу все проблемы текущей структуры, с которыми вы столкнулись при добавлении поддержки 12-струнных гитар.

Мы добавляем свойство в GuitarSpec, но нам приходится изменять код в методе search() класса Inventory, а также в конструкторе класса Guitar.

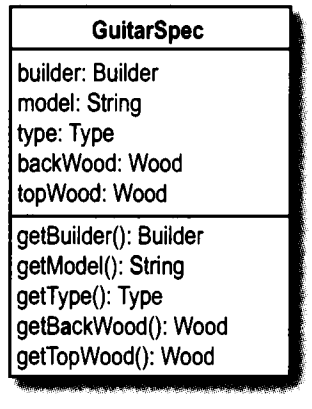
Вот то, что получилось у нас... Вы тоже написали нечто похожее?

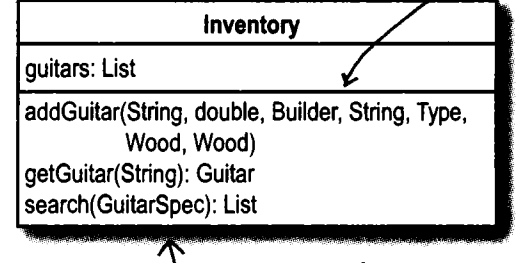
Необходимо изменить конструктор этого класса, так как он получает все свойства GuitarSpec и создает объект GuitarSpec.



Метод класса addGuitar() имеет дело со всеми свойствами гитар. Новые свойства — это проблема новых изменений метода.

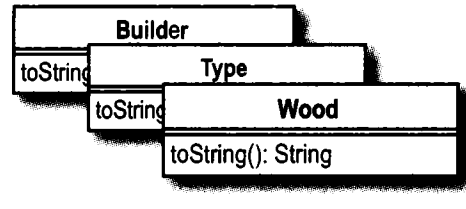
Мы должны добавить свойство numStrings в класс GuitarSpec.





В класс необходимо включить метод getNumStrings(), возвращающий количество струн у гитары.

Другая проблема: нам приходится изменять метод search(), чтобы обеспечить поддержку нового свойства в GuitarSpec.



Вот оно что... Добавление нового свойства в класс GuitarSpec не должно приводить к изменению кода в Guitar и Inventory. Нельзя ли решить проблему применением дополнительной инкапсуляции?

Верно, мы должны инкапсулировать характеристики гитары и изолировать их от остального кода системы поиска.

Хотя свойство добавляется в класс **GuitarSpec**, нам приходится изменять еще два класса: **Guitar** и **Inventory**. Конструктор **Guitar** должен получать дополнительное свойство, а метод **search()** класса **Inventory** должен выполнять дополнительное сравнение.

Этот конструктор создает объект GuitarSpec. Следовательно, при каждом изменении спецификации его код тоже придется изменять.

Повторное использование этого кода усложнено. Классы взаимозависимы, и вам не удастся использовать один класс без использования других.

```
public Guitar(String serialNumber,
              double price,
              Builder builder,
              String model, Type type,
              Wood backWood, Wood topWood) {
  this.serialNumber = serialNumber;
  this.price = price;
  this.spec = new GuitarSpec(builder, model,
                  type, backWood, topWood);
}
```

Guitar.java

```
public List search(GuitarSpec searchSpec) {
  List matchingGuitars = new LinkedList();
  for (Iterator i = guitars.iterator(); i.hasNext(); ) {
    Guitar guitar = (Guitar)i.next();
    GuitarSpec guitarSpec = guitar.getSpec();
    if (searchSpec.getBuilder() != guitarSpec.getBuilder())
      continue;
    String model = searchSpec.getModel().toLowerCase();
    if ((model != null) && (!model.equals("")) &&
        (!model.equals(guitarSpec.getModel().toLowerCase())))
      continue;
    if (searchSpec.getType() != guitarSpec.getType())
      continue;
    if (searchSpec.getBackWood() != guitarSpec.getBackWood())
      continue;
    if (searchSpec.getTopWood() != guitarSpec.getTopWood())
      continue;
    matchingGuitars.add(guitar);
  }
  return matchingGuitars;
}
```

Inventory.java

Головоломка

Мало найти недостатки в приложении Рика и определить, что оно нуждается в дальнейшей инкапсуляции. Нужно понять, как именно мы должны изменить приложение, чтобы упростить его повторное использование и расширение.

Проблема:

Добавление нового свойства в **GuitarSpec.java** приводит к изменениям в коде **Guitar.java** и **Inventory.java**. Структуру приложения следует изменить так, чтобы добавление свойств в **GuitarSpec** не влияло на код остальных компонентов приложения.

Ваша задача:

1. Добавьте свойство **numStrings** и метод **getNumStrings()** в **GuitarSpec.java**.
2. Измените **Guitar.java**, чтобы свойства **GuitarSpec** были изолированы от конструктора класса.
3. Измените метод **search()** method в **Inventory.java**, чтобы операция сравнения двух объектов **GuitarSpec** делегировалась классу **GuitarSpec** (вместо прямого сравнения).
4. Обновите код **FindGuitarTester.java** в соответствии с новыми классами и убедитесь в том, что все работает.
5. Сравните свои ответы с нашими на с. 78 и приготовьтесь протестировать полученное приложение.

Не знаете, что такое «делегирование»? Давайте разберемся...

На этом шаге от вас потребуется лишь обновить код создания каталога для использования нового конструктора Guitar.

Часто задаваемые вопросы

В: Вы сказали, что сравнения «делегируются» классу GuitarSpec. Что это такое — «делегирование»?

О: Когда объекту требуется выполнить некоторую задачу и вместо того, чтобы выполнять ее самостоятельно, он может поручить ее выполнение (или выполнение ее части) другому объекту — это и называется делегированием.

В нашей структуре кода метод search() класса Inventory должен не сравнивать два объекта GuitarSpec самостоятельно, а запросить информацию о результате сравнения у GuitarSpec. Метод search() делегирует сравнение GuitarSpec.

В: И зачем это нужно?

О: Делегирование упрощает повторное использование кода. Каждый объект имеет дело с собственной функциональностью (вместо распределения кода, относящегося к поведению одного объекта, по всему приложению).

Один из самых распространенных примеров делегирования в Java — метод equals(). Вместо того чтобы сравнивать два объекта, метод вызывает equals() для одного объекта и передает другой объект при вызове. Далее он просто получает ответ true или false от метода equals().

В: Каким образом делегирование упрощает повторное использование кода?

О: Делегирование позволяет каждому классу обеспечить выполнение проверки равенства (или любой другой операции) для объектов своего типа. Таким образом, объекты в меньшей степени зависят друг от друга, что способствует *ослаблению связности*. Слабосвязанные объекты можно извлечь из одного приложения и легко использовать в другом, потому что они не имеют жестких связей с кодом других объектов.

В: Еще раз — что такое «слабая связность»?

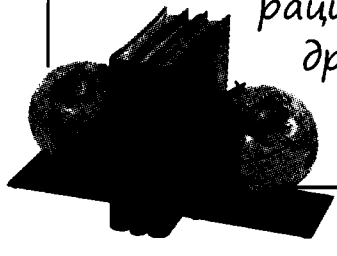
О: Это означает, что каждый объект в вашем приложении решает свою конкретную задачу и не делает ничего лишнего. Функциональность приложения делится на четко определенные объекты, каждый из которых специализируется на выполнении одной конкретной задачи.

В: И почему это хорошо?

О: Слабосвязанные приложения обычно обладают большей гибкостью и способностью к изменениям. Так как каждый объект более или менее независим от других объектов, изменения в поведении одного объекта не потребуют модификации других объектов. Это существенно упрощает добавление новых возможностей или функциональности.

За партой

Делегирование — выполнение операции объектом по поручению другого объекта.



Головоломка. Решение

Мало найти недостатки в приложении Рика и определить, что оно нуждается в дальнейшей инкапсуляции. Нужно понять, как именно мы должны изменить приложение, чтобы упростить его повторное использование и расширение.

Проблема:

Добавление нового свойства в **GuitarSpec.java** приводит к изменениям в коде **Guitar.java** и **Inventory.java**. Структуру приложения следует изменить так, чтобы добавление свойств в **GuitarSpec** не влияло на код остальных компонентов приложения.

Ваша задача:

1. Добавьте свойство **numStrings** и метод **getNumStrings()** в **GuitarSpec.java**.

Здесь все просто...

Не забудьте обновить конструктор GuitarSpec.

```
public class GuitarSpec {

  // Другие свойства
  private int numStrings;

  public GuitarSpec(Builder builder, String model,
    Type type, int numStrings, Wood backWood, Wood topWood) {
    this.builder = builder;
    this.model = model;
    this.type = type;
    this.numStrings = numStrings;
    this.backWood = backWood;
    this.topWood = topWood;
  }

  // Другие методы

  public int getNumStrings() {
    return numStrings;
  }
}
```

GuitarSpec.java

2 Измените **Guitar.java**, чтобы свойства **GuitarSpec** были изолированы от конструктора класса.

```
public Guitar(String serialNumber, double price, GuitarSpec spec) {
  this.serialNumber = serialNumber;
  this.price = price;
  this.spec = spec;
}
```

Теперь объект GuitarSpec передается, а не создается в конструкторе.

Guitar.java

3 Измените метод **search()** в **Inventory.java**, чтобы операция сравнения двух объектов **GuitarSpec** делегировалась классу **GuitarSpec** (вместо прямого сравнения).

```
public List search(GuitarSpec searchSpec) {
  List matchingGuitars = new LinkedList();
  for (Iterator i = guitars.iterator(); i.hasNext(); ) {
    Guitar guitar = (Guitar)i.next();
    if (guitar.getSpec().matches(searchSpec))
      matchingGuitars.add(guitar);
  }
  return matchingGuitars;
}
```

Метод search() стал намного проще.

Inventory.java

Большая часть кода search() переместилась в метод matches() из GuitarSpec.java.

```
public boolean matches(GuitarSpec otherSpec) {
  if (builder != otherSpec.builder)
    return false;
  if ((model != null) && (!model.equals("")) &&
      (!model.equals(otherSpec.model)))
    return false;
  if (type != otherSpec.type)
    return false;
  if (numStrings != otherSpec.numStrings)
    return false;
  if (backWood != otherSpec.backWood)
    return false;
  if (topWood != otherSpec.topWood)
    return false;
  return true;
}
```

Теперь при добавлении свойств в GuitarSpec изменения вносятся только в этом классе, но не в Guitar.java или Inventory.java.

GuitarSpec.java

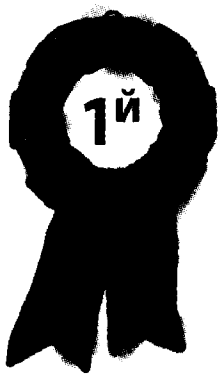
Последнее тестирование (проверка готовности кода к повторному использованию)

С того момента, когда Рик показал нам первую версию своего каталога гитар, мы проделали изрядную работу. Давайте убедимся в том, что последняя версия не нарушает требований Рика и его клиентов и соответствует нашим целям – созданию хорошо спроектированного, простого в сопровождении приложения, пригодного для повторного использования.

Результат, который должен отображаться при запуске FindGuitarTester в новом коде.

```
File Edit Window Help ReuseRules
%java FindGuitarTester
Erin, you might like these guitars:
  We have a Fender Stratocastor 6-string electric guitar:
     Alder back and sides,
     Alder top.
  You can have it for only $1499.95!
  ----
  We have a Fender Stratocastor 6-string electric guitar:
     Alder back and sides,
     Alder top.
  You can have it for only $1549.95!
  ----
```

Эрин получает пару вариантов на выбор, а Рик снова может продавать гитары своим требовательным покупателям.



Поздравляем! Вы превратили поисковую систему Рика, которая работала с ошибками, в программу с качественной структурой кода.

Что мы сделали

Давайте оглянемся назад и посмотрим, что же мы сделали для того, чтобы программа Рика заработала.

Помните наши 3 шага? Мы выполнили их, чтобы превратить неработоспособную программу в полностью функциональный, хорошо спроектированный продукт.

Мы начали с исправления некоторых функциональных ошибок в поисковой системе Рика.

1. Убедитесь в том, что поведение программы соответствует требованиям заказчика.

Затем в приложение была добавлена новая функциональность, чтобы метод поиска возвращал список гитар.

В процессе добавления новой функциональности мы позаботились о том, чтобы нововведения имели прочную структурную основу.

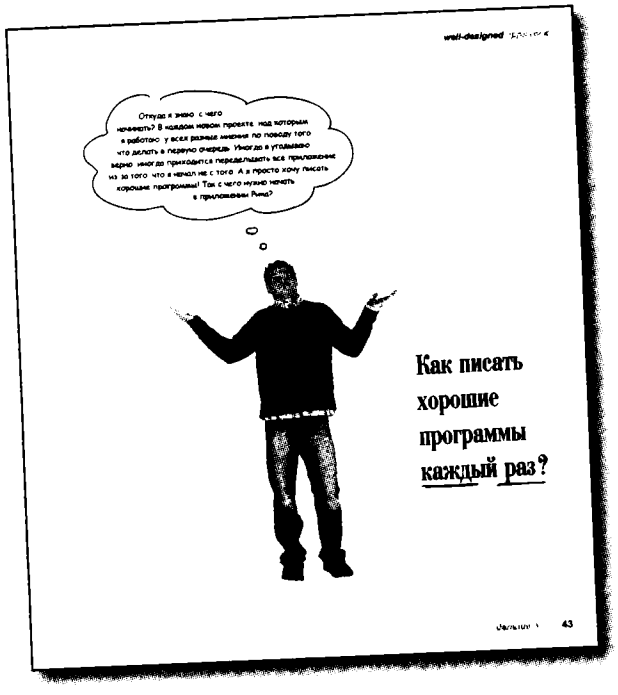
2. Применяйте базовые ОО-принципы для повышения гибкости.

Свойства гитар были инкапсулированы, чтобы в приложение можно было легко добавлять новые свойства.

3. Постарайтесь создать структуру кода, упрощающую его сопровождение и повторное использование.

Мы даже реализовали делегирование, чтобы объекты в меньшей степени зависели друг от друга, а их повторное использование создавало меньше проблем.

Помните этого несчастного?



Он просто хотел писать хорошие программы. Что же ответить на его вопрос? Как раз за разом создавать хорошие программы?

Необходима последовательность действий, выполнение которых гарантирует, что ваша программа работоспособна и хорошо спроектирована. Эта последовательность может быть простой, как три шага из приложения Рика; просто найдите ту схему, которая подходит лично вам, и используйте ее во всех своих программных проектах.

Методология объектно-ориентированного анализа и проектирования помогает неизменно создавать хорошие программы.

Или сокращенно ООАП.

Все это время, говоря о трех шагах по созданию хороших программ, мы в действительности говорили об ООАП.

ООАП – методология создания программ, которые должны делать то, что положено, и при этом быть хорошо спроектированными. А это означает гибкость кода, простоту внесения изменений, сопровождения и повторного использования кода.

Цель ООАП — написание хороших программ, а не формализация рабочего процесса!

Заказчики довольны, когда их приложения РАБОТАЮТ. Мы получаем от заказчика набор требований, по которым можно определить, решает ли создаваемое приложение поставленную задачу. Сценарии использования и диаграммы помогают в этом, но все сводится к определению того, как, по мнению заказчика, должно работать приложение.

О требованиях мы поговорим в главе 2.

Заказчики довольны, когда их приложения ПРОДОЛЖАЮТ РАБОТАТЬ. Никому не понравится, если приложение, которое прекрасно работало вчера, сегодня вдруг работать перестало. Хорошо спроектированное приложение отличается надежностью и не ломается каждый раз, когда заказчик захочет использовать его необычным образом. Классы и диаграммы последовательности помогают в представлении проблем проектирования, и все же главное — написание хорошо спроектированного, надежного кода.

Вы уже кое-что знаете о хрупкости приложений.

Заказчики довольны, когда их приложения могут СОВЕРШЕНСТВОВАТЬСЯ. Представьте: заказчик просит реализовать простую новую возможность, а ему говорят, что на это потребуется две недели работы и $25 000. Кому это понравится? Применение таких ОО-методов, как инкапсуляция, композиция и делегирование, обеспечит простое сопровождение и расширение ваших приложений.

Хотите узнать больше о делегировании, композиции и агрегировании? Мы подробно рассмотрим эти темы в главе 5, а потом вернемся к ним в главе 8.

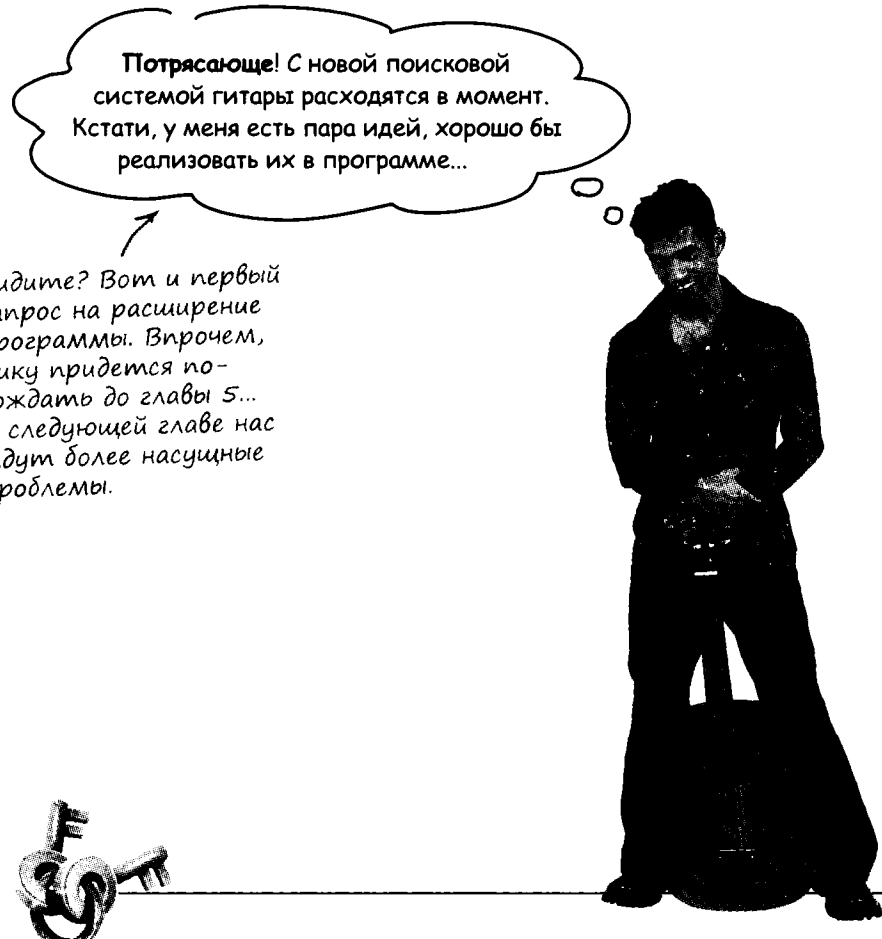
Программисты довольны, когда написанный ими код может ИСПОЛЬЗОВАТЬСЯ ПОВТОРНО. Вам приходилось когда-нибудь написать программу для одного заказчика, а потом осознать, что почти то же самое можно использовать для другого заказчика? Немного поработав над своим приложением, вы сможете обеспечить простое повторное использование кода — для этого следует избавиться от всех зависимостей и связей, которые на самом деле не нужны. В этом вам сильно помогут такие концепции, как принцип открытости/закрытости (OCP, Open-Closed Principle) и принцип единственной обязанности (SRP, Single Responsibility Principle).

Эти принципы покажут себя во всей красе в главе 8.

Программисты довольны, когда их приложения получаются ГИБКИМИ. Иногда небольшая переработка превращает хорошее приложение в библиотеку, которая может использоваться в разных целях. На этой стадии вы перестаете быть программистом-практиком и начинаете мыслить как архитектор (а вы в курсе, что эти парни зарабатывают намного больше?) Умение видеть общую картину — вот что для этого нужно.

Главы 6 и 7 посвящены тому, как научиться видеть эту «общую картину» и создавать качественную структуру кода для ваших приложений.

ВСЕ ЭТО — ООАП! И дело вовсе не в нелепых диаграммах... А в том, чтобы создавать профессиональные приложения, которые приведут в восторг ваших заказчиков, а вы почувствуете себя настоящим мастером своего дела.



КЛЮЧЕВЫЕ МОМЕНТЫ

- Чтобы нарушить работу хрупкого приложения, нужно совсем немного.
- ОО-принципы — такие, как инкапсуляция и делегирование, — помогают создавать гибкие приложения.
- Инкапсуляцией называется разбиение приложения на четко обособленные логические компоненты, позволяющие объектам скрывать свои данные и методы от других объектов.
- Делегированием называется поручение выполнения операции другому объекту.
- В начале любого проекта следует разобраться, чего же хочет заказчик.
- Когда базовая функциональность приложения будет реализована, начинайте работать над структурой, делая ее более гибкой.
- К работоспособному приложению, обладающему г структурой кода, применяются принципы проекти ния. Это приводит к дальнейшему совершенствоі структуры и упрощению повторного использования
- Найдите части приложения, которые часто изменя и постарайтесь отделить их от частей, остаюі неизменными.
- Если плохо спроектированное приложение работа у заказчика претензий не будет. Однако вам пред мучения и долгие часы сверхурочной работы, в те которых вы будете возиться с решением возника проблем.
- Объектно-ориентированный анализ и проектиро (ООАП) предоставляет механизм построения хо спроектированных приложений, которыми будет пен как заказчик, так и программист.

Кто и что делает?

Решение

Вы уже много узнали о том, как пишутся хорошие программы, но впереди вас ждет долгий путь! Переведите дух и вспомните некоторые термины и принципы, которые были описаны в этой главе. Соедините термины слева с описаниями этих терминов и принципов в правом столбце.

Гибкость

Инкапсуляция

Функциональность

Паттерн проектирования

Без меня заказчик никогда не будет доволен. Как бы хорошо ни было спроектировано ваше приложение, я — то, что определяет успех приложения у заказчика.

Я упрощаю повторное использование кода, а также слежу за тем, чтобы вы не пытались решать уже решенную ранее задачу.

Я используюсь для отделения неизменных частей вашего кода от частей, которые могут измениться; благодаря мне вы сможете легко вносить изменения в свой код, не нарушая работу других частей.

Используйте меня, чтобы ваша программа могла измениться и расти без постоянной переработки. Я борюсь с хрупкостью ваших приложений.

Возьми в руку карандаш
Решение

Что бы вы изменили в этом коде?

У приведенного кода имеется серьезный недостаток; попробуйте его найти. Запишите внизу, в чем, по-вашему, заключается этот недостаток и как бы вы его исправили.

Каждый раз, когда в GuitarSpec.java добавляется новое свойство или изменяется метод, также приходится вносить изменения в метод search() из Inventory.java. Нужно поручить сравнения GuitarSpec и инкапсулировать эти свойства из Inventory.

Архитектура оставляет желать лучшего. Каждый раз, когда в GuitarSpec будет добавляться новое свойство, этот код придется изменять.

```
public List search(GuitarSpec searchSpec) {
  List matchingGuitars = new LinkedList();
  for (Iterator i = guitars.iterator(); i.hasNext(); ) {
    Guitar guitar = (Guitar)i.next();
    GuitarSpec guitarSpec = guitar.getSpec();
    if (searchSpec.getBuilder() != guitarSpec.getBuilder())
      continue;
    String model = searchSpec.getModel().toLowerCase();
    if ((model != null) && (!model.equals("")) &&
        (!model.equals(guitarSpec.getModel().toLowerCase())))
      continue;
    if (searchSpec.getType() != guitarSpec.getType())
      continue;
    if (searchSpec.getBackWood() != guitarSpec.getBackWood())
      continue;
    if (searchSpec.getTopWood() != guitarSpec.getTopWood())
      continue;
    matchingGuitars.add(guitar);
  }
  return matchingGuitars;
}
```

Inventory.java

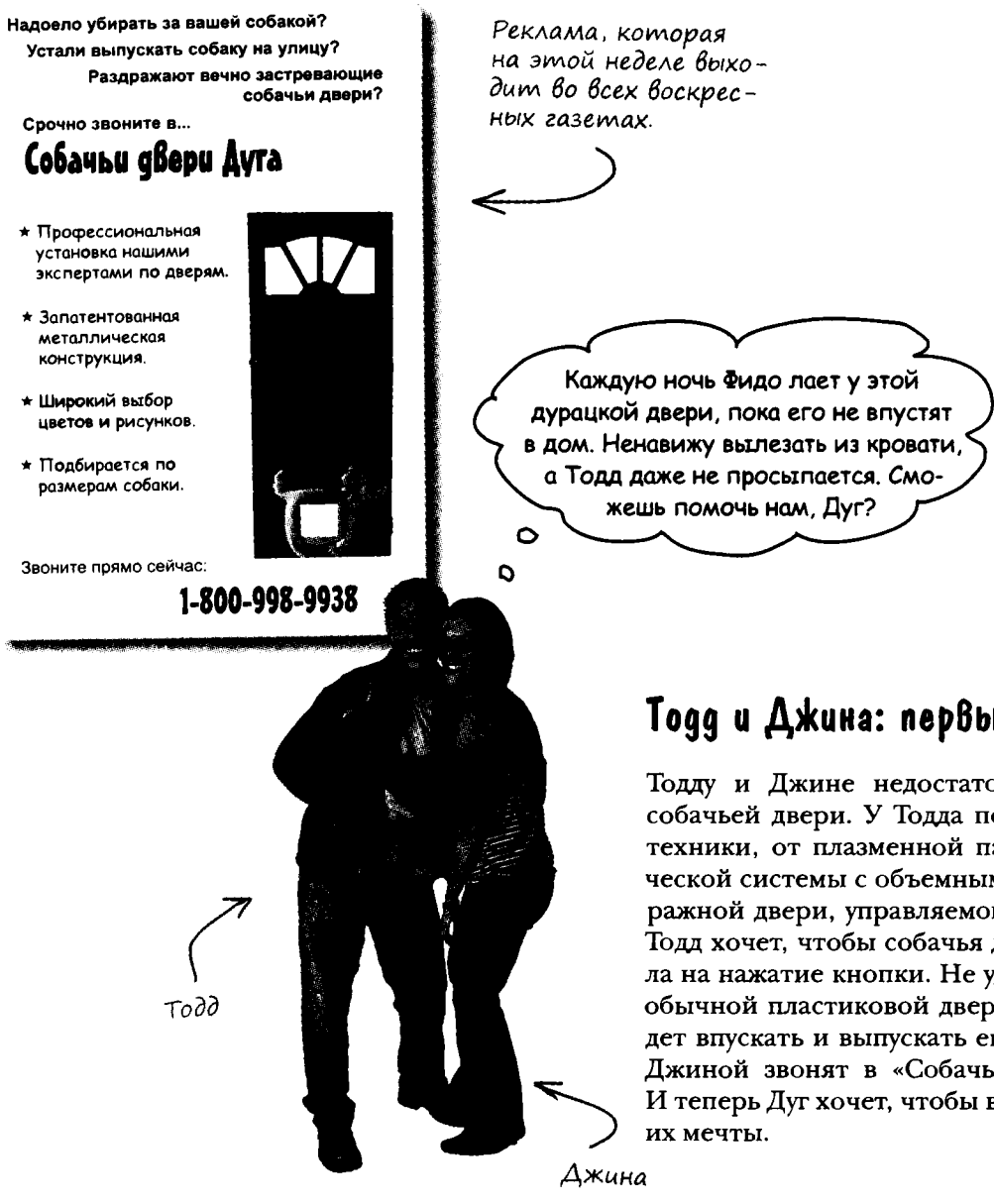
Подумайте: какие задачи решает класс Inventory? Выполнение операций с каталогом Рика? Тогда при чем тут сравнение двух объектов GuitarSpec? Сравнением объектов GuitarSpec должен заниматься класс GuitarSpec, а не класс Inventory.

Дайте им то, что они хотят

Хорошо, когда заказчик доволен. Вы ведь уже знаете, что написание хороших программ начинается с того, что вы проверяете, работает ли программа так, как хочет заказчик. Но как узнать, чего на самом деле хочет заказчик? И как убедиться в том, что заказчик вообще знает, чего он хочет? На помощь приходят хорошо составленные требования. В этой главе вы узнаете, как выполнить требования заказчика и предоставить ему именно то, чего он хотел. После этого на всех ваших проектах можно будет ставить «знак качества», а вы сделаете еще один большой шаг к заветной цели — написанию хороших программ... раз за разом.

Вам поручена новая задача

Вас взяли на работу ведущим программистом в начинающую фирму – «Собачьи двери Дуга». Дуг сейчас как раз работает над новой высокотехнологичной моделью. Он решил, что вам можно поручить написание программного обеспечения для перспективного продукта.



Тодд и Джина: первые заказчики

Тодду и Джине недостаточно «обычной» собачьей двери. У Тодда полный комплект техники, от плазменной панели до акустической системы с объемным звучанием и гаражной двери, управляемой дистанционно. Тодд хочет, чтобы собачья дверь реагировала на нажатие кнопки. Не удовлетворившись обычной пластиковой дверцей, которая будет впускать и выпускать его собаку, Тодд с Джиной звонят в «Собачьи двери Дуга»... И теперь Дуг хочет, чтобы вы сделали дверь их мечты.

Работа над дверью начинается

Прежде всего нам понадобится класс, представляющий собачью дверь. Назовем его **DogDoor** и добавим несколько простых методов:

```
public class DogDoor {

  private boolean open;

  public DogDoor() {
    this.open = false;
  }

  public void open() {
    System.out.println("The dog door opens.");
    open = true;
  }

  public void close() {
    System.out.println("The dog door closes.");
    open = false;
  }

  public boolean isOpen() {
    return open;
  }
}
```

Предполагается, что класс DogDoor будет взаимодействовать с домашним оборудованием Дуга.

Все очень просто: open() открывает дверь...

...а метод close() ее закрывает.

Этот метод возвращает текущее состояние двери (открыта или закрыта).

Весь этот код...

...идет в файл DogDoor.java...

DogDoor.java

...который будет управлять оборудованием собачьей двери Тодда и Джины.

Фидо надеется на вас... Не говоря уже о Тодде, Джине и вашем начальнике Дуге.

Развлечения с магнитами

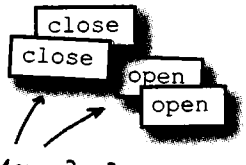
Теперь напишем другой класс Remote для пульта, который будет управлять дверью. Тодд и Джина будут использовать пульт, чтобы открывать дверь, не вставая с кровати.

Будьте внимательны... Некоторые магниты могут оказаться лишними.

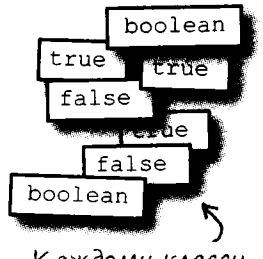
Да, это очень простое упражнение. Не расслабляйтесь, все только начинается.

```
public class Remote {

  private ___________ door;

  public Remote(_____________ _______) {
    this.door = door;
  }

  public void pressButton() {
    System.out.println("Pressing the remote control button...");
    if (_____._______()) {
      door._________();
    } else {
      door._________();
    }
  }
}
```

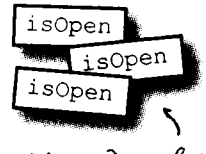
Сравните свой ответ с нашим, приведенным на с. 139.



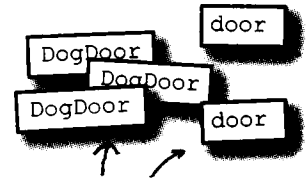
Методы для управления дверью.



Каждому классу необходимо немного логики, не так ли?



Метод проверяет, открыта или закрыта дверь.



Магниты для взаимодействия с объектом собачьей двери.

Пробный запуск

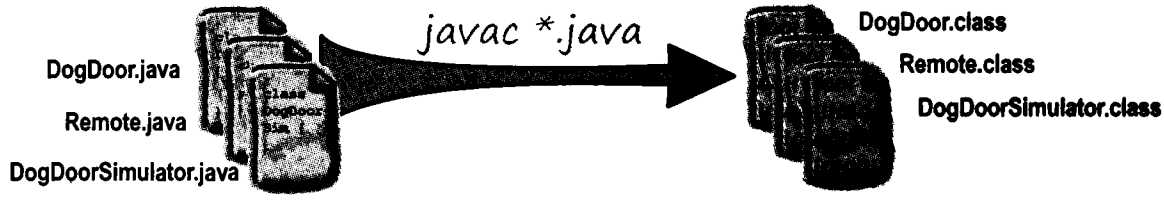
Проверим, что у нас получилось. Пора опробовать новую дверь в деле.

1 Создайте класс для тестирования двери (DogDoorSimulator.java).

```
public class DogDoorSimulator {
  public static void main(String[] args) {
    DogDoor door = new DogDoor();
    Remote remote = new Remote(door);
    System.out.println("Fido barks to go outside...");
    remote.pressButton();
    System.out.println("\nFido has gone outside...");
    remote.pressButton();
    System.out.println("\nFido's all done...");
    remote.pressButton();
    System.out.println("\nFido's back inside...");
    remote.pressButton();
  }
}
```

DogDoorSimulator.java

2 Откомпилируйте весь исходный код Java.



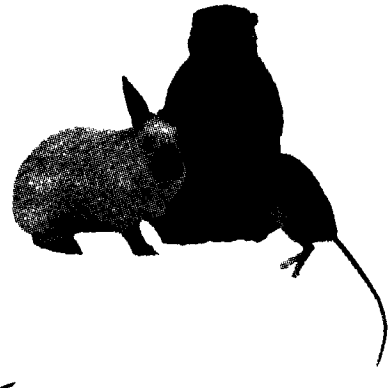
3 Выполните код!

```
File Edit Window Help Woof
%java DogDoorSimulator
Fido barks to go outside...
Pressing the remote control button...
The dog door opens.

Fido has gone outside...
Pressing the remote control button...
The dog door closes.

Fido's all done...
Pressing the remote control button...
The dog door opens.

Fido's back inside...
Pressing the remote control button...
The dog door closes.
```

Работает! Можно показывать Toggy и Джине...

Но когда попробовала Джина...

Ой... Действительно, дверь впустила Фидо, но не только его одного.



Возьми в руку карандаш

Как грызуны проникают в кухню Джины? Запишите внизу, каким недостатком, по вашему мнению, обладает текущая версия собачьей двери.

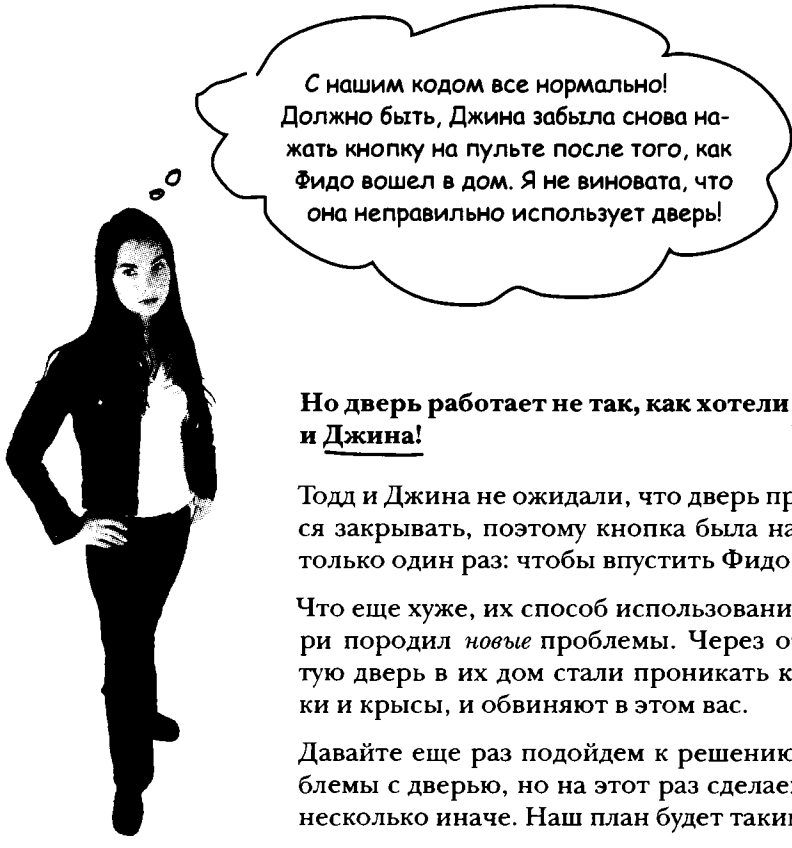
_____

_____

_____

_____

Не переходите к следующей странице, пока не напишете ответ к этому упражнению.



Но дверь работает не так, как хотели Тодд и Джина!

Тодд и Джина не ожидали, что дверь придется закрывать, поэтому кнопка была нажата только один раз: чтобы впустить Фидо.

Что еще хуже, их способ использования двери породил *новые* проблемы. Через открытую дверь в их дом стали проникать кролики и крысы, и обвиняют в этом вас.

Давайте еще раз подойдем к решению проблемы с дверью, но на этот раз сделаем это несколько иначе. Наш план будет таким:

1. **Собрать требования к двери.**
2. **Разобраться, как именно должна работать дверь.**
3. **Получить всю необходимую дополнительную информацию от Тодда и Джины.**
4. **Сделать дверь ПРАВИЛЬНО!**

Похоже, на этот раз нам придется побеседовать с Тоддом и Джиной более обстоятельно.

На этот раз мы уделяем куда больше внимания шагу 1, не так ли?

1. Убедитесь в том, что поведение программы соответствует требованиям заказчика.

Так что же именно называется требованием?

Обычно требование представляет собой описание одной операции. Эту операцию можно протестировать и по результатам проверить, выполняется ли требование.

Это конкретная операция, которую ваша система должна выполнять для правильной работы.

Система управления собачьей дверью должна выполнять различные операции: открывать, закрывать, выпускать Фидо, не пускать внутрь грызунов... Все, чего захотят Тодд и Джина, становится частью функциональности системы.

Под «системой» понимается приложение или проект, над которыми вы работаете. В нашем случае системой является полная конфигурация управления собачьей дверью (которая, кстати, включает пульт дистанционного управления).

Помните: правильность работы системы определяется заказчиком. Если какое-то требование не выполняется, даже если заказчик сам забыл сказать вам об этом, — значит, система работает неправильно!

За партой

Требование — отдельная формулировка, описывающая, как должны работать некий продукт или функция системы. Чаще всего используется в формальном смысле в ходе проектирования систем или программных продуктов.

Слушайте заказчика

В том, что касается требований, лучше всего **дать заказчику высказаться**. Обращайте внимание на то, *что* должна делать система; *как* она это будет делать, вы разберетесь позднее.

То, что говорят Тодд и Джина; ваша задача — преобразовать их слова в требования.

Фидо около 12 дюймов ростом, и мы не хотим, чтобы ему приходилось пригибаться, — это вредно для позвоночника.

Джина: И мы хотим, чтобы дверь автоматически закрывалась через несколько секунд. Не хочу вставать по ночам, чтобы закрывать дверь.

Вы: На пульте должна быть одна кнопка или отдельные кнопки «Открыть» и «Закрыть»?

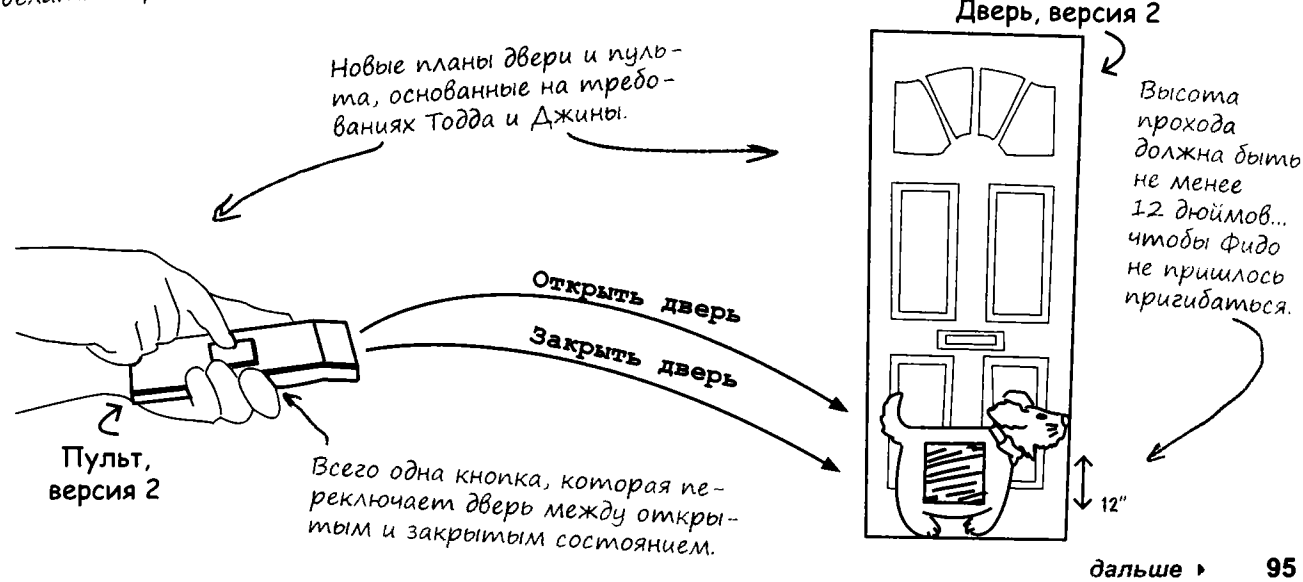
Тодд: Если дверь закрывается автоматически, то две кнопки не нужны, верно? Давайте ограничимся одной кнопкой.

Вы: Конечно. Значит, если дверь закрыта — кнопка открывает ее. А если дверь открыта, то кнопка закрывает ее (на тот случай, если дверь застряла).

И не думайте о коде на этой стадии — пока важно понять, что должна делать дверь.

Тодд: Замечательно. Джина, у тебя есть предложения?

Джина: Нет, по-моему, все учтено. Это будет собачья дверь нашей мечты.



Создание списка требований

Теперь, когда мы знаем, чего хотят Тодд и Джина, запишем новый набор требований. Ничего особо сложного...

Собачья дверь для Тодда и Джины, версия 2.0

Список требований

1. Высота дверного проема должна быть не менее 12 дюймов.
2. Кнопка на пульте открывает дверь, если она закрыта, и закрывает ее, если дверь открыта.
3. После того, как дверь откроется, она должна автоматически закрываться (если не была закрыта до этого).

Дверь просто закрывается через несколько секунд после того, как она открылась.

Простой список того, что, по мнению заказчика, должна делать система, которую вы создаете.

Сравните с замечаниями Тодда и Джины на с. 95... Видите, как слова превратились в набор требований?

Обязательно оставьте свободное место... В ходе работы над проектом почти всегда появляются новые требования.

Специальный приз

Теперь у вас есть список того, что нужно сделать для создания двери Тодда и Джины. Но это не все – теперь вы сможете четко объяснить своему начальнику, над чем вы сейчас работаете и что, по вашему мнению, осталось сделать для завершения проекта.

И какой прок от этого списка? Раньше Тодд и Джина уже забыли сказать, что дверь должна автоматически закрываться... А если они еще что-нибудь забудут?

Вы должны понимать, как будет использоваться собачья дверь.

Теперь вы знаете одну из самых сложных сторон сбора требований заказчика – иногда даже сам заказчик не знает, чего он хочет! Значит, прежде чем точно определить, как должна работать система, необходимо задать правильно сформулированные вопросы. После этого можно начать мыслить дальше описания, предоставленного заказчиком, и предвидеть возможные проблемы еще до их возникновения.

В нашем случае система состоит из двери и пульта.

Возьми в руку карандаш

Как вы думаете, о чем не подумали Тодд и Джина (в том, что касается их новой собачьей двери)? Что необходимо учесть, чтобы Тодд и Джина были довольны дверью, которую вы для них создаете?

Как на самом деле должна работать дверь?

Вы уже знаете, чего хотят от двери Тодд и Джина, однако ваша задача — проследить за тем, чтобы дверь действительно *работала*. Может быть, попутно вы откроете некоторые возможности, которые нужны Тодду и Джине, но о которых они не подумали.

Давайте запишем, что именно происходит, когда Фидо нужно выйти на улицу:

Список требований со страницы 96.

Собачья дверь для Тодда и Джины, версия 2.0
Список требований

1. Выс
мен
2. Кн
зак
та
3. По
ав
за

Новый список, в котором подробно расписана работа собачьей двери.

Собачья дверь для Тодда и Джины, версия 2.0
Как должна работать дверь

1. Фидо лает, чтобы его выпустили.
2. Тодд или Джина слышат лай.
3. Тодд или Джина нажимают knopky на пульте.
4. Дверь открывается.
5. Фидо выходит на улицу.
6. Фидо делает свои дела.
7. Фидо заходит в дом.
8. Дверь автоматически закрывается.

По этому описанию мы можем определить, не были ли упущены какие-либо требования.

После завершения шага 8 Фидо находится в доме, а Тодд и Джина счастливы.

Часто задаваемые вопросы

В: Значит, требование — это просто описание того, чего хочет от своего приложения заказчик?

О: Вообще-то требование — нечто намного большее, чем описание пожеланий заказчика (хотя они и являются хорошей отправной точкой). Для начала определите, чего хочет и ожидает ваш заказчик и что, по его мнению, должна делать система, которую вы для него пишете. Но это далеко не все...

Помните: большинство людей хотят, чтобы их программы работали даже при возникновении проблем. Следовательно, вы должны предвидеть, что может пойти не так, и добавить требования, которые будут справляться с этими проблемами. Хороший набор требований выходит за рамки того, что говорит заказчик, и обеспечивает работу системы даже в аномальных или непредвиденных обстоятельствах.

В: Система для Тодда и Джины — это просто собачья дверь, верно?

О: Система — это все необходимое для достижения целей заказчика. В случае собачьей двери система включает не только саму дверь, но и пульт дистанционного управления. Без пульта собачья дверь не будет полноценно работать.

И хотя Тодд, Джина и Фидо не являются частью системы, о них тоже нужно подумать в ходе проектирования. Получается, что ваши заботы вовсе не ограничиваются конкретной собачьей дверью.

В: Не понимаю, почему я должен думать о том, как Тодд и Джина будут использовать собачью дверь и что может пойти не так. Ведь это их проблемы, а не мои?

О: Помните первый шаг на пути к написанию хороших программ? Вы должны позаботиться о том, чтобы приложение работало так, как хочет заказчик, — даже если бы лично вы так его не использовали. Это означает, что вы должны действительно хорошо понять, что должна делать система и как ваши заказчики будут ее использовать.

Собственно, предоставить Тодду и Джине качественную, работающую дверь можно только при одном условии — если вы будете лучше понимать их систему и точно знать, что нужно для ее работы. В этом случае вы сможете предвидеть проблемы и решать их еще до того, как Тодд и Джина заметят какие-то неполадки.

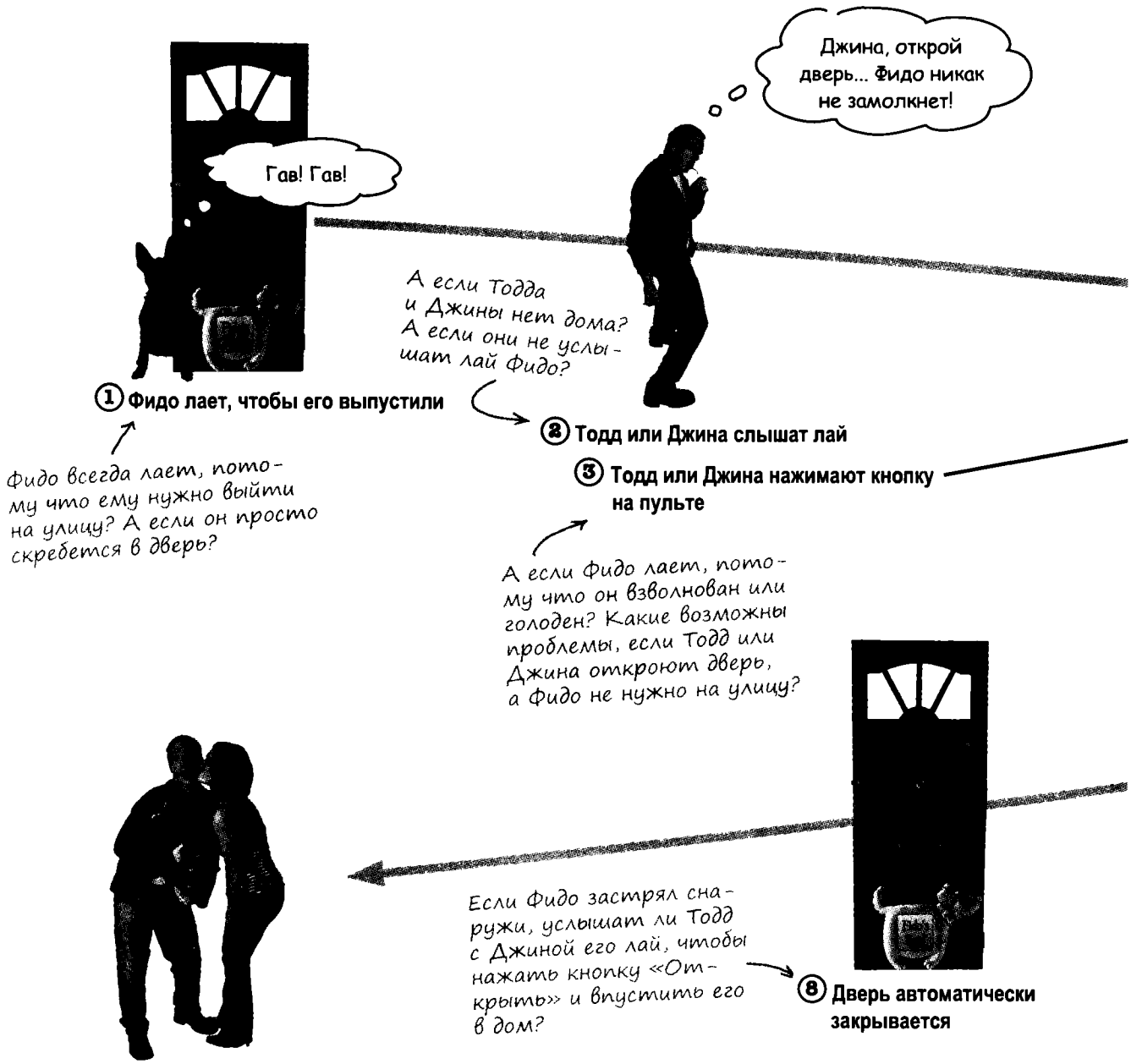
В: Значит, мне придется предугадать все плохое, что может случиться, когда Тодд и Джина используют свою дверь?

О: Вот именно! Более того, давайте прямо сейчас займемся этим...

Лучший способ получить хорошие требования — разобраться в том, что должна делать система.

Планирование возможных неудач

Приведенная ниже диаграмма показывает, как должна работать собачья дверь Тодда и Джины; все цифры соответствуют пунктам списка на с. 98. Но в жизни порой случаются непредвиденные ситуации. Мы записали некоторые потенциальные проблемы.



Сможете придумать что-нибудь еще? Вот и отлично... Чем больше потенциальных проблем вы найдете, тем менее хрупким будет ваше приложение. Запишите свои варианты непредвиденных событий прямо на диаграмме.

Должны ли мы думать о том, что произойдет, если дверь заклинит? Или это проблема оборудования?

Мне гораздо лучше!

④ Дверь автоматически открывается

⑤ Фидо выходит на улицу

А если Фидо останется внутри?

⑥ Фидо делает свои дела

Что произойдет, если дверь автоматически закроется к тому моменту, когда Фидо надумает вернуться?

⑦ Фидо заходит в дом

Альтернативные пути для решения проблем

Итак, мы нашли несколько моментов, в которых могут возникнуть ошибки. Пора обновить наш список, чтобы обеспечить нормальную работу двери. Давайте запишем, что произойдет, если дверь закроется до того, как Фидо вернется домой.

Список требований со страницы 96. Позднее его нужно будет обновить, но пока сойдет и так.

Собачья дверь для Тодда и Джины, версия 2.0

Список требований

1. Выс
   12 д
2. Кно
   та,
3. По
   то
   до

Собачья дверь для Тодда и Джины, версия 2.0

Как должна работать дверь

1. Фидо лает, чтобы его выпустили.
2. Тодд или Джина слышат лай.
3. Тодд или Джина нажимают кнопку на пульте.
4. Дверь открывается.
5. Фидо выходит на улицу.
6. Фидо делает свои дела.
   6.1 Дверь автоматически закрывается.
   6.2 Фидо лает, чтобы вернуться в дом.
   6.3 Тодд или Джина слышат лай (повторно).
   6.4 Тодд или Джина нажимают кнопку на пульте.
   6.5 Дверь открывается (повторно).
7. Фидо заходит в дом.
8. Дверь автоматически закрывается.

Нумерация второго уровня используется для обозначения «подпунктов», которые могут происходить на шаге 6.

Если Фидо остается снаружи, для его возвращения необходимо выполнить ряд дополнительных шагов. Эти дополнительные шаги называются альтернативным путем.

Все новые шаги решают проблему с закрытием двери до того, как Фидо вернулся в дом.

С добавлением новых шагов Фидо сможет попасть в дом, несмотря на то что Тодд и Джина не предвидели возникновения этой проблемы.

Погодите, мы же это проходили в колледже... А это случайно не **вариант использования**?

Да! Все это время мы создавали варианты использования.

Когда мы записываем последовательность шагов, которые позволяют Фидо выйти из дома, на самом деле мы записываем **вариант использования**.

Вариант использования (use case) — последовательность шагов, выполняемых системой с какой-либо целью. В случае Тодда и Джины эта цель проста: вывести Фидо из дома, а затем вернуть его обратно.

Смотрите! Да это же вариант использования!

Собачья дверь для Тодда и Джины, версия 2.0

Как должна работать дверь

1. Фидо лает, чтобы его выпустили.
2. Тодд или Джина слышат лай.
3. Тодд или Джина нажимают кнопку на пульте.
4. Дверь открывается.
5. Фидо выходит на улицу.
6. Фидо делает свои дела.
   - 6.1 Дверь автоматически закрывается.
   - 6.2 Фидо лает, чтобы вернуться в дом.
   - 6.3 Тодд или Джина слышат лай (повторно).
   - 6.4 Тодд или Джина нажимают кнопку на пульте.
   - 6.5 Дверь открывается (повторно).
7. Фидо заходит в дом.
8. Дверь автоматически закрывается.

Оказывается, мы уже написали вариант использования для собачьей двери Тодда и Джины.

(Снова) о вариантах использования

Мы пишем вариант использования уже почти 10 страниц. Но давайте разберемся, в чем же заключается суть этой последовательности действий — варианта использования собачьей двери Тодда и Джины.

Вариант использования описывает, что ваша система делает для достижения конкретной цели заказчика.

Варианты использования отвечают на вопрос «что». Что должна делать собачья дверь? Не забывайте, что «как» нас пока не интересует... Этим мы займемся чуть позже.

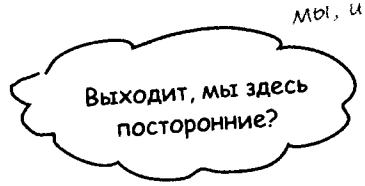
Наше внимание сосредоточено на том, что система должна «делать». Что должно произойти, чтобы Фидо мог выйти из дома (а затем вернуться обратно в дом)?

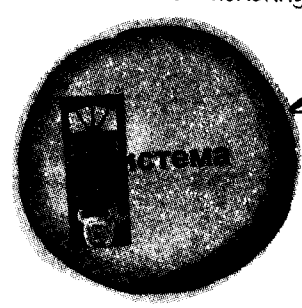
Один вариант использования предназначен для одной цели. Цель Тодда и Джины — выпустить Фидо из дома так, чтобы им не пришлось вставать с кровати.

Если Тодд и Джина вдруг захотят подсчитать, сколько раз Фидо проходит через дверь, это будет другая цель — и соответственно нам понадобится другой вариант использования.

Цель заказчика определяет смысл варианта использования: ради чего выполняются все эти шаги? Мы действуем в интересах заказчика, помните? Система должна помочь заказчику реализовать поставленную цель.

Пользователь (или пользователи) находится вне системы, он не является ее частью. Фидо использует систему, и он находится вне системы; Джина поставила цель системы, и она тоже находится вне ее.





Дверь и пульт являются частью системы, то есть находятся в системе.

Вариант использования подробно описывает, что должна делать дверь, когда Фидо захочет выйти на улицу.

Собачья дверь для Тодда и Джины, версия 2.0
Как должна работать дверь

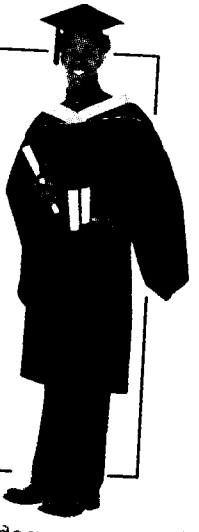
1. Фидо лает, чтобы его выпустили.
2. Тодд или Джина слышат лай.
3. Тодд или Джина нажимают кнопку на пульте.
4. Дверь открывается.
5. Фидо выходит на улицу.
6. Фидо делает свои дела.
   6.1 Дверь автоматически закрывается.
   6.2 Фидо лает, чтобы вернуться в дом.
   6.3 Тодд или Джина слышат лай (повторно).
   6.4 Тодд или Джина нажимают кнопку на пульте.
   6.5 Дверь открывается (повторно).
7. Фидо заходит в дом.
8. Дверь автоматически закрывается.

Вариант использования завершается с достижением цели заказчика — Фидо вернулся в дом, а Тодду и Джине не пришлось вставать с кровати.

Это альтернативный путь, но он направлен на достижение той же цели, что и в основном пути, поэтому он является частью того же варианта использования.

За партой

Вариант использования. Метод представления потенциальных требований новой системы или обновленной версии программного продукта. Каждый вариант использования содержит одно или несколько описаний взаимодействия системы с пользователем или другой системой для достижения поставленной цели.



Один вариант, три части

Хороший вариант использования состоит из трех частей. Если вы хотите, чтобы работа была успешно выполнена, вам понадобятся все три части.

1

Очевидная ценность

Super Buy

Каждый вариант использования должен иметь очевидную **ценность** для системы. Если вариант использования не помогает заказчику добиться его цели, то пользы от такого варианта будет не много.

Вариант использования должен помогать Тодду и Джине решить проблему с Фидо.

Вариант использования запускается, когда Фидо начинает лаять... и завершается, когда Фидо возвращается в дом, сделав все свои дела.

Начало и конец

STOP

2

Каждый вариант использования должен иметь четко определенную **начальную** и **конечную точки**. Некоторое событие должно инициировать процесс, а потом выполнение некоторого условия должно указать на его завершение.

Внешний инициатор

3

Каждый вариант использования запускается внешним инициатором, находящимся вне системы. Иногда инициатором может быть человек, но в общем случае это может быть любой фактор за пределами системы.

В варианте с собачьей дверью Фидо является внешним инициатором. Это он запускает весь процесс.

Развлечения с магнитами

Ниже приведен вариант использования и магниты для всех трех составляющих частей хорошего варианта использования (в одной части — «Начало и конец» — используются два магнита). Ваша задача — определить, где должен находиться каждый из магнитов, и разместить его в правильной части варианта использования.

Подсказка: С одной из частей все должно быть очень просто... если посмотреть на картинку.

Собачья дверь для Тодда и Джины, версия 2.0

Как должна работать дверь

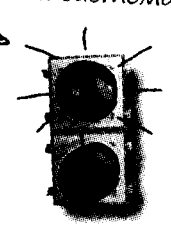
1. Фидо лает, чтобы его выпустили.
2. Тодд или Джина слышат лай.
3. Тодд или Джина нажимают кнопку на пульте.
4. Дверь открывается.
5. Фидо выходит на улицу.
6. Фидо делает свои дела.
   - 6.1 Дверь автоматически закрывается.
   - 6.2 Фидо лает, чтобы вернуться в дом.
   - 6.3 Тодд или Джина слышат лай (повторно).
   - 6.4 Тодд или Джина нажимают кнопку на пульте.
   - 6.5 Дверь открывается (повторно).
7. Фидо заходит в дом.
8. Дверь автоматически закрывается.

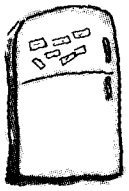
Поместите этот магнит на ту часть варианта, которая обладает очевидной ценностью для Тодда и Джины.

Поместите этот магнит на условие варианта, которое определяет, что процесс должен остановиться.

Кто запускает работу варианта использования?

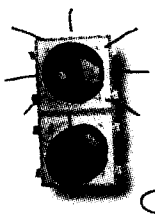
Что прекращает выполнение варианта? (Обычно это некоторое действие за пределами системы).





Развлечения с магнитами. Решение

Ниже приведен вариант использования и магниты для всех трех составляющих частей хорошего варианта использования. Ваша задача — определить, где должен находиться каждый из магнитов, и разместить его в правильной части варианта использования.



Начало варианта использования. Ничего не происходит, пока Фидо не залает.

Собачья дверь для Тодда и Джины, версия 2.0

Как должна работать дверь

1. Фидо лает, чтобы его выпустили.
2. Тодд или Джина слышат лай.
3. Тодд или Джина нажимают кнопку на пульте.
4. Дверь открывается.
5. Фидо выходит на улицу.
6. Фидо делает свои дела.
   - 6.1 Дверь автоматически закрывается.
   - 6.2 Фидо лает, чтобы вернуться в дом.
   - 6.3 Тодд или Джина слышат лай (повторно).
   - 6.4 Тодд или Джина нажимают кнопку на пульте.
   - 6.5 Дверь открывается (повторно).
7. Фидо заходит в дом.
8. Дверь автоматически закрывается.

Фидо в данном варианте использования является внешним инициатором.

Ценность варианта несомненна — Тодд и Джина могут оставаться в кровати, а Фидо выходит наружу.

Условие остановки... Фидо вернулся в дом, дверь закрыта.

Часто задаваемые вопросы

В: Итак, варианты использования — это просто последовательность шагов, которые должны быть выполнены правильно работающей системой?

О: В большинстве случаев — да. Но помните: одно из ключевых свойств варианта использования заключается в том, что он сосредоточен на достижении одной конкретной цели. Если система выполняет несколько операций — скажем, выпускает Фидо и отслеживает количество выходов за день, — вам потребуется несколько вариантов.

В: Выходит, моя система должна иметь вариант использования для каждой цели, верно?

О: Точно! Если система решает всего одну задачу, вероятно, одного варианта будет достаточно. Если же она решает 10 или 15 задач, потребуется гораздо больше вариантов использования.

В: А вариант использования — то, что делает система для достижения этой цели?

О: Именно так. Если записать то, что должна сделать система для выполнения операции, вероятно, у вас получится вариант использования.

В: Но вариант использования получился недостаточно конкретным. Почему в нем не упоминается класс Remote или класс DogDoor?

О: Варианты использования помогают проектировщику понять, что должна делать система, — и часто — объяснить ее смысл другим (скажем, заказчику или начальству). Если вариант использования сосредоточится на конкретных подробностях уровня программного кода, он будет бесполезен для всех, кроме программиста. Как правило, варианты использования должны формулироваться простым повседневным языком. Если вариант содержит большое количество терминов или технического жаргона, скорее всего, вариант стал слишком конкретным, что снижает пользу от его применения.

В: Вариант использования — то же, что диаграмма вариантов использования?

О: Нет, вариант использования обычно представляет собой последовательность шагов (хотя их можно записать и другим способом, но об этом мы поговорим в Приложении). Диаграммы вариантов использования — способ визуального представления вариантов использования, но мы уже поработали над диаграммой, описывающей работу системы (если забыли — обращайтесь к с. 101). Диаграммы вариантов использования будут рассматриваться в главе 6.

В: Как мой вариант использования будет преобразован в программный код?

О: Это другой этап процесса написания приложения. Более того, вскоре мы разберемся, как написать конкретный код на базе варианта использования для Тодда и Джины, — это произойдет буквально через несколько страниц.

Но вариант использования не должен подробно описывать, как вы будете писать свой код. Впрочем, это не мешает вам подумать заранее о том, как вариант использования будет преобразовываться в код.

В: Если вариант использования не помогает мне в написании кода, тогда какой от него прок? Зачем попусту тратить время?

О: Варианты использования помогают писать код — просто они не содержат подробностей программного уровня. Например, если бы мы не написали вариант использования для Тодда и Джины, то мы бы не поняли, что Фидо может застрять снаружи или что дверь должна закрываться автоматически. Все это стало ясно благодаря варианту.

Помните: невозможно написать хорошую программу, если ваше приложение не делает то, что нужно заказчику. Варианты использования помогают разобраться в том, что же ему нужно, — и только после этого можно переходить к написанию кода, который реализует систему, описанную вариантом использования.

Проверка требований по варианту использования

Итак, сейчас у нас имеется исходный набор требований и хороший вариант использования. Но мы должны *вернуться* к требованиям и убедиться в том, что они описывают все, что должна делать система. И как раз в этом нам поможет вариант использования:

Собачья дверь для Тодда и Джины, версия 2.0

Список требований

1. Высота дверного проема должна быть не менее 12 дюймов.
2. Кнопка на пульте открывает дверь, если она закрыта, и закрывает ее, если дверь открыта.
3. После того, как дверь откроется, она должна автоматически закрываться (если не была закрыта до этого).

Список требований, полученный от Тодда и Джины...

...а это то, что должна делать дверь.

Собачья дверь для Тодда и Джины, версия 2.0

Как должна работать дверь

1. Фидо лает, чтобы его выпустили.
2. Тодд или Джина слышат лай.
3. Тодд или Джина нажимают кнопку на пульте.
4. Дверь открывается.
5. Фидо выходит на улицу.
6. Фидо делает свои дела.
   - 6.1 Дверь автоматически закрывается.
   - 6.2 Фидо лает, чтобы вернуться в дом.
   - 6.3 Тодд или Джина слышат лай (повторно).
   - 6.4 Тодд или Джина нажимают кнопку на пульте.
   - 6.5 Дверь открывается (повторно).
7. Фидо заходит в дом.
8. Дверь автоматически закрывается.

Все ли на месте?

Теперь нужно просмотреть вариант использования и проверить, включено ли в требования все, что должна делать система.

Возьми в руку карандаш

Все ли учтено в требованиях?

Слева перечислено все, что должна делать дверь (прямо из варианта использования на с. 110). Ваша задача — идентифицировать требование для каждого шага варианта использования и записать его номер в поле рядом с шагом варианта. Если какой-либо шаг варианта не имеет соответствующих требований, поставьте прочерк.

Три наших требования... Каждое из них может использоваться для любого шага варианта использования.

Собачья дверь для Тодда и Джины, версия 2.0
Как должна работать дверь

1. Фидо лает, чтобы его выпустили. ____
2. Тодд или Джина слышат лай. ____
3. Тодд или Джина нажимают кнопку на пульте. ____
4. Дверь открывается. ____
5. Фидо выходит на улицу. ____
6. Фидо делает свои дела. ____
   - 6.1 Дверь автоматически закрывается. ____
   - 6.2 Фидо лает, чтобы вернуться в дом. ____
   - 6.3 Тодд или Джина слышат лай (повторно). ____
   - 6.4 Тодд или Джина нажимают кнопку на пульте. ____
   - 6.5 Дверь открывается (повторно). ____
7. Фидо заходит в дом. ____
8. Дверь автоматически закрывается. ____

Поставьте в каждом поле 1, 2, 3 или прочерк.

Собачья дверь для Тодда и Джины, версия 2.0
Список требований

1. Высота дверного проема должна быть не менее 12 дюймов.
2. Кнопка на пульте открывает дверь, если она закрыта, и закрывает ее, если дверь открыта.
3. После того, как дверь откроется, она должна автоматически закрываться (если не была закрыта до этого).

А вы нашли в этом варианте использования шаги, которые, как вам кажется, не имеют соответствующего требования? Если вы думаете, что в список нужно включить дополнительные требования, запишите их внизу:

_____

_____

_____

Возьми в руку карандаш
Решение

Все ли учтено в требованиях?

Слева перечислено все, что должна делать дверь (прямо из варианта использования на с. 110). Ваша задача — идентифицировать требование для каждого шага варианта использования и записать его номер в поле рядом с шагом варианта. Если какой-либо шаг варианта не имеет соответствующих требований, поставьте прочерк.

Собачья дверь для Тодда и Джины, версия 2.0

Как должна работать дверь

1. Фидо лает, чтобы его выпустили.	-
2. Тодд или Джина слышат лай.	-
3. Тодд или Джина нажимают кнопку на пульте.	2
4. Дверь открывается.	2
5. Фидо выходит на улицу.	1
6. Фидо делает свои дела.	-
6.1 Дверь автоматически закрывается.	3
6.2 Фидо лает, чтобы вернуться в дом.	-
6.3 Тодд или Джина слышат лай (повторно).	-
6.4 Тодд или Джина нажимают кнопку на пульте.	2
6.5 Дверь открывается (повторно).	2
7. Фидо заходит в дом.	1
8. Дверь автоматически закрывается.	3

Многое из того, что происходит с системой, не связано ни с какими требованиями.

Здесь можно поставить прочерк, потому что реализовывать нажатие кнопки нам не придется... Впрочем, пункт 2 тоже подходит, потому что нажать кнопку без пульта не получится.

Понятно, почему? Фидо не сможет выйти наружу, если проем будет слишком мал.

Если разобраться с требованиями главного пути, с альтернативным путем будет уже проще.

А вы нашли в этом варианте использования шаги, которые, как вам кажется, не имеют соответствующего требования? Если вы думаете, что в список нужно включить дополнительные требования, запишите их внизу:

Нет, наши требования описывают все, что должна делать система. Осталось просто написать код реализации этих требований, так?

Ну теперь-то можно заняться написанием кода?

Имея под рукой вариант использования и требования, мы можем написать код, которым, как мы **знаем**, Тодд и Джина останутся довольны. Давайте еще раз обратимся к требованиям и посмотрим, для чего же именно нужно будет написать код:

Этим будет заниматься Дуг и специалисты по оборудованию... Для этого требования код писать не нужно.

Собачья дверь для Тодда и Джины, версия 2.0.
Список требований

1. **Высота дверного проема должна быть не менее 12 дюймов.**
2. **Кнопка на пульте открывает дверь, если она закрыта, и закрывает ее, если дверь открыта.**
3. **После того, как дверь откроется, она должна автоматически закрываться (если не была закрыта до этого).**

Код для реализации этого требования уже имеется.

То, что добавили Тодд и Джина после разговора.... Мы должны написать код для автоматического закрывания двери.

Автоматическое закрывание двери

Remote.java

Осталось запрограммировать всего одно требование: дверь должна автоматически закрываться через какое-то время после открытия. Давайте вернемся к классу **Remote** и сделаем это.

```
import java.util.Timer;
import java.util.TimerTask;

public class Remote {

  private DogDoor door;

  public Remote(DogDoor door) {
    this.door = door;
  }

  public void pressButton() {
    System.out.println("Pressing the remote control button...");
    if (door.isOpen()) {
      door.close();
    } else {
      door.open();

      final Timer timer = new Timer();
      timer.schedule(new TimerTask() {
        public void run() {
          door.close();
          timer.cancel();
        }
      }, 5000);
    }
  }
}
```

Эти две директивы import необходимы для использования классов таймеров Java.

Проверяет состояние двери перед открытием или закрытием.

Класс Remote уже содержит код закрытия двери в том случае, если она открыта.

Создание нового таймера для отложенного закрытия двери.

Эта задача закрывает дверь, а потом отключает таймер.

Здесь мы указываем, сколько следует ждать перед выполнением задачи... В нашем случае продолжительность ожидания равна 5 секундам, что соответствует 5000 миллисекунд.

Часто задаваемые вопросы

В: А зачем нам класс Timer? Почему бы не использовать для закрытия двери программные потоки Java?

О: В использовании **Thread** для закрывания двери нет ничего плохого. Собственно, класс **Timer** именно так и работает: он запускает фоновый объект **Thread**. Но класс **Timer** упрощает запланированное выполнение задачи, поэтому его использование в классе **Remote** выглядит логично.

В: Почему переменная timer объявлена с ключевым словом final?

О: Потому что нам понадобится вызвать ее метод **cancel()** в анонимном классе **TimerTask**. Если вам потребуется обращаться к переменным внешнего класса (**Remote** в нашем случае) из анонимного класса, эти переменные должны быть объявлены с ключевым словом final.

В: Зачем вызывать cancel()? Разве таймер сам не остановится после выполнения TimerTask?

О: Остановится, но в большинстве JVM до уничтожения **Timer** уборщиком мусора пройдет целая вечность. В итоге ваш код проработает несколько часов, прежде чем будет корректно завершен. Это нехорошо, а ручной вызов **cancel()** решает проблему.

Нужен новый тест!

Старый тест уже не годится... Он предполагает, что Тодд и Джина закрывают дверь вручную, а не по таймеру. Давайте обновим наш тест, чтобы в нем использовался обновленный класс **Remote**:

DogDoorSimulator.java

```
public class DogDoorSimulator {

  public static void main(String[] args) {
    DogDoor door = new DogDoor();
    Remote remote = new Remote(door);

    System.out.println("Fido barks to go outside...");
    remote.pressButton();

    System.out.println("\nFido has gone outside...");
    remote.pressButton();

    System.out.println("\nFido's all done...");
    remote.pressButton();

    System.out.println("\nFido's back inside...");
    remote.pressButton();
  }
}
```

То же, что в более ранней версии, но нажатие кнопки открывает дверь и запускает таймер закрытия двери.

В новой, улучшенной версии Джине не нужно нажимать кнопку, чтобы закрыть дверь. Это теперь происходит автоматически.

Так как дверь закрывается по таймеру, у Фидо достаточно времени, чтобы вернуться обратно. Джине не нужно открывать дверь, чтобы впустить Фидо в дом.

Еще одно место, в котором можно удалить часть кода... дверь закрывается автоматически.

В: Что-то я запутался в коде таймера: что здесь происходит? Объясните еще раз.

О: Ничего страшного... Сейчас необязательно вникать во все тонкости Java. Принципиально то, что наш вариант использования помог написать хорошие требования, а требования помогли понять, как написать программу для правильно работающей двери. И это намного важнее того, как — и даже на каком языке — написан код управления дверью.

В: Итак, новый тест проверяет главный путь нашего варианта использования, верно?

О: Да, верно. Снова откройте с. 115 и вспомните, как работает дверь... Именно это поведение тестирует новый класс **DogDoorSimulator**. Мы хотим убедиться в том, что новая дверь Тодда и Джины работает именно так, как они хотели.

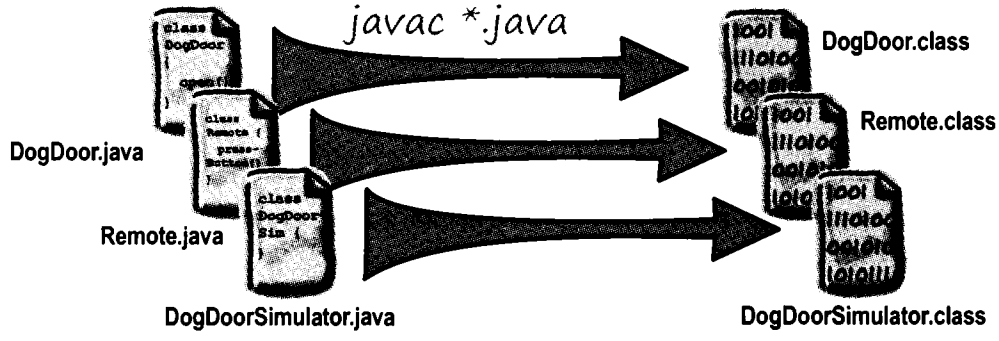
В: А почему мы не проверяем альтернативный путь?

О: Очень хороший вопрос. Давайте протестируем эту версию двери, а потом поговорим на эту тему подробнее...

Пробный запуск, Версия 2.0

Пора посмотреть, окупились ли все наши старания.
Протестируем новую, улучшенную собачью дверь.

1 Откомпилируйте весь исходный код Java в классы.



2 Запустите код!

```
File Edit Window Help InAndOut
%java DogDoorSimulator
Fido barks to go outside...
Pressing the remote control button...
The dog door opens.
Fido has gone outside...
Fido's all done...
Fido's back inside...
```

Тик!

Так!

```
File Edit Window Help TimeIsAThief
%java DogDoorSimulator
Fido barks to go outside...
Pressing the remote control button...
The dog door opens.
Fido has gone outside...
Fido's all done...
Fido's back inside...
The dog door closes.
```

Тик!

Так!

Проходит несколько секунд между открытием двери...

...и ее закрытием.

Работает! Покажем Тодду и Джине...

Ваша система должна работать в реальном мире...

Не думаю, что программу можно показывать Тодду и Джине... Как насчет альтернативного пути, когда Фидо остается снаружи, а дверь закрывается за ним?

Верное замечание... протестировать нужно не только основной путь, но и альтернативные пути.

Разве не замечательно, когда программы с первого раза работают так, как положено? К сожалению, в реальном мире так почти никогда не бывает. Прежде чем показывать новую дверь Тодду и Джине, необходимо проверить работоспособность двери в том случае, когда Фидо *не возвращается* домой после выгула.

...поэтому вы должны планировать и тестировать возможные неполадки.

МОЗГОВОЙ ШТУРМ

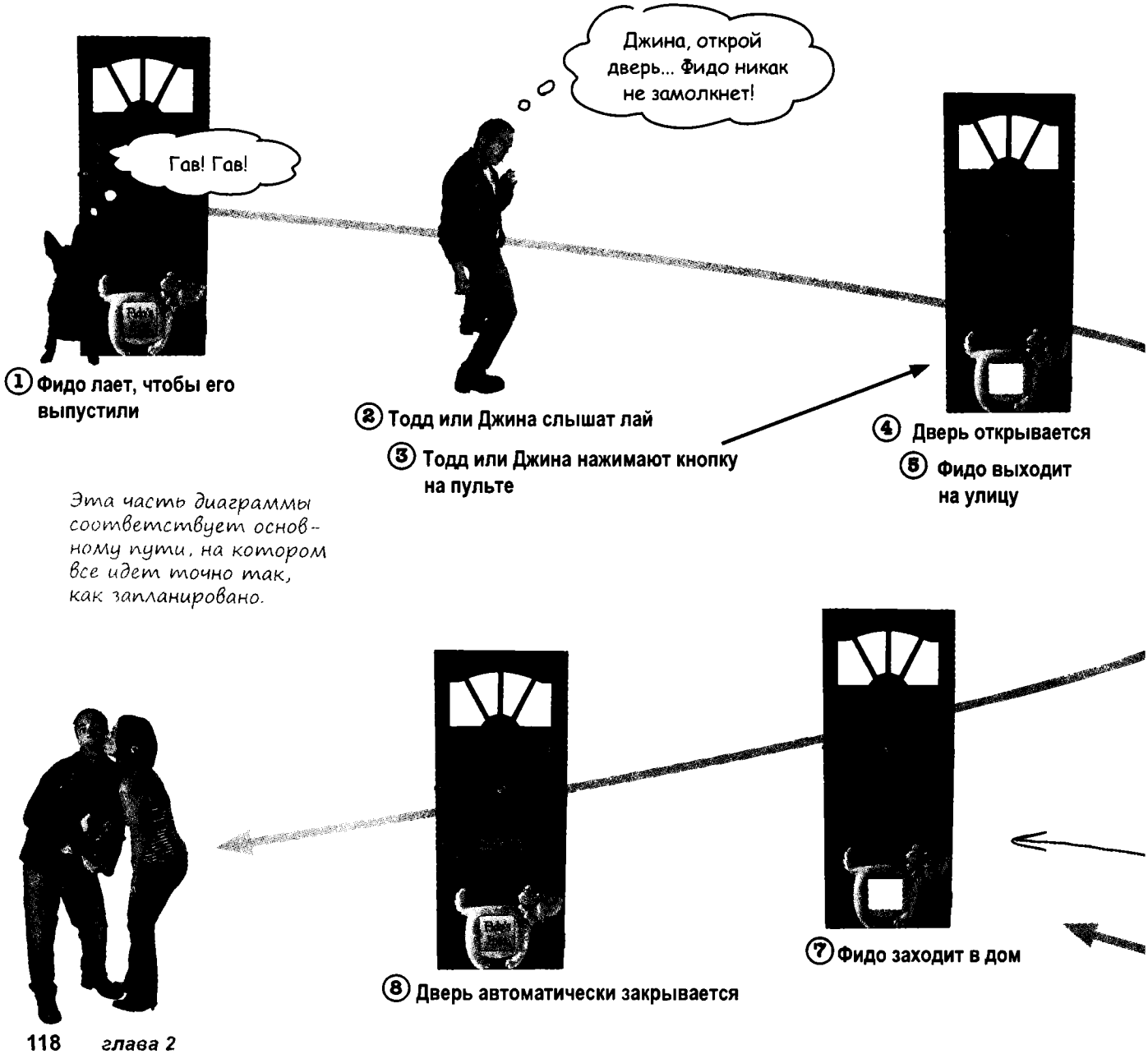
Как бы вы изменили класс `DogDoorSimulator`, чтобы протестировать возможную задержку Фидо на улице?

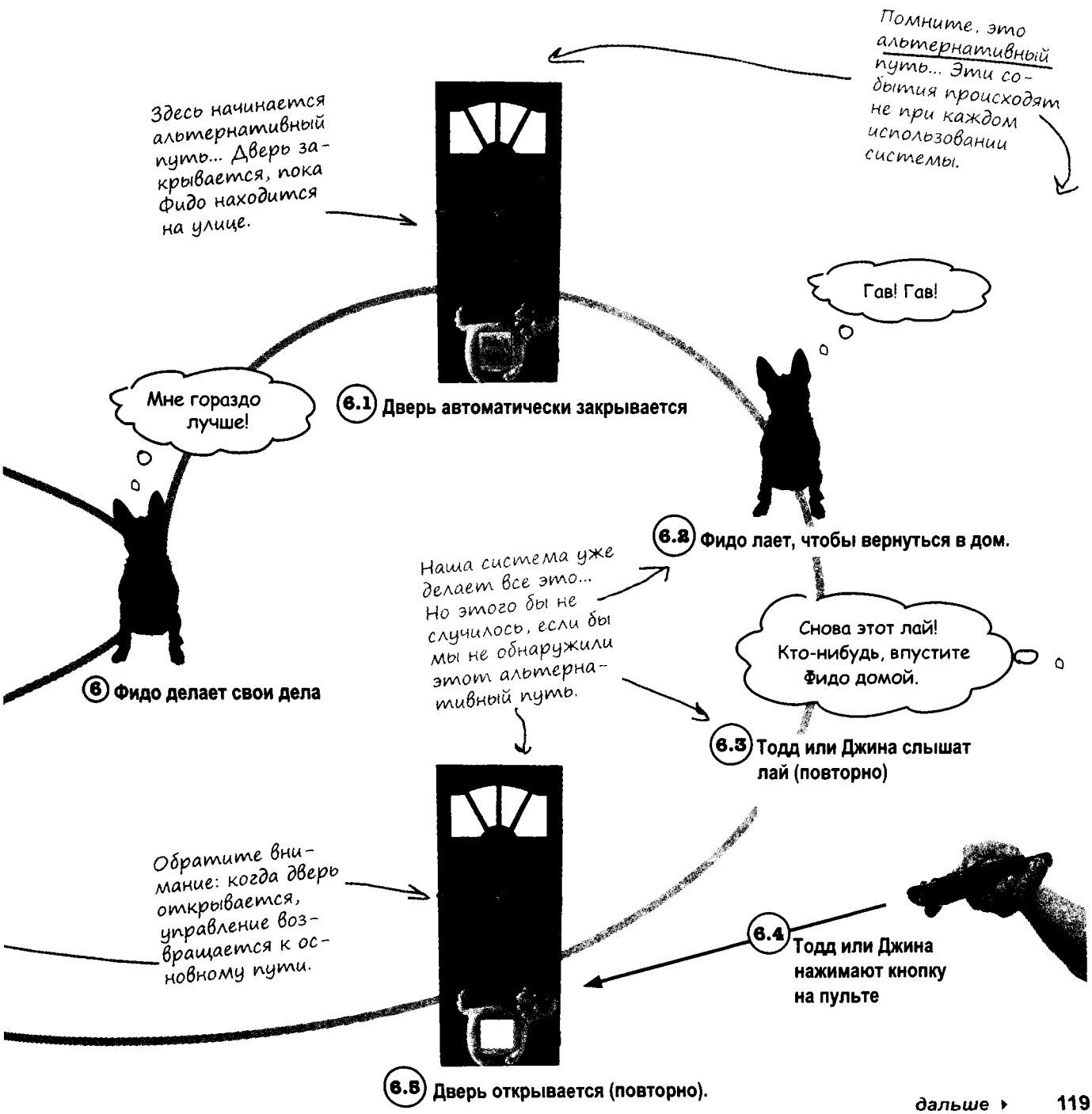
МОЗГОВОЙ ШТУРМ

Можете ли вы предложить еще хотя бы один альтернативный путь для собачьей двери Тодда и Джины? Запишите вариант использования и обновите список требований для своего альтернативного пути.

Анализ альтернативного пути

Давайте убедимся в том, что мы хорошо понимаем, что происходит на альтернативном пути, и можем обновить **DogDoorSimulator** для тестирования нового пути. Ниже приведен исходный основной путь со страницы 118 вместе с альтернативным путем, который мы определили и включили в вариант использования:




Помните, это альтернативный путь... События происходят не при каждом использовании системы.
Здесь начинается альтернативный путь... Дверь закрывается, пока Фидо находится на улице.
Гав! Гав!
Мне гораздо лучше!
6.1 Дверь автоматически закрывается
6.2 Фидо лает, чтобы вернуться в дом.
Наша система уже делает все это... Но этого бы не случилось, если бы мы не обнаружили этот альтернативный путь.
Снова этот лай! Кто-нибудь, впустите Фидо домой.
6 Фидо делает свои дела
6.3 Тодд или Джина слышат лай (повторно)
Обратите внимание: когда дверь открывается, управление возвращается к основному пути.
6.4 Тодд или Джина нажимают кнопку на пульте
6.5 Дверь открывается (повторно).

Развлечения с магнитами

DogDoorSimulator.java

Пора обновить тестовую программу, причем на этот раз вам наконец-то придется заняться написанием кода. Ниже приведена наша версия DogDoorSimulator.

Расставьте магниты из нижней части страницы на правильные места в коде. Если у вас возникнут трудности, обратитесь к диаграмме на предыдущей странице — на ней показано, что происходит на каждом шаге программы. Да, и чтобы было интереснее... все магниты с точками, точками с запятой и круглыми скобками упали с холодильника. Разместите их там, где они нужны.

```
public class DogDoorSimulator {

  public static void main(String[] args) {
    DogDoor door = new DogDoor();
    Remote remote = new Remote(door);

    ____________________________________
    ____________________________________
    System.out.println("\nFido has gone outside...");
    System.out.println("\nFido's all done...");

    try {
      Thread.currentThread().__________(10000);
    } catch (InterruptedException e) { }

    ____________________________________
    ____________________________________
    ____________________________________
    ____________________________________
    System.out.println("\nFido's back inside...");
  }
}
```

Начало альтернативного пути.

Программа должна какое-то время ожидать, а потом автоматически закрывать дверь.

Здесь альтернативный путь снова возвращается к основной линии.

System.out.println
System.out.println
System.out.println
System.out.println
System.out.println
System.out.println

pressButton
pressButton
pressButton
pressButton

remote
remote
remote
remote

waitFor
wait
sleep

Методы использования пульта.

Методы, вызываемые для программного потока Java.

Разные выводимые сообщения.

"Fido scratches at the door."
"\nFido starts barking..."
"...but he's stuck outside!"
"...so Todd grabs the remote control."
"...but he's stuck inside!"
"...so Gina grabs the remote control."

Пробный запуск, версия 2.1

Внесите изменения в свою копию **DogDoorSimulator.java**, затем перекомпилируйте класс. Теперь все готово к тестированию альтернативного пути нашего варианта использования.

```
File Edit Window Help InLikeFlynn
%java DogDoorSimulator
Fido barks to go outside...
Pressing the remote control butt
The dog door opens.

Fido has gone outside...

Fido's all done...
```

Тик!

Так!

Дверь открывается, Фидо выходит наружу.

```
Edit Window Help TheOutsiders
va DogDoorSimulator
barks to go outside...
ssing the remote control button...
e dog door opens.

Fido has gone outside...

Fido's all done...
The dog door closes.
...but he's stuck outside!

Fido starts barking...
...so Gina grabs the remote control.
Pressing the remote control button...
The dog door opens.

Fido's back inside...
```

Но Фидо начинает гоняться за жуками, и дверь закрывается, пока он еще не вернулся в дом.

Фидо лает, чтобы его пустили в дом, и Джина нажимает кнопку на пульте...

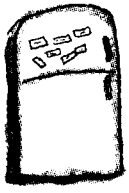
...и Фидо возвращается в уютный дом с кондиционером.

```
File Edit Window Help ThereAndBackAgain
%java DogDoorSimulator
Fido barks to go outside...
Pressing the remote control butt
The dog door opens.

Fido has gone outside...

Fido's all done...
The dog door closes.
...but he's stuck outside!

Fido starts barking...
...so Gina grabs the remote control.
Pressing the remote control button...
The dog door opens.

Fido's back inside...
The dog door closes.
```

Тик!

Так!

Вскоре дверь снова закрывается, а кролики, грызуны и насекомые остаются снаружи.



Развлечения с магнитами. Решение

Вот как выглядит наша тестовая программа. Убедитесь в том, что ваш ответ не отличается от нее.

DogDoorSimulator.java

```
public class DogDoorSimulator {

  public static void main(String[] args) {
    DogDoor door = new DogDoor();
    Remote remote = new Remote(door);

    System.out.println( ) ;
    remote . pressButton ( ) ;
    System.out.println("\nFido has gone outside...");
    System.out.println("\nFido's all done...");

    try {
      Thread.currentThread().sleep(10000);
    } catch (InterruptedException e) { }

    System.out.prin( ( but he's stuck outsid ) ) ;
    System.out.println ido starts barking..." ;
    System.out.println( ..so Gina grabs the remote control." ) ;
    remote . pressButton ( ) ;
    System.out.println("\nFido's back inside...");
  }
}
```

Точки, точки с запятой и круглые скобки расставляются там, где они необходимы.

Мы могли выбрать сообщение, в котором пульт берет Тодд... Но мы же договорились, что тестируем реальную ситуацию, помните? Конечно, основная работа достанется на долю Джины.

Новая дверь установлена

Хорошие варианты использования, требования, основные пути, альтернативные пути, работоспособная тестовая программа – мы определенно находимся на правильном пути к написанию хорошей программы. Пора доставить новую дверь Тодду и Джине.

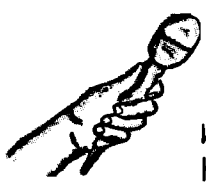
Тодд и Джина могут спать спокойно — а значит, заказчики довольны.

Фидо в доме, а кролики, мыши и сурки остались снаружи.

Тот самый результат, на который мы надеялись еще на с. 92. Видите, насколько полезны хорошо сформулированные требования?

Приложение работает, заказчики довольны

Мы не только выполнили требования Тодда и Джины, но и позаботились о том, чтобы дверь работала в изначально непредвиденной ситуации – например, когда Фидо остается на улице.



ОТКРОВЕННО О ВАРИАНТАХ

Интервью недели: Как найти правильный путь

HeadFirst: Привет, Основной Путь.

Счастливый Путь: Вообще-то я предпочитаю, чтобы меня называли «Счастливым путем». Знаю, во многих книгах используется термин «Основной Путь», но людям проще запомнить, кто я такой, когда меня называют «Счастливым Путем».

HeadFirst: Прошу прощения. Как бы то ни было, мы очень рады, что вы сегодня зашли к нам, Счастливый Путь, – и притом минута в минуту.

Счастливый Путь: Спасибо... Я всегда пунктуален, для меня это очень важно.

HeadFirst: Вот как? Вы никогда не опаздываете?

Счастливый путь: Представьте, никогда. А еще я никогда не забываю о важных встречах. И никогда не совершаю ошибок, у меня не бывает никаких неожиданностей... Вы всегда можете рассчитывать на то, что у меня все пройдет именно так, как вам нужно, каждый раз.

HeadFirst: И так вы заработали свое имя? Люди счастливы от того, что вы всегда все делаете вовремя и никогда не ошибаетесь?

Счастливый Путь: Нет, но вы почти угадали. Меня называют «Счастливый Путь», потому что со мной все идет так, как вы надеетесь. На «Счастливом Пути» не бывает никаких неприятностей.

HeadFirst: Должен признать, это меня поражает – с вами никогда ничего не случается плохого. Вы уверены, что живете в реальном мире?

Счастливый Путь: Не поймите меня неправильно... Безусловно, в реальном мире случается всякое. Но в таких случаях я просто передаю управление своему приятелю – Альтернативному Пути.

HeadFirst: Кажется, я начинаю понимать... Выходит, проблемы возможны, но их решением занимается Альтернативный Путь.

Счастливый Путь: Да, примерно так. Но меня это не слишком беспокоит. Я работаю тогда, когда светит солнышко и все идет так, как надеются люди.

HeadFirst: Наверное, вам очень нравится ваша работа?

Счастливый Путь: В основном нравится, конечно. Но неприятности случаются довольно часто. Меня редко проходят от начала до конца. Рано или поздно управление обычно передается Альтернативному Пути, но мы с ним хорошо ладим, так что проблем нет.

HeadFirst: И вам никогда не кажется, что Альтернативный Путь вмешивается в ваши дела? Наверное, между вами возникают трения...

Счастливый Путь: Вовсе нет. Я хочу сказать, что мы занимаемся одним делом: ведем заказчика к его цели и следим за тем, чтобы он был доволен. И если мы четко определены, это существенно упростит программирование приложения.

HeadFirst: Что ж, интервью подходит к концу. В следующий раз мы постараемся пригласить Альтернативный Путь и выслушать его точку зрения. А до тех пор старайтесь держаться Счастливого Пути, но не забывайте о возможных проблемах!

Кто и что делает?

Слева перечислены некоторые термины, которые вы узнали в этой главе. Справа приведены описания и условия использования. Ваша задача — соединить термины в левом столбце с их назначениями в правом столбце.

Внешний _____

Запускает последовательность шагов, описанную в варианте использования. Без этого вариант использования не смог бы стартовать.

_____ **использования**

То, что должна делать система при успешной работе.

Начальное _____

Сообщает, когда вариант использования завершен. Без этого варианты использования продолжали бы работать бесконечно.

Требование

Помогает в сборе требований. Описывает, что должна делать система.

_____ **ценность**

То, как работает система, когда все идет нормально. Обычно именно это описывает заказчик, говоря о системе.

_____ **условие**

Всегда самый первый шаг в варианте использования.

_____ **путь**

Без этого вариант использования никому не нужен. Варианты использования, не обладающие этим, всегда абсолютно бесполезны.

Ой... Части терминов куда-то пропали. Найдите справа соответствие для термина и заполните пропуски.

Кто и что делает? Решение

Слева перечислены некоторые термины, которые вы узнали в этой главе. Справа приведены описания и условия использования. Ваша задача — соединить термины в левом столбце с их назначениями в правом столбце.

Внешний Initiator

Вариант **использования**

Начальное условие

Требование

Очевидная **ценность**

Конечное **условие**

Основной **путь**

Запускает последовательность шагов, описанную в варианте использования. Без этого вариант использования не смог бы стартовать.

То, что должна делать система при успешной работе.

Сообщает, когда вариант использования завершен. Без этого варианты использования продолжали бы работать бесконечно.

Помогает в сборе требований. Описывает, что должна делать система.

То, как работает система, когда все идет нормально. Обычно именно это описывает заказчик, говоря о системе.

Всегда самый первый шаг в варианте использования.

Без этого вариант использования никому не нужен. Варианты использования, не обладающие этим, всегда абсолютно бесполезны.

Проверьте, совпадают ли предложенные вами варианты с нашими.

Возьми в руку карандаш

Давайте напишем еще несколько вариантов использования.

Ниже изображены еще три потенциальных заказчика, интересующихся собачьими дверями Дуга. Напишите для каждого случая вариант использования, решающий проблему этого заказчика.

Фирма Дуга сотрудничает с местной компанией, обеспечивающей безопасность жилья, помогая ей справиться с растущей клиентской базой и требованиями такого рода

Битси постоянно пытается выбраться из дома через заднюю дверь или окна кухни. Мне нужна система, которая запирает собачью дверь и окна, когда я ввожу код, чтобы Битси не могла убежать.

Битси

Кристен

Брюс постоянно лает, и я не знаю, хочет он на улицу или нет. Вы можете построить дверь, которая бы открывалась, если он поскребет ее лапой?

Текс постоянно таскает грязь в дом. Мне нужна дверь, которая автоматически закрывается каждый раз, когда он выходит наружу, и остается закрытой, пока я не нажму кнопку, чтобы впустить его.

Джон

Текс

Холли

Брюс

Ответы на с. 128.

Возьми в руку карандаш
Решение

Давайте напишем еще несколько вариантов использования.

Ниже изображены еще три потенциальных заказчика, интересующихся собачьими дверями Дуга. Напишите для каждого случая вариант использования, решающий проблему этого заказчика.

Битси постоянно пытается выбраться из дома через заднюю дверь или окна кухни. Мне нужна система, которая запирает собачью дверь и окна, когда я ввожу код, чтобы Битси не могла убежать.

Битси

Кристен

Вариант использования для Кристен состоит всего из двух шагов: она вводит код, дверь и окна запираются.

Собачья дверь для Кристен и Битси

1. **Кристен вводит код.**
2. **Собачья дверь и все окна в доме запираются.**

Хотя это и собачья дверь, Битси на самом деле никак не влияет на поведение системы!

Текс постоянно таскает грязь в дом. Мне нужна дверь, которая автоматически закрывается каждый раз, когда он выходит наружу, и остается закрытой, пока я не нажму кнопку, чтобы впустить его.

Требование Джона очень похоже на то, чего хотели Тодд и Джина. Одна из сторон грамотного составления набора требований — умение понять, что вы когда-то уже делали нечто похожее на то, чего хочет заказчик.

Джон

Текс

Брюс постоянно лает, и я не знаю, хочет он на улицу или нет. Вы можете построить дверь, которая бы открывалась, если он поскребет ее лапой?

Собачья дверь для Холли и Брюса

1. Брюс царапает собачью дверь.
2. Дверь открывается.
3. Брюс выходит из дома.
4. Дверь автоматически закрывается.
4. Брюс делает свои дела.
5. Брюс снова царапает дверь.
6. Дверь открывается.
7. Брюс заходит в дом.
8. Дверь автоматически закрывается.

В исходной просьбе Холли многих пунктов нет. Но если хорошенько подумать о том, как будет использоваться ее система, вычислить недостающее будет несложно.

Хотя Джон сказал, что Текс обычно возвращается грязным, это происходит не всегда... Так что приведение лап Текса в порядок следует оформить в виде альтернативного пути.

Для этого варианта использования необходима дополнительная информация... Похоже, нам придется задать Джону еще несколько вопросов.

Собачья дверь для Джона и Текса

1. Собачья дверь (как-то) открывается.
2. Текс выходит на улицу.
3. Дверь автоматически закрывается.
4. Текс делает свои дела.
   4.1 Текс пачкает лапы.
   4.2 Джон вытирает лапы Текса.
5. Джон нажимает кнопку.
6. Дверь открывается.
7. Текст заходит в дом.
8. Дверь автоматически закрывается.

Развлечения с магнитами

Помните три составляющие варианта использования? Пора применить то, что мы узнали, на практике. Ниже представлено несколько примеров вариантов использования; ваша задача — расставить магниты, которые лежат внизу, в правильных местах каждого варианта использования.

Если вы что-то забыли, вернитесь к с. 106.

Собачья дверь для Холли и Брюса

Брюс царапает дверь, чтобы его выпустили на улицу. Дверь автоматически открывается, Брюс выходит. Дверь закрывается по истечении заданного промежутка времени. Брюс делает свои дела, потом снова царапает дверь. Дверь автоматически открывается, Брюс заходит в дом. Дверь автоматически закрывается.

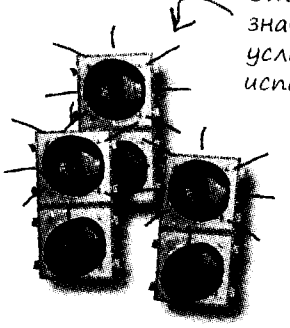
Если Брюе царапает дверь, но остается внутри (или снаружи), он может снова поцарапать дверь, чтобы открыть ее (изнутри или снаружи).

Скорее всего вы без особых трудностей поймете смысл альтернативного формата вариантов использования. А если что-то покажется непонятным, обращайтесь к описанию альтернативных форматов вариантов использования в приложении I.

Собачья дверь для Кристен и Битси

1. **Кристен вводит код.**
2. **Собачья дверь и все окна в доме запираются.**

Используйте эти магниты для обозначения очевидной ценности вариантов использования.



Этот магнит обозначает начальное условие варианта использования.

Собачья дверь для Джона и Текса

Первичный субъект: Текс

Вторичный субъект: Джон

Предусловия: дверь открыта, чтобы Текс мог выйти наружу.

Цель: Текс делает свои дела и возвращается обратно, не занося грязь в дом.

Основной путь

1. Текс выходит из дома.
2. Дверь автоматически закрывается.
3. Текс делает свои дела.
4. Джон нажимает кнопку.
5. Дверь открывается.
6. Текс заходит в дом.
7. Дверь автоматически закрывается.

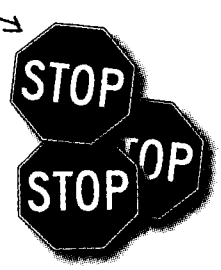
Расширения

3.1 Текс пачкает лапы.

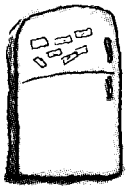
3.2 Джон вытирает лапы Текса.

Ответы на с. 132.

Этим магнитом обозначается конечное условие варианта использования. Как узнать, когда выполнение варианта использования завершено?



Здесь Фидо представляет внешний инициатор варианта использования, который запускает вариант на выполнение.



Развлечения с магнитами. Решение

Помните три составляющие варианта использования? Пора применить то, что мы узнали, на практике. Ниже представлено несколько примеров вариантов использования; ваша задача — расставить магниты, которые лежат внизу, в правильных местах каждого варианта использования.

Очевидная ценность — Super Buy

Начало и конец — STOP

Внешний инициатор

Собачья дверь для Холли и Брюса

Брюс царапает дверь, чтобы его выпустили на улицу. Дверь автоматически открывается, Брюс выходит. Дверь закрывается по истечении заданного промежутка времени. Брюс делает свои дела, потом снова царапает дверь. Дверь автоматически открывается, Брюс заходит в дом. Дверь автоматически закрывается.

Если Брюс царапает дверь, но остается внутри (или снаружи), он может снова поцарапать дверь, чтобы открыть ее (изнутри или снаружи).

STOP

Super Buy

Брюс может выйти из дома, а Холли даже не придется открывать и закрывать дверь (и даже слушать, когда Брюс залает).

Очевидная ценность в самом варианте использования не указывается (по крайней мере, в большинстве форматов вариантов), так что ее приходится определять отдельно.

В этом формате варианта использования следует обращать особое внимание на конечное условие; при наличии альтернативных путей обычно оно не является последним приложением.

Собачья дверь для Кристен и Битси

Кристен вводит код.

2. Собачья дверь и все окна в доме запираются.

STOP

Super Buy

Битси не сможет выйти из дома, пока Кристен ее не выпустит.

Начальное условие и внешний инициатор обычно указываются в первом шаге варианта использования.

Конечное условие обычно указывается в последнем шаге варианта использования.

В этом формате варианта использования внешний инициатор всегда является первичным субъектом.

Собачья дверь для Джона и Текса

Первичный субъект: Текс

Вторичный субъект: Джон

Предусловия: дверь открыта, чтобы Текс мог выйти наружу.

Цель: Текс делает свои дела и возвращается обратно, не [illegible]енося грязь в дом.

Super Buy

Там, где цель варианта использования сформулирована явно, она может рассматриваться как очевидная ценность.

Основной путь

1. Текс выходит из дома.
2. Дверь автоматически закрывается.
3. Текс делает свои дела.
4. Джон нажимает кнопку.
5. Дверь открывается.
6. Текс заходит в дом.
7. Дверь автоматически закрыва[illegible]

STOP

Расширения

3.1 Текс пачкает лапы.

3.2 Джон вытирает лапы Текса.

Ищите последний шаг основного пути, а не последний шаг расширений.

Возьми в руку карандаш

В чем же проявляется настоящая сила вариантов использования?

Вы уже видели, какую помощь варианты использования оказывают в составлении полных списков требований. Ниже приведено еще несколько примеров вариантов использования. Ваша задача — определить, полон ли список, приведенный рядом с каждым вариантом, или в него необходимо включить дополнительные требования.

Собачья дверь для Кристен и Битси

Вариант использования

1. Кристен вводит код.
2. Собачья дверь и все окна в доме запираются.

Собачья дверь для Кристен и Битси

Список требований

1. Клавиатура должна обеспечивать ввод кода из 4 цифр.
2. Клавиатура должна уметь запирать собачью дверь.

Список требований для собачьей двери Кристен. Может, по варианту использования вы найдете какие-то неполные или отсутствующие требования? В таком случае запишите дополнительные требования, которые, по вашему мнению, должны быть реализованы дверью.

Помните Кристен и Битси?

Собачья дверь Холли и Брюса

Вариант использования

1. Брюс царапает собачью дверь.
2. Дверь открывается.
3. Брюс выходит из дома.
4. Дверь автоматически закрывается.
4. Брюс делает свои дела.
5. Брюс снова царапает дверь.
6. Дверь открывается.
7. Брюс заходит в дом.
8. Дверь автоматически закрывается.

Холли с нетерпением ждет установки новой двери. Дверь будет работать, она вся в предвкушении!

Собачья дверь Холли и Брюса

Список требований

1. Дверь должна распознавать царапанье собаки.
2. Дверь должна уметь открываться по команде (из п.1).

Все ли учтено? Вы должны позаботиться о том, чтобы Холли осталась довольна новой дверью.

Ответы на с. 136.

Возьми в руку карандаш
Решение

В чем же проявляется настоящая сила вариантов использования?

Вы уже видели, какую помощь варианты использования оказывают в составлении полных списков требований. Ниже приведено еще несколько примеров вариантов использования. Ваша задача — определить, полон ли список, приведенный рядом с каждым вариантом, или в него необходимо включить дополнительные требования.

Собачья дверь для Кристен и Битси

Вариант использования

1. Кристен вводит код.
2. Собачья дверь и все окна в доме запираются.

Собачья дверь для Кристен и Битси

Список требований

1. Клавиатура должна обеспечивать ввод кода из 4 цифр.
2. Клавиатура должна уметь запирать собачью дверь и все окна в доме.
3. Клавиатура должна уметь отпирать собачью дверь и все окна в доме.

Маленькая тонкость — в варианте использования ничего не сказано о том, как Битси будет возвращаться домой; получается, вариант использования и список требований не полны. Вряд ли Кристи обрадуется, когда поймет, что ее окна не открываются.

Неполное требование... Кристен хочет иметь возможность запирать двери и окна.

Будьте внимательны! Хорошие варианты использования порождают хорошие требования, а плохие — или неполные — варианты приводят к появлению ПЛОХИХ требований!

Собачья дверь Холли и Брюса

Вариант использования

1. Брюс царапает собачью дверь.
2. Дверь открывается.
3. Брюс выходит из дома.
4. Дверь автоматически закрывается.
4. Брюс делает свои дела.
5. Брюс снова царапает дверь.
6. Дверь открывается.
7. Брюс заходит в дом.
8. Дверь автоматически закрывается.

Собачья дверь Холли и Брюса

Список требований

1. Дверь должна распознавать царапанье собаки.
2. Дверь должна уметь открываться по команде (из п.1).
3. Дверь должна автоматически закрываться.

Требование, знакомое нам по двери для Тодда и Джины.

Инструментарий ООАП

Самое важное в ООАП — написание хороших программ, а это возможно только при одном условии: если вы позаботитесь о том, чтобы ваша программа делала именно то, чего хочет заказчик. В этой главе мы рассмотрели некоторые инструменты, благодаря которым при виде построенной вами системе на лице заказчика появится довольная улыбка. Некоторые ключевые инструменты, которые следует держать под рукой:

Требования

Хорошие требования помогают убедиться в том, что ваша система работает так, как хочет заказчик.

Проследите за тем, чтобы требования распространялись на все шаги вариантов использования вашей системы.

Из вариантов использования можно узнать то, о чем заказчик забыл вам рассказать.

Анализ вариантов использования может выявить неполные или отсутствующие требования.

Некоторые ключевые инструменты, о которых было рассказано в этой главе.

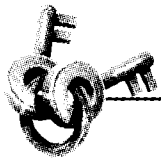
В следующих главах эти категории* пополнятся множеством новых инструментов.

Основы ООАП

Принципы ООАП

* Читателям книги «Head First Design Patterns» (Э. Фримен, Э. Фримен. Паттерны проектирования. Питер, 2012) эти категории покажутся знакомыми... Потому что ООАП и паттерны проектирования идут рука об руку.

КЛЮЧЕВЫЕ МОМЕНТЫ



- Требования — то, что должна делать ваша система, чтобы она могла считаться правильно работающей.
- Исходные требования обычно предоставляются заказчиком.
- Чтобы сформировать хороший набор требований, следует создать варианты использования для вашей системы.
- Варианты использования подробно описывают, как должна работать ваша система.
- Вариант использования имеет одну цель, но может иметь несколько путей для достижения этой цели.
- Вариант использования — это просто текстовое описание того, как работает ваша система.
- У вас должен быть по меньшей мере один вариант использования для каждой цели, которую должна реализовать ваша система.
- Когда варианты использования будут готовы, можно переходить к уточнению и расширению требований.
- Хороший набор требований делает возможными все варианты использования.
- Ваша система должна работать в условиях реального мира, а не по по тем правилам, которые вы для нее установили.
- Если что-то пойдет не так, система должна иметь альтернативные пути для достижения ее целей.

Развлечения с магнитами. Решение

Класс DogDoor готов, осталось написать класс для пульта дистанционного управления. Мы создали заготовку класса, а ваша задача — закончить построение кода Remote при помощи магнитов, находящихся в нижней части страницы.

Будьте внимательны... Некоторые магниты могут оказаться лишними.

```
public class Remote {

  private DogDoor door;

  public Remote(DogDoor door) {
    this.door = door;
  }

  public void pressButton() {
    System.out.println("Pressing the remote control button...");
    if (door.isOpen) {
      door.close();
    } else {
      door.open();
    }
  }
}
```

Оставшиеся магниты.

isOpen, boolean, DogDoor, isOpen, true, false, true, false, close, boolean

Я тебя люблю, ты мой идеал... Теперь изменись

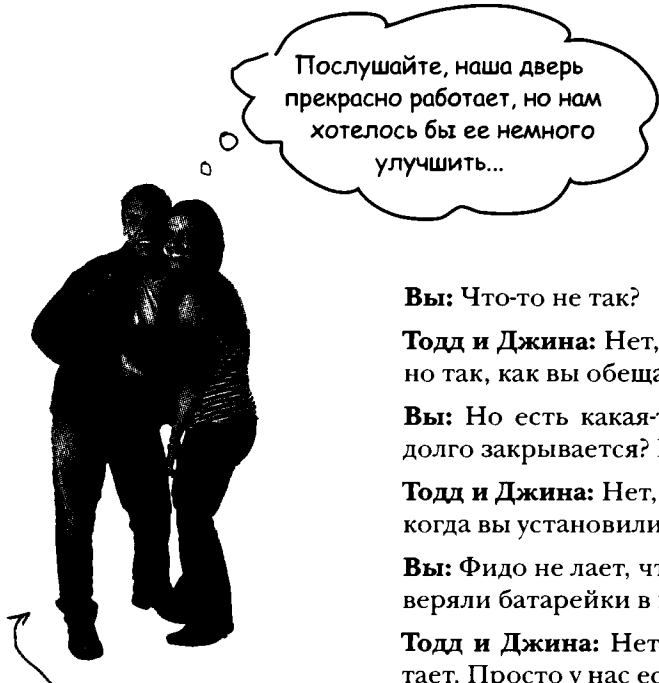
Думаете, вы сделали то, что нужно заказчику? Как бы не так... Вы поговорили с заказчиком, собрали требования, записали варианты использования и выдали убойное приложение. Можно расслабиться, верно? Верно... Пока заказчик не решит, что ему нужно **что-то отличное от того, о чем он говорил вам**. То, что вы сделали, ему понравилось, честно, но **теперь это не совсем то, что нужно**. В реальном мире **требования всегда изменяются**; вы должны адаптироваться к этим изменениям и сделать так, чтобы заказчик был доволен.

Вы — герой!

Солнце сияет, вы держите бокал с «пинаколадой», в кармане шорт лежит пачка стодолларовых купюр... Так живет программист, который сделал «Собачьи двери Дуга» успешным предприятием. Дверь, которую вы построили для Тодда и Джины, прославила фирму, и теперь Дуг продает ее по всему миру.

Дуг отлично зарабатывает на вашем коде.

А потом был телефонный звонок...



Тодд и Джина прерывают ваш отпуск.

Вы: Что-то не так?

Тодд и Джина: Нет, нет, все хорошо. Дверь работает точно так, как вы обещали.

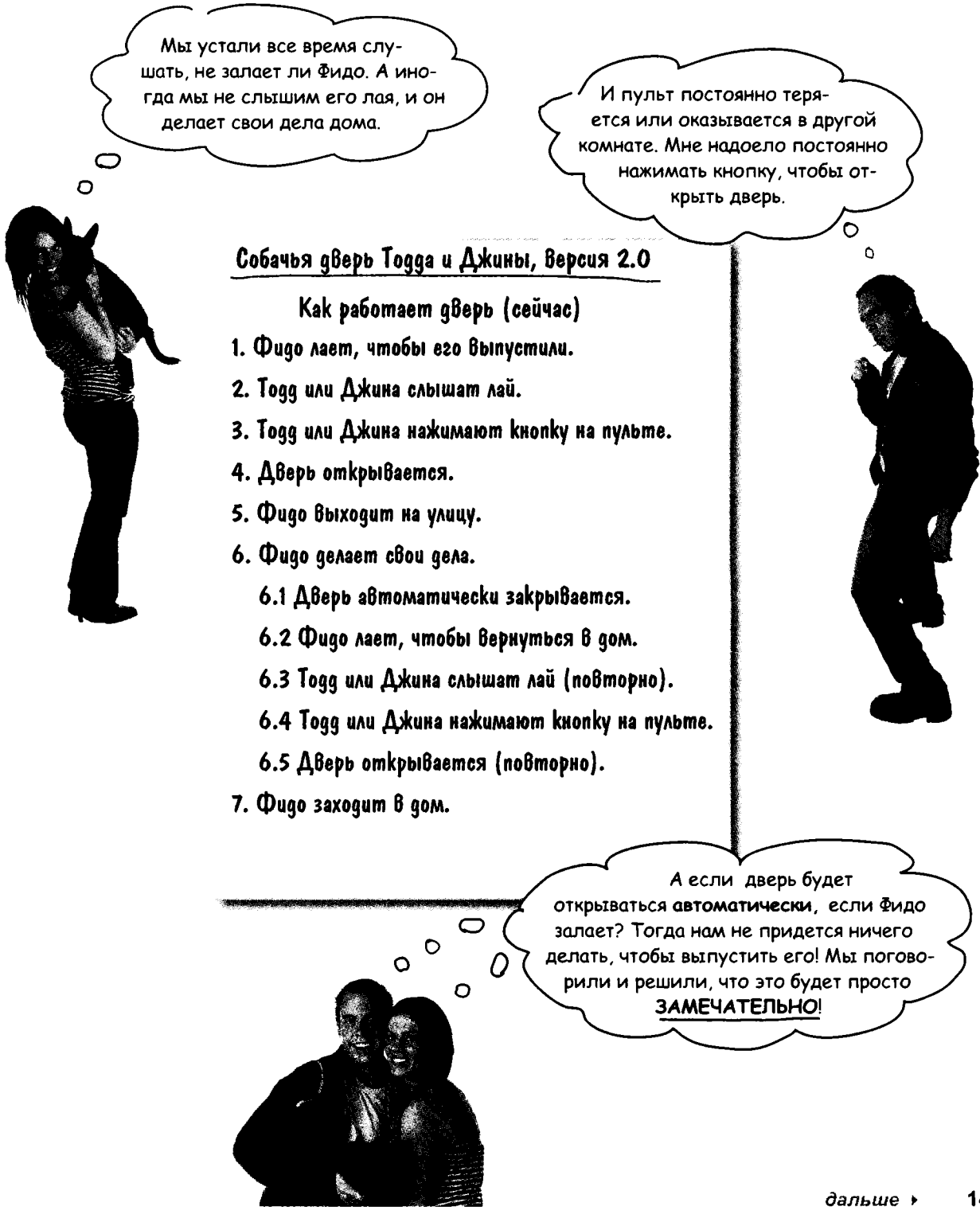
Вы: Но есть какая-то проблема, верно? Дверь слишком долго закрывается? Кнопка на пульте не работает?

Тодд и Джина: Нет, правда... Все работает, как в тот день, когда вы установили дверь и показали нам.

Вы: Фидо не лает, чтобы его выпустили? Кстати, а вы проверяли батарейки в пульте?

Тодд и Джина: Нет, честное слово, дверь отлично работает. Просто у нас есть пара идей по поводу изменений...

Вы: Но если все работает, то в чем проблема?


Мы устали все время слушать, не залает ли Фидо. А иногда мы не слышим его лая, и он делает свои дела дома.
И пульт постоянно теряется или оказывается в другой комнате. Мне надоело постоянно нажимать кнопку, чтобы открыть дверь.
Собачья дверь Тодда и Джины, версия 2.0
Как работает дверь (сейчас)
1. Фидо лает, чтобы его выпустили.
2. Тодд или Джина слышат лай.
3. Тодд или Джина нажимают кнопку на пульте.
4. Дверь открывается.
5. Фидо выходит на улицу.
6. Фидо делает свои дела.
6.1 Дверь автоматически закрывается.
6.2 Фидо лает, чтобы вернуться в дом.
6.3 Тодд или Джина слышат лай (повторно).
6.4 Тодд или Джина нажимают кнопку на пульте.
6.5 Дверь открывается (повторно).
7. Фидо заходит в дом.
А если дверь будет открываться автоматически, если Фидо залает? Тогда нам не придется ничего делать, чтобы выпустить его! Мы поговорили и решили, что это будет просто ЗАМЕЧАТЕЛЬНО!

Снова беремся за карандаш

Пора снова поработать над собачьей дверью Тодда и Джины. Нужно, чтобы дверь открывалась каждый раз, когда Фидо залает. И начнем мы с...

Одну минуту... Да что же это! Мы уже построили **работающую** дверь, и заказчик сказал, что это **хорошо**. А теперь только потому, что у них появилась новая мысль, мы должны вносить изменения?

Заказчик всегда прав

Даже если требования изменились, вы должны быть готовы обновить приложение и позаботиться о том, чтобы оно работало так, как хочет заказчик. Если у клиента возникли новые потребности, приложение придется изменить в соответствии с новыми потребностями.

Дуга такие вещи только радуют, потому что он может взять с Тодда и Джины дополнительную плату.

Мозговой штурм

Вы только что обнаружили единственный постоянный фактор в области анализа и проектирования программного обеспечения. Как вы думаете, что это за константа?

Единственный постоянный фактор в области анализа и проектирования программного обеспечения*

На что можно всегда рассчитывать при написании программ?

Как бы вы ни работали, что бы вы ни создавали, на каком бы языке вы ни программировали – какой единственный фактор будет присутствовать всегда?

ИЗМЕНЕНИЯ

(прочитайте ответ в зеркале)

Как бы хорошо ни было спроектировано приложение, со временем оно всегда будет расти и изменяться. Вы будете находить новые решения проблем, языки программирования будут развиваться, ваши милые заказчики выдадут новые безумные требования, которые заставят вас «исправлять» работающие приложения.

Требования всегда изменяются. Но при наличии хороших вариантов использования можно быстро адаптировать программный продукт к новым требованиям.

Возьми в руку карандаш

Требования постоянно изменяются — в разгаре работы над проектом или когда вы думаете, что работа уже завершена. Запишите возможные причины изменений в требованиях приложений, над которыми вы работаете в настоящее время.

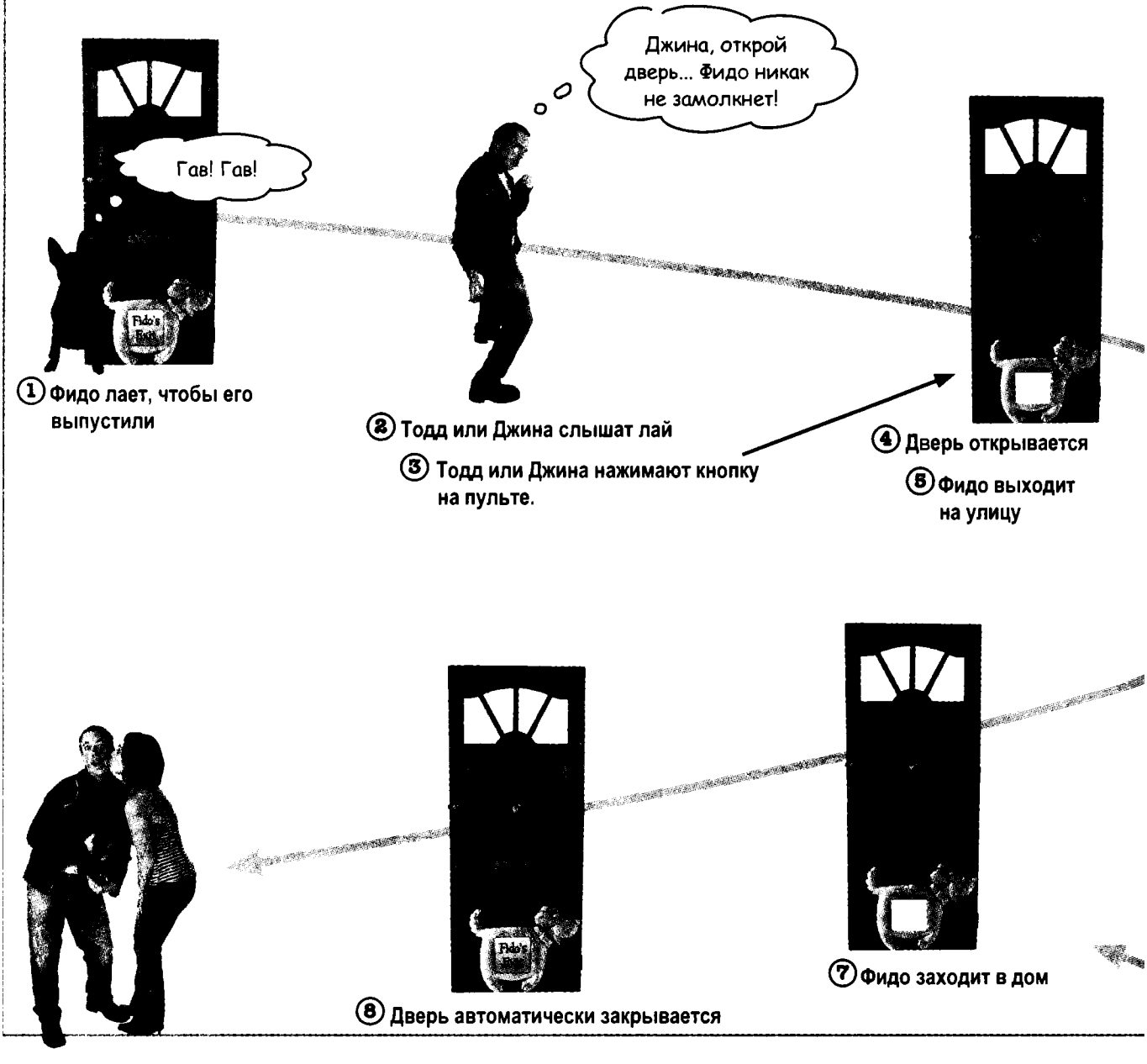
Заказчик решил, что приложение должно работать по-другому.

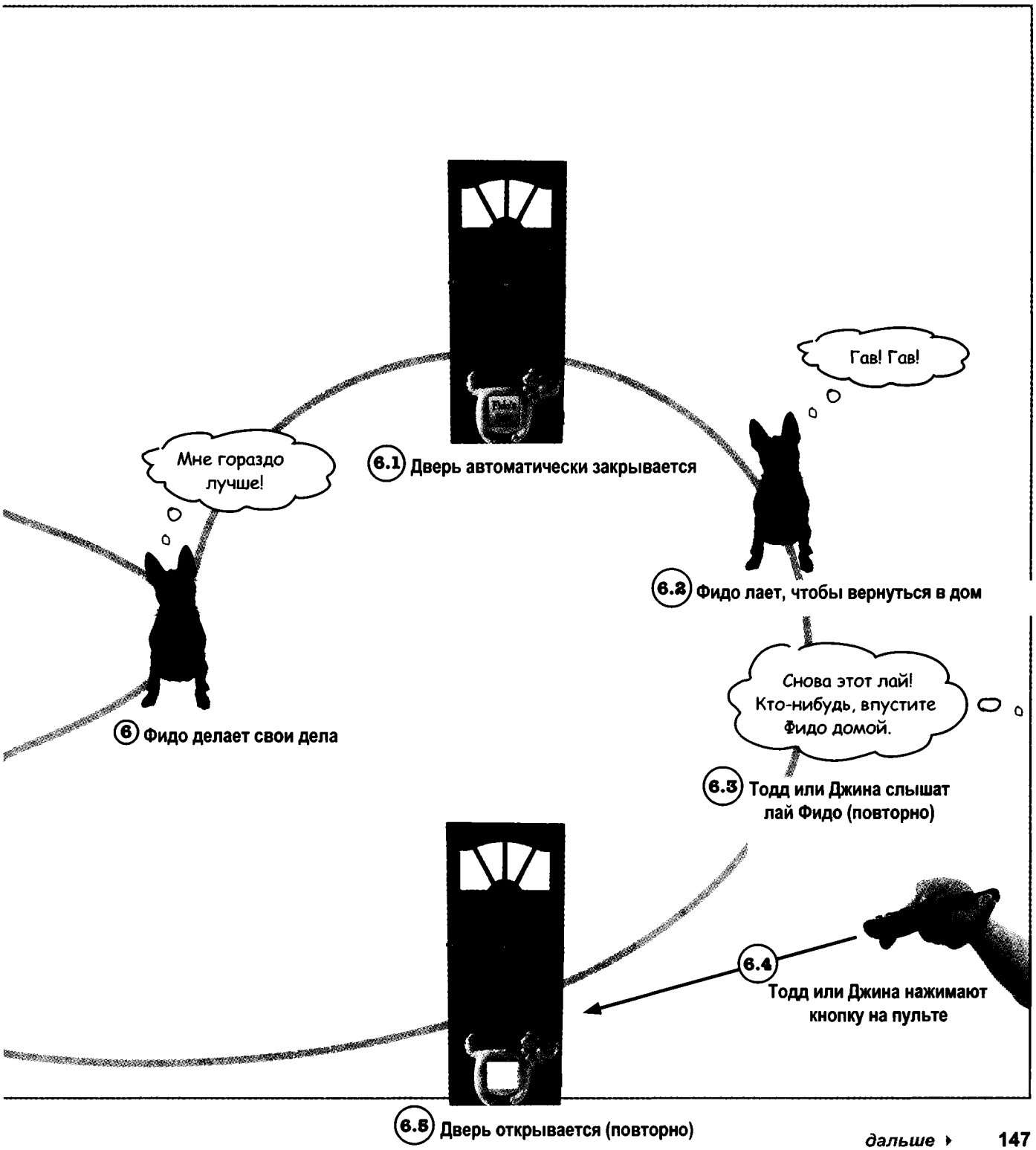
Мой начальник полагает, что вместо настольного приложения лучше использовать формат веб-приложения.

Если вы читали книгу «Head First Design Patterns» (Э. Фримен, Э. Фримен. Паттерны проектирования. СПб.: Питер, 2012), эта страница может показаться знакомой. Авторы так хорошо описали процесс внесения изменений в проекты, что мы решили прихватить их идеи и внести с небольшими ИЗМЕНЕНИЯМИ. Спасибо, Бет и Эрик!

Добавление устройства распознавания лая в дверь Тодда и Джины

Обновите диаграмму и добавьте альтернативный путь, в котором Фидо лает, новый распознаватель лая слышит его, и дверь автоматически открывается. Пульт дистанционного управления тоже должен работать, поэтому удалять с диаграммы ничего не нужно; просто добавьте новый путь, в котором Фидо своим лаем открывает дверь.

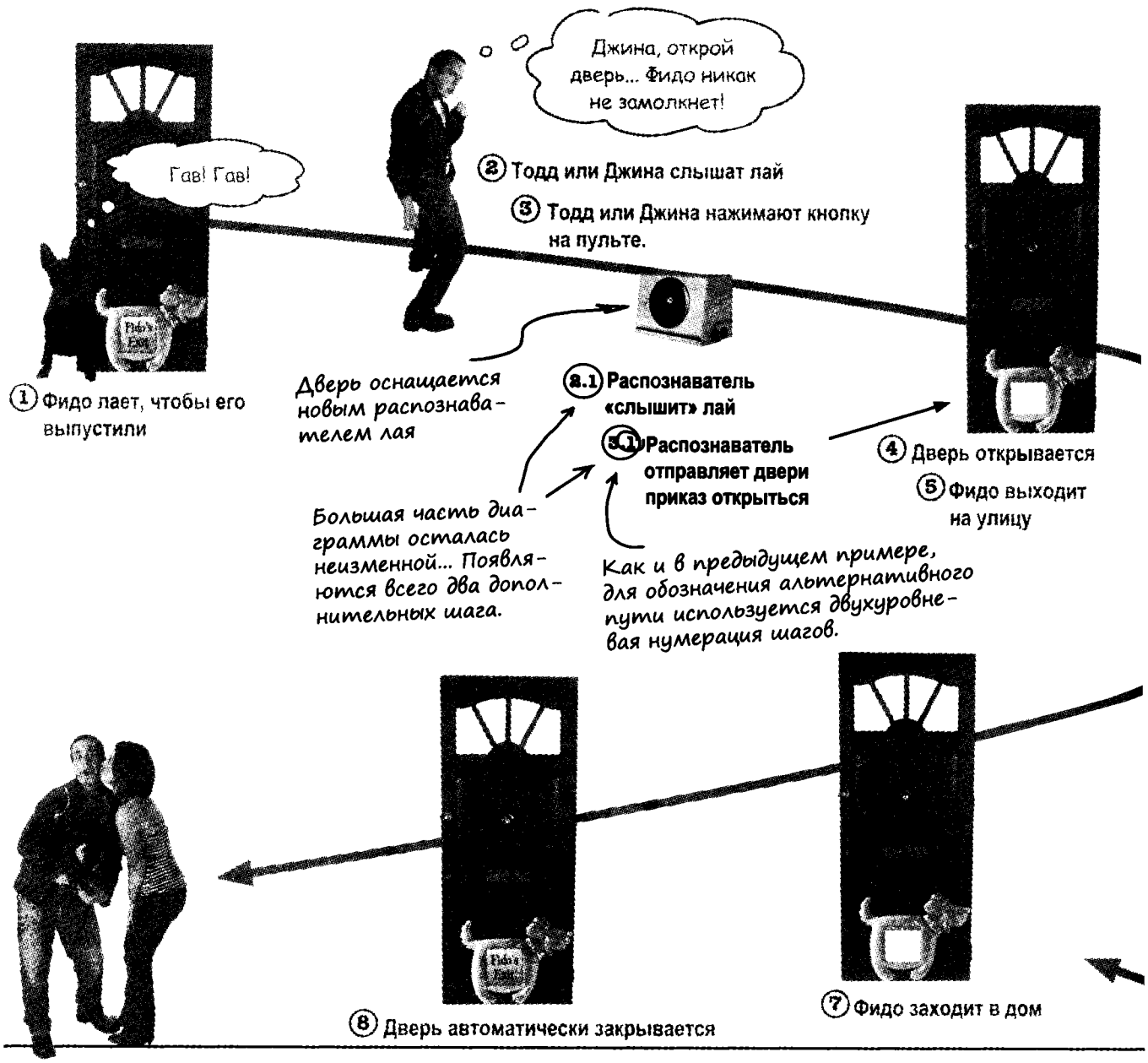


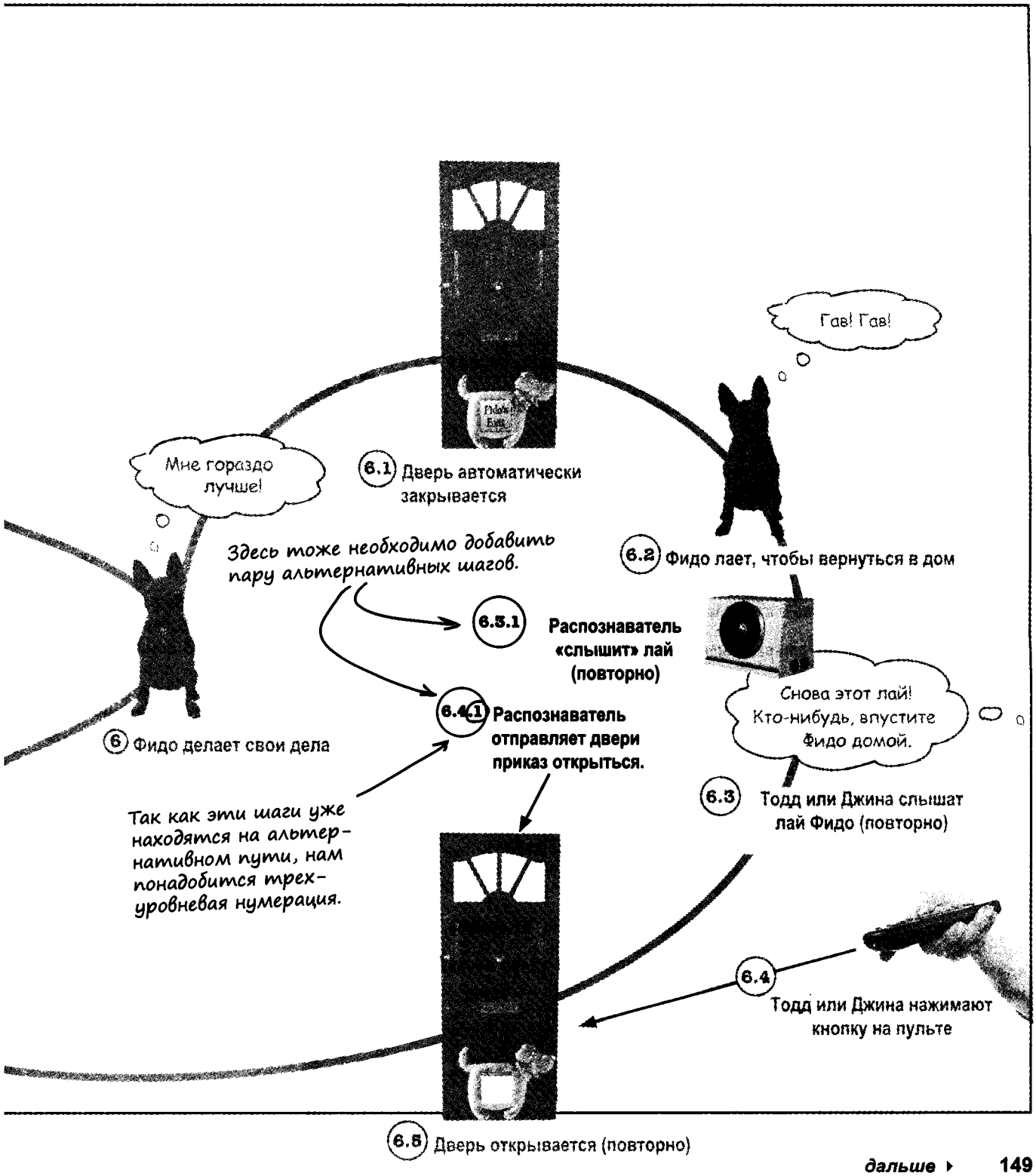

Гав! Гав!
Мне гораздо лучше!
6.1 Дверь автоматически закрывается
6.2 Фидо лает, чтобы вернуться в дом
Снова этот лай! Кто-нибудь, впустите Фидо домой.
6 Фидо делает свои дела
6.3 Тодд или Джина слышат лай Фидо (повторно)
6.4 Тодд или Джина нажимают кнопку на пульте
6.5 Дверь открывается (повторно)

Упражнение
Решение

Дуг изобрел устройство для распознавания лая, но вы должны понять, как использовать его в системе с собачьей дверью.

Ниже представлено наше решение проблемы Тодда и Джины и наша реализация двери, распознающей собачий лай. Посмотрите, похожи ли ваши дополнения к диаграмме на то, что получилось у нас.




Гав! Гав!
Мне гораздо лучше!
6.1 Дверь автоматически закрывается
Здесь тоже необходимо добавить пару альтернативных шагов.
6.2 Фидо лает, чтобы вернуться в дом
6.3.1 Распознаватель «слышит» лай (повторно)
Снова этот лай! Кто-нибудь, впустите Фидо домой.
6.4.1 Распознаватель отправляет двери приказ открыться.
6 Фидо делает свои дела
6.3 Тодд или Джина слышат лай Фидо (повторно)
Так как эти шаги уже находятся на альтернативном пути, нам понадобится трехуровневая нумерация.
6.4 Тодд или Джина нажимают кнопку на пульте
6.5 Дверь открывается (повторно)

Вариант использования окончательно запутался. С этими альтернативными путями не поймешь, что вообще происходит!

Дополнительный путь?
Альтернативный путь?
Кто знает?

Собачья дверь для Тодда и Джины, версия 2.1

Как должна работать дверь

1. Фидо лает, чтобы его выпустили.
2. Тодд или Джина слышат лай.
   - 2.1. Распознаватель «слышит» лай.
3. Тодд или Джина нажимают кнопку на пульте.
   - 3.1. Распознаватель отправляет двери приказ открыться.
4. Дверь открывается.
5. Фидо выходит на улицу.
6. Фидо делает свои дела.
   - 6.1. Дверь автоматически закрывается.
   - 6.2. Фидо лает, чтобы вернуться в дом.
   - 6.3. Тодд или Джина слышат лай (повторно).
     - 6.3.1. Распознаватель «слышит» лай (повторно).
   - 6.4. Тодд или Джина нажимают кнопку на пульте.
     - 6.4.1. Распознаватель отправляет двери приказ открыться.
   - 6.5. Дверь открывается (повторно).
7. Фидо заходит в дом.
8. Дверь автоматически закрывается.

Теперь появились альтернативные шаги для пунктов 2 и 3.

Даже в альтернативных путях появились свои альтернативы.

Эти действия изображены в виде вложенных шагов, но в действительности они образуют совершенно новый путь в варианте использования.

Шаги второго уровня образуют дополнительный набор, по которому можно пройти...

...но фактически они образуют новый путь прохождения варианта использования.

Все равно сценарий получается запутанным. Кажется, будто Тодд и Джина всегда слышат лай Фидо, а распознаватель речи распознает его только иногда. Но ведь это не то, чего хотят Тодд с Джиной...

А вы видите, о чем говорит Джеральд? Тодд и Джина хотели, чтобы им больше не пришлось прислушиваться к лаю Фидо.

Собачья дверь для Тодда и Джины, версия 2.1

Как должна работать дверь

1. Фидо лает, чтобы его выпустили.
2. Тодд или Джина слышат лай.
   2.1. Распознаватель «слышит» лай.
3. Тодд или Джина нажимают кнопку на пульте.
   3.1. Распознаватель отправляет двери приказ открыться.
4. Дверь открывается.
5. Фидо выходит на улицу.
6. Фидо делает свои дела.
   6.1. Дверь автоматически закрывается.
   6.2. Фидо лает, чтобы вернуться в дом.
   6.3. Тодд или Джина слышат лай (повторно).
      6.3.1. Распознаватель «слышит» лай (повторно).
   6.4. Тодд или Джина нажимают кнопку на пульте.
      6.4.1. Распознаватель отправляет двери приказ открыться.
   6.5. Дверь открывается (повторно).
7. Фидо заходит в дом.
8. Дверь автоматически закрывается.

В новом варианте использования нужно указать, что происходит либо шаг 2, либо шаг 2.1...

...а потом либо шаг 3, либо шаг 3.1.

Здесь происходит либо шаг 6.3, либо шаг 6.3.1...

...а затем либо шаг 6.4, либо шаг 6.4.1.

Варианты использования должны быть понятными для Вас

Если вариант использования вам кажется запутанным, *просто перепишите его.* Существует множество разных способов записи вариантов; важно, чтобы выбранный способ был понятным для вас, вашей группы и тех людей, которым придется его объяснять. Давайте перепишем вариант использования со страницы 151, чтобы он стал более понятным.

Шаги, которые могут происходить вместо шагов главного пути, переместились вправо.

Эта надпись напоминает о том, что шаги в левом столбце составляют основной путь сценария.

Собачья дверь для Тодда и Джины, версия 2.2

Как должна работать дверь

Основной путь	Альтернативные пути
1. Фидо лает, чтобы его выпустили.	
2. Тодд или Джина слышат лай.	2.1. Распознаватель «слышит» лай.
3. Тодд или Джина нажимают кнопку на пульте.	3.1. Распознаватель отправляет двери приказ открыться.
4. Дверь открывается.	
5. Фидо выходит на улицу.	
6. Фидо делает свои дела.	
6.1 Дверь автоматически закрывается.	
6.2 Фидо лает, чтобы вернуться в дом.	
6.3 Тодд или Джина слышат лай (повторно).	6.3.1. Распознаватель «слышит» лай (повторно).
6.4 Тодд или Джина нажимают кнопку на пульте.	6.4.1. Распознаватель отправляет двери приказ открыться.
6.5 Дверь открывается (повторно).	
7. Фидо заходит в дом.	
8. Дверь автоматически закрывается.	

Когда из двух шагов остается только один, он всегда будет применяться при прохождении варианта использования.

Стало немного понятнее: используется шаг 2 ИЛИ шаг 2.1, затем шаг 3 ИЛИ шаг 3.1.

Эти шаги второго уровня необязательны... Они могут использоваться, а могут и не использоваться. Но они все равно находятся в левом столбце, потому что они не заменяют шагов основного пути.

Эти шаги в правом столбце могут заменять шаги 6.3 и 6.4. При прохождении варианта использования выбирается только один шаг: ЛИБО шаг в левом столбце, ЛИБО шаг в правом столбце.

Как бы ни выполнялся этот вариант использования, он всегда приходит к шагу 8 основного пути.

Если вариант использования можно записывать так, как мы хотим, почему бы не сделать распознаватель лая частью основного пути? Ведь именно этот путь должен использоваться в большинстве случаев, не так ли?

Отличная мысль!

Основной путь должен описывать действия, выполняемые в большинстве случаев. Так как Тодд и Джина хотят, чтобы распознаватель выпускал Фидо чаще, чем кнопка на пульте, мы переместим эти шаги в основной путь:

Собачья дверь для Тодда и Джины, версия 2.3

Как должна работать дверь

Основной путь

1. Фидо лает, чтобы его выпустили.
2. Распознаватель «слышит» лай.
3. Распознаватель отправляет двери приказ открыться.
4. Дверь открывается.
5. Фидо выходит на улицу.
6. Фидо делает свои дела.
   - 6.1 Дверь автоматически закрывается.
   - 6.2. Фидо лает, чтобы вернуться в дом.
   - 6.3. Распознаватель «слышит» лай (повторно).
   - 6.4. Распознаватель отправляет двери приказ открыться.
   - 6.5. Дверь открывается (повторно).
7. Фидо заходит в дом.
8. Дверь автоматически закрывается.

Альтернативные пути

2.1. Тодд или Джина слышат лай.

3.1. Тодд или Джина нажимают кнопку на пульте.

6.3.1. Тодд или Джина слышат лай (повторно).

6.4.1. Тодд или Джина нажимают кнопку на пульте.

Теперь шаги, в которых задействован распознаватель лая, находятся на основном, а не на альтернативном пути.

Тодд и Джина хотят относительно редко использовать пульт, поэтому шаги, относящиеся к его использованию, лучше вынести в альтернативный путь.

От начала до конца

С новыми альтернативными путями в нашем новом варианте использования существует несколько последовательностей шагов, при которых Фидо выходит на улицу, а потом возвращается обратно. Ниже показана лишь одна из возможных последовательностей прохождения варианта:

Выберем этот альтернативный путь, чтобы Тодд или Джина открыли дверь с пульта.

Собачья дверь для Тодда и Джины, версия 2.3

Как должна работать дверь

Основной путь	Альтернативные пути
1. Фидо лает, чтобы его выпустили.	
2. Распознаватель «слышит» лай.	2.1. Распознаватель «слышит» лай.
3. Распознаватель отправляет двери приказ открыться.	3.1. Распознаватель отправляет двери приказ открыться.
4. Дверь открывается.	
5. Фидо выходит на улицу.	
6. Фидо делает свои дела.	
6.1 Дверь автоматически закрывается.	
6.2. Фидо лает, чтобы вернуться в дом.	
6.3. Распознаватель «слышит» лай (повторно).	6.3.1. Распознаватель «слышит» лай (повторно).
6.4. Распознаватель отправляет двери приказ открыться.	6.4.1. Распознаватель отправляет двери приказ открыться.
6.5. Дверь открывается (повторно).	
7. Фидо заходит в дом.	
8. Дверь автоматически закрывается.	

Прохождение этого варианта использования всегда начинается с шага 1.

Здесь выбирается дополнительный отрезок пути, при котором Фидо не успевает вернуться в дом.

Дверь снова открывается Тоддом или Джиной на альтернативном пути.

Перемещение по стрелкам определяет конкретную последовательность прохождения варианта использования. Такая последовательность называется **сценарием**. Обычно в одном варианте использования существует несколько сценариев.

Прохождение всегда завершается на шаге 8, когда Фидо снова находится в доме.

Часто задаваемые вопросы

В: Я понимаю смысл основного пути варианта использования, но нельзя ли еще раз объяснить, что это за альтернативный путь?

О: Альтернативный путь — один или несколько шагов, которые в варианте использования не являются обязательными или образуют альтернативную последовательность прохождения варианта. Альтернативные пути могут включать дополнительные шаги, добавленные в основной путь, или же определяют совершенно иную последовательность достижения цели, чем в основном пути.

В: Значит, если Фидо выходит наружу и не может вернуться в дом, эта ситуация является частью альтернативного пути?

О: Точно. В нашем варианте использования альтернативный путь состоит из шагов 6.1, 6.2, 6.3, 6.4 и 6.5. Это дополнительные шаги, через которые может проходить система, и они нужны только в том случае, если Фидо не может вернуться в дом. Но они принадлежат альтернативному пути, потому что Фидо не всегда застревает снаружи, — система может перейти от шага 6 прямо к шагу 7.

В: И для их обозначения используется нумерация второго уровня — типа 6.1 и 6.2?

О: Точно. Поскольку альтернативный путь с дополнительными шагами — это всего лишь последовательность шагов, которые могут происходить *как часть* другого шага на основном пути варианта использования. В ситуации, когда Фидо застревает на улице, используются шаги основного пути 6 и 7; соответственно нумерация шагов альтернативного начинается с 6.1 и следует до 6.5; эти шаги являются необязательной частью шага 6.

В: И как называется ситуация с наличием двух разных путей через некоторую часть варианта использования?

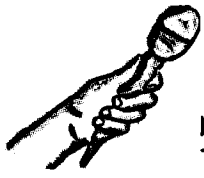
О: На самом деле это всего лишь другая разновидность альтернативного пути. На одном пути Тодд или Джина слышат лай Фидо и открывают дверь, на другом распознаватель «слышит» лай и открывает дверь. Система спроектирована так, что дверь открывается только одним из двух способов — либо с пульта, либо распознавателем, но не обоими сразу.

В: Возможно ли существование нескольких альтернативных путей в одном сценарии?

О: Безусловно. Вариант может содержать альтернативные пути с дополнительными шагами, а от начального условия к конечному можно перейти разными способами. Возможны даже альтернативные пути, приводящие к более раннему завершению варианта использования... Но для собачьей двери Тодда и Джины нам такие сложности не нужны.

Полный путь прохождения варианта использования, от первого шага до последнего, называется сценарием.

Обычно варианты использования содержат несколько разных сценариев, но эти сценарии всегда приводят к одной цели пользователя.



Откровенно о вариантах использования

Интервью недели: Признания Альтернативного Пути

HeadFirst: Привет, Альтернативный Путь. Говорят, вы жалуетесь на тяжелую жизнь. Расскажите, что происходит?

Альтернативный Путь: Просто я иногда не ощущаю своей причастности к общему делу. Без меня невозможно создать сколько-нибудь приличный вариант использования, но, похоже, в большинстве случаев меня попросту игнорируют.

HeadFirst: Игнорируют? Но вы же сами сказали, что вы являетесь частью почти любого варианта использования. Это значит, что вам отводится важная роль.

Альтернативный Путь: Конечно, так может показаться. Но даже когда я присутствую в варианте использования, меня можно обойти по другой последовательности шагов... Будто меня там вовсе нет – кому же это понравится?

HeadFirst: А можно конкретный пример?

Альтернативный Путь: Пару дней назад я работал в составе варианта использования по покупке компакт-дисков в новом интернет-магазине. Это было так захватывающе... А потом оказалось, что я обрабатываю ситуацию с недействительной кредитной картой покупателя.

HeadFirst: Но это же важная работа! В чем проблема?

Альтернативный Путь: Наверное, важная, но я постоянно оставался в стороне! Народ приходит, покупает диски, их карты нормально проходят проверку. Хотя я и был частью варианта использования, я не входил в основные сценарии.

HeadFirst: Ага, понятно. Значит, пока чья-то карта не будет отклонена, вы не участвуете в происходящем?

Альтернативный Путь: Точно! Конечно, специалисты по финансам и безопасности меня хвалят, они только и говорят о том, насколько я ценен для компании, – но кому захочется все время сидеть на месте без дела?

HeadFirst: Кажется, я начинаю понимать. Но ведь вы все равно играете свою роль в варианте использования, верно? Даже если к вам обращаются относительно редко, время от времени это все же происходит.

Альтернативный Путь: Да, это так; все мы работаем для достижения общей цели. Я как-то не сообразил, что могу играть важную роль в варианте использования, хотя и остаюсь в стороне.

HeadFirst: Поймите, вариант использования без вас будет неполным.

Альтернативный Путь: Да, пункты 3.1 и 4.1 мне это часто говорили. Конечно, их альтернативный путь предназначен для клиентов, у которых уже есть учетная запись в системе, так что они используются постоянно. Им легко говорить!

HeadFirst: Не покидайте нас, Альтернативный Путь. Мы знаем, что вы являетесь важной частью варианта использования!

Возьми в руку карандаш

Сколько сценариев в варианте использования Тодда и Джины?

Сколько разных способов прохождения варианта использования Тодда и Джины вы сможете найти? Помните, что в одних случаях выбирается один из нескольких альтернативных путей, а в других часть альтернативного пути может пропускаться.

Собачья дверь для Тодда и Джины, версия 2.3

Как должна работать дверь

Основной путь	Альтернативные пути
1. Фидо лает, чтобы его выпустили.	
2. Распознаватель «слышит» лай.	2.1. Распознаватель «слышит» лай.
3. Распознаватель отправляет двери приказ открыться.	3.1. Распознаватель отправляет двери приказ открыться.
4. Дверь открывается.	
5. Фидо выходит на улицу.	
6. Фидо делает свои дела.	
6.1 Дверь автоматически закрывается.	
6.2. Фидо лает, чтобы вернуться в дом.	
6.3. Распознаватель «слышит» лай (повторно).	6.3.1. Распознаватель «слышит» лай (повторно).
6.4. Распознаватель отправляет двери приказ открыться.	6.4.1. Распознаватель отправляет двери приказ открыться.
6.5. Дверь открывается (повторно).	
7. Фидо заходит в дом.	
8. Дверь автоматически закрывается.	

Ответы на следующей странице

Чтобы вам было проще начать, мы записали последовательность прохождения для варианта, обозначенного стрелками.

1. 1, 2.1, 3.1, 4, 5, 6, 6.1, 6.2, 6.3.1, 6.4.1, 6.5, 7, 8
2. _____
3. _____
4. _____
5. _____
6. _____
7. _____
8. _____

Возможно, некоторые поля останутся незаполненными.

Возьми в руку карандаш
Решение

Сколько сценариев в варианте использования Тодда и Джины?

Сколько разных способов прохождения варианта использования Тодда и Джины вы сможете найти? Помните, что в одних случаях выбирается один из нескольких альтернативных путей, а в других часть альтернативного пути может пропускаться.

Собачья дверь для Тодда и Джины, версия 2.3

Как должна работать дверь

Основной путь	Альтернативные пути
1. Фидо лает, чтобы его выпустили.	
2. Распознаватель «слышит» лай.	2.1. Распознаватель «слышит» лай.
3. Распознаватель отправляет двери приказ открыться.	3.1. Распознаватель отправляет двери приказ открыться.
4. Дверь открывается.	
5. Фидо выходит на улицу.	
6. Фидо делает свои дела.	
6.1 Дверь автоматически закрывается.	
6.2. Фидо лает, чтобы вернуться в дом.	
6.3. Распознаватель «слышит» лай (повторно).	6.3.1. Распознаватель «слышит» лай (повторно).
6.4. Распознаватель отправляет двери приказ открыться.	6.4.1. Распознаватель отправляет двери приказ открыться.
6.5. Дверь открывается (повторно).	
7. Фидо заходит в дом.	
8. Дверь автоматически закрывается.	

Основной путь варианта использования.

В этих двух вариантах не используется необязательный альтернативный путь, в котором Фидо застревает снаружи.

1. 1, 2.1, 3.1, 4, 5, 6, 6.1, 6.2, 6.3.1, 6.4.1, 6.5, 7, 8
2. 1, 2, 3, 4, 5, 6, 7, 8
3. 1, 2.1, 3.1, 4, 5, 6, 7, 8
4. 1, 2.1, 3.1, 4, 5, 6, 6.1, 6.2, 6.3, 6.4, 6.5, 7, 8

Если выбирается шаг 2.1, также будет выбран шаг 3.1.

Если выбирается шаг 6.3.1, то также будет выбран шаг 6.4.1.

5. 1, 2, 3, 4, 5, 6, 6.1, 6.2, 6.3.1, 6.4.1, 6.5, 7, 8
6. 1, 2, 3, 4, 5, 6, 6.1, 6.2, 6.3, 6.4, 6.5, 7, 8
7. <nothing else>
8. <nothing else>

Готовимся к написанию кода...

Итак, наш вариант использования готов, и мы разобрали все возможные сценарии использования собачьей двери. Теперь можно переходить к написанию кода для реализации новых требований Тодда и Джины. Давайте разберемся, что же, собственно, необходимо сделать...

Думаю, нужно проверить список требований по новому варианту использования. Если требования Тодда и Джины изменились, то и наш список требований тоже мог измениться, верно?

При каждом изменении варианта использования необходимо заново проверить требования.

Не забывайте, что главная цель хорошего варианта использования – это получение хороших требований. Если ваш вариант использования изменился, это может означать, что и требования тоже изменились. Давайте проанализируем требования и посмотрим, не нужно ли в них что-нибудь добавить.

Собачья дверь для Тодда и Джины, версия 2.2

Список требований

1. **Высота дверного проема должна быть не менее 12 дюймов.**
2. **Кнопка на пульте открывает дверь, если она закрыта, и закрывает ее, если дверь открыта.**
3. **После того, как дверь откроется, она должна автоматически закрываться (если не была закрыта до этого).**

Запишите все дополнительные требования, которые были обнаружены в ходе работы над вариантами для новой собачьей двери на с. 158.

Доработка списка требований

С учетом появления двух новых альтернативных путей необходимо добавить в список пару новых требований. В приведенном ниже списке вычеркнуты шаги, которые уже обеспечиваются существующими требованиями. Следовательно, нам предстоит дополнить список несколькими новыми пунктами:

Собачья дверь для Тодда и Джины, версия 2.3

Как должна работать дверь

Основной путь	Альтернативные пути
1. ~~Фидо лает, чтобы его выпустили.~~	
2. Распознаватель «слышит» лай.	~~2.1. Распознаватель „слышит" лай.~~
3. Распознаватель отправляет двери приказ открыться.	~~3.1. Распознаватель отправляет двери приказ открыться.~~
~~4. Дверь открывается.~~	
~~5. Фидо выходит на улицу.~~	
~~6. Фидо делает свои дела.~~	
~~6.1 Дверь автоматически закрывается.~~	
~~6.2 Фидо лает, чтобы вернуться в дом.~~	
6.3. Распознаватель «слышит» лай (повторно).	6.3.1. ~~Распознаватель «слышит» лай (повторно).~~
6.4. Распознаватель отправляет двери приказ открыться.	~~6.4.1. Распознаватель отправляет двери приказ открыться.~~
~~6.5. Дверь открывается (повторно).~~	
~~7. Фидо заходит в дом.~~	
~~8. Дверь автоматически закр~~	

В действительности здесь задействованы два требования: умение «слышать» лай и открывание двери.

Многие шаги основного пути были реализованы в главе 2.

Стоит напомнить, что в предыдущей главе шаги альтернативного пути были частью основного пути варианта использования...

...так что мы уже обработали требования, относящиеся к ним.

Формально эти шаги отличаются от 2 и 3, но требования остаются теми же.

Собачья дверь для Тодда и Джины, версия 2.3

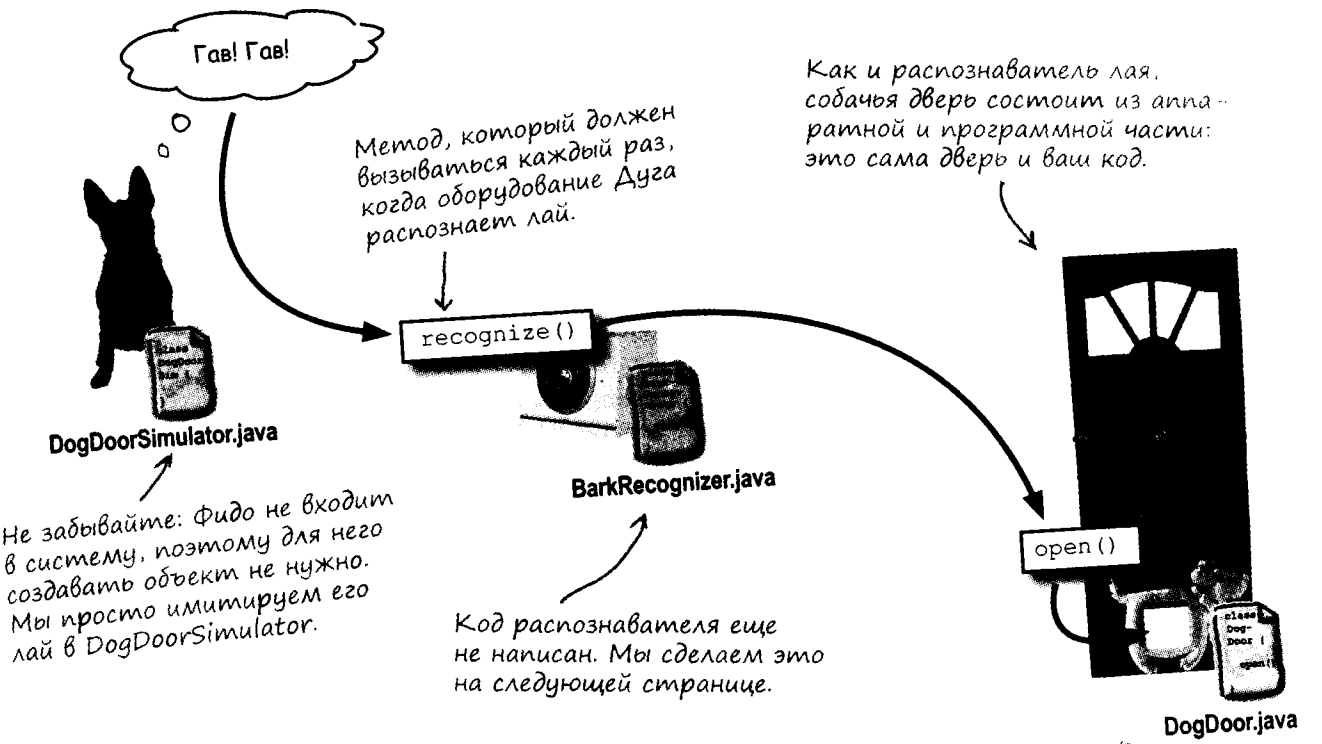
Список требований

1. Высота дверного проема должна быть не менее 12 дюймов.
2. Кнопка на пульте открывает дверь, если она закрыта, и закрывает ее, если дверь открыта.
3. После того, как дверь откроется, она должна автоматически закрываться (если не была закрыта до этого).
4. Распознаватель должен уметь «слышать» собачий лай.
5. Если распознаватель «слышит» лай, он должен открыть дверь.

Два новых требования, которые необходимо добавить в список.

А теперь можно снова заняться программированием

С новыми требованиями появляется новый код. Нам понадобятся: лающая собака; распознаватель, который обнаруживает ее лай; и дверь, которая открывается:



Не забывайте: Фидо не входит в систему, поэтому для него создавать объект не нужно. Мы просто имитируем его лай в DogDoorSimulator.

Код распознавателя еще не написан. Мы сделаем это на следующей странице.

С кодом этого класса ничего делать не нужно. Метод open(), который будет вызываться опознавателем, уже написан, так что код изменять вообще не придется.

И хотя мы еще работаем над тем, чтобы привести программу в соответствие с пожеланиями заказчика, это хороший признак качественного проектирования. Хорошая работа!

Кто сказал «гав»?

BarkRecognizer.java

Когда оборудование Дуга «слышит» лай, должен выполняться некий программный код. Давайте создадим класс **BarkRecognizer** и напишем метод, который будет реагировать на собачий лай:

```
public class BarkRecognizer {

  private DogDoor door;

  public BarkRecognizer(DogDoor door) {
    this.door = door;
  }

  public void recognize(String bark) {
    System.out.println("   BarkRecognizer: Heard a '" +
        bark + "'");
    door.open();
  }
}
```

В этой переменной хранится собачья дверь, с которой связан текущий распознаватель.

Класс BarkRecognizer должен знать, какую дверь он будет открывать.

Каждый раз, когда оборудование распознает лай Фидо, оно будет вызывать этот метод.

Нужно лишь вывести сообщение об услышанном лае...

...а затем открыть собачью дверь.

Часто задаваемые вопросы

В: И это все? Что-то класс BarkRecognizer делает не много.

О: Да, в этой версии — не много. Требования просты (когда собака лает, открыть дверь), поэтому ваш код тоже получается простым. Каждый раз, когда оборудование обнаруживает лай, оно вызывает метод **recognize()** нашего нового класса **BarkRecognizer**, и мы открываем собачью дверь. Помните: код должен быть настолько простым, насколько это возможно; не добавляйте сложность, если без нее можно обойтись.

В: А что произойдет, если залает другая собака? Разве класс BarkRecognizer не должен проверить, что лает именно Фидо, прежде чем открывать дверь?

О: Очень интересный вопрос! Класс **BarkRecognizer** воспринимает любой лай, но ведь мы не хотим открывать дверь посторонним собакам, верно? Вероятно, эту проблему нужно будет решить позднее. А может, вам стоит поразмыслить над ней, пока мы будем заниматься тестированием.

Думаю, теперь у нас есть все необходимое. Давайте протестируем класс BarkRecognizer и посмотрим, удастся ли нам снова угодить Тодду и Джине.

Сначала убедимся в том, что мы реализовали новые требования Тодда и Джины:

Еще одно требование, относящееся к оборудованию Дуга. А пока мы воспользуемся тестовой программой, чтобы вызвать реакцию распознавателя и протестировать написанный код.

Собачья дверь для Тодда и Джины, версия 2.3

Список требований

1. **Высота дверного проема должна быть не менее 12 дюймов.**
2. **Кнопка на пульте открывает дверь, если она закрыта, и закрывает ее, если дверь открыта.**
3. **После того, как дверь откроется, она должна автоматически закрываться (если не была закрыта до этого).**
4. **Распознаватель должен уметь «слышать» собачий лай.**
5. **Если распознаватель «слышит» лай, он должен открыть дверь.**

Код, который мы только что написали... Каждый раз, когда распознаватель «слышит» лай, он открывает дверь.

Хмм... Наш распознаватель на самом деле ничего не распознает. Он просто открывает дверь на ЛЮБОЙ лай. Вскоре мы вернемся к этой проблеме.

Проверка новой двери

Варианты использования, требования, код — все это вело нас к этому торжественному моменту. Давайте проверим, все ли работает так, как должно.

DogDoorSimulator.java

1 Обновление исходного кода DogDoorSimulator:

```
public class DogDoorSimulator {

  public static void main(String[] args) {
    DogDoor door = new DogDoor();
    BarkRecognizer recognizer = new BarkRecognizer(door);
    Remote remote = new Remote(door);

    // Имитация лая для распознавателя
    System.out.println("Fido starts barking.");
    recognizer.recognize("Woof");

    System.out.println("\nFido has gone outside...");

    System.out.println("\nFido's all done...");

    try {
      Thread.currentThread().sleep(10000);
    } catch (InterruptedException e) { }

    System.out.println("...but he's stuck outside!");

    // Имитация повторного лая
    System.out.println("Fido starts barking.");
    recognizer.recognize("Woof");

    System.out.println("\nFido's back inside...");
  }
}
```

Создаем экземпляр BarkRecognizer, связываем его с дверью и приказываем отслеживать собачий лай.

Реального устройства у нас нет, мы просто имитируем распознавание лая оборудованием.*

Здесь вступает в действие наш новый класс BarkRecognizer.

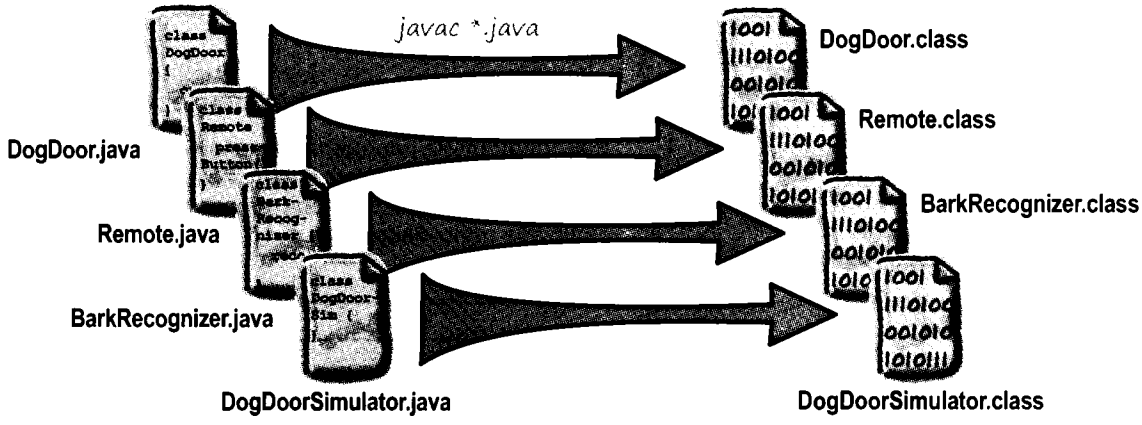
Здесь имитируется промежуток времени, проведенный Фидо на дворе.

Тестируем процесс при нахождении Фидо на улице — просто чтобы убедиться в том, что все работает как положено.

Обратите внимание: на этот раз Тодд и Джина не нажимают кнопку на пульте.

*Авторы искренне хотели приложить к книге устройство, распознающее собачий лай... Но отдел маркетинга утверждает, что никто не станет покупать книгу стоимостью $299.95. Ничего не поделаешь!

2 **Перекомпилируйте весь исходный код Java в классы.**



3 **Запустите программу и проследите за тем, как работает автоматизированная собачья дверь.**

Проходит несколько секунд, Фидо играет на улице.

```
File Edit Window Help YouBarkLikeAPoodle
%java DogDoorSimulator
Fido starts barking.
  BarkRecognizer: Heard a 'Woof'
The dog door opens.

Fido has gone outside...

Fido's all done...
...but he's stuck outside!
Fido starts barking.
  BarkRecognizer: Heard a 'Woof'
The dog door opens.

Fido's back inside...
```

МОЗГОВОЙ ШТУРМ

У нашего кода имеется серьезный недостаток, который проявляется в тестовой программе. Удастся ли вам найти его? Как бы вы исправили этот недостаток?

Возьми в руку карандаш

Какой из сценариев мы тестируем?

Сможете ли вы определить, какой сценарий из варианта использования тестируется в нашей программе? Запишите последовательность действий, выполняемых тестовой программой (при необходимости вернитесь к варианту использования на с. 153):

_____

Мозговой штурм

У нашего кода имеется серьезный недостаток, который проявляется в тестовой программе. Удастся ли вам найти его? Как бы вы исправили этот недостаток?

Возьми в руку карандаш
Решение

Какой из сценариев мы тестируем?

А вы правильно определили, какой сценарий из варианта использования тестируется в нашей программе? Ниже приведена последовательность действий из варианта использования на с. 153:

1, 2, 3, 4, 5, 6, 6.1, 6.2, 6.3, 6.4, 6.5, 7, 8

А вы нашли, что не так в последней версии собачьей двери?

В новой версии дверь не закрывается автоматически!

В вариантах, в которых Тодд или Джина нажимают кнопку на пульте, выполняется следующий код:

```
public void pressButton() {
  System.out.println("Pressing the remote control button...");
  if (door.isOpen()) {
    door.close();
  } else {
    door.open();

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      public void run() {
        door.close();
        timer.cancel();
      }
    }, 5000);
  }
}
```

Когда Тодд или Джина нажимают кнопку на пульте, этот код также запускает таймер для автоматического закрывания двери.

Этот таймер ожидает 5 секунд, а потом отправляет двери команду закрыться.

Remote.java

А в **BarkRecognizer** мы открываем дверь, но не закрываем ее:

```
public void recognize(String bark) {
  System.out.println("   BarkRecognizer: " +
    "Heard a '" + bark + "'");
  door.open();
}
```

Дверь открывается, но не закрывается.

BarkRecognizer.java

Похоже, владелец фирмы Дуг совершенно точно знает, что вам нужно сделать.

Даже я знаю, что нужно сделать. Добавь в BarkRecognizer таймер, как в классе пульта, и все снова заработает. Тодд и Джина ждут!

А что ВЫ думаете об идее Дуга?

Мне не нравится идея Дуга. Я не хочу размещать одинаковый код в классах пульта и распознавателя лая.

Конечно, дублирование кода нежелательно. Но где должен находиться код, закрывающий дверь?

Вообще-то операция закрывания двери должна выполняться самой дверью, а не пультом дистанционного управления и не распознавателем. Почему бы не приказать экземпляру DogDoor закрыть себя?

Пусть дверь всегда закрывается автоматически.

Так как Джина не хочет, чтобы дверь хоть когда-нибудь оставалась открытой, дверь должна *всегда* закрываться автоматически. Следовательно, код, автоматически закрывающий дверь, можно переместить в класс **DogDoor**. В этом случае независимо от того, *почему* открылась дверь, она всегда будет автоматически закрываться.

И хотя это решение относится к структуре системы, оно также направлено на выполнение пожеланий заказчика. Не забывайте, что в ходе работы над функциональностью системы стоит попутно думать о хорошей структуре кода.

Обновление класса двери

Код, закрывающий дверь, перемещается из класса **Remote** в класс **DogDoor**:

DogDoor.java

Также необходимо добавить директивы import для java.util.Timer и java.util.TimerTask.

```
public class DogDoor {
  public void open() {
    System.out.println("The dog door opens.");
    open = true;

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      public void run() {
        close();
        timer.cancel();
      }
    }, 5000);
  }

  public void close() {
    System.out.println("The dog door closes.");
    open = false;
  }
}
```

Этот код прежде находился в Remote.java.

Теперь дверь будет закрываться автоматически... Даже если добавятся новые устройства, которые могут ее открывать. Отлично!

Упрощение класса пульта

Перенесенный код исключается из класса **Remote**, потому что дверь теперь автоматически закрывает себя:

```
public void pressButton() {
  System.out.println("Pressing the remote control button...");
  if (door.isOpen()) {
    door.close();
  } else {
    door.open();

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      public void run() {
        door.close();
        timer.cancel();
      }
    }, 5000);
  }
}
```

Remote.java

Последний пробный запуск

С момента знакомства с Тоддом и Джиной мы внесли немало изменений в разработанную для них дверь. Давайте протестируем приложение и посмотрим, все ли в нем работает правильно. Внесите изменения в **Remote.java** и **DogDoor.java**, обеспечивающие автоматическое закрытие двери, снова откомпилируйте классы и запустите тестовую программу.

```
File Edit Window Help PestControl
%java DogDoorSimulator
Fido starts barking.
  BarkRecognizer: Heard a 'Woof'
The dog door opens.

Fido has gone outside...

Fido's all done...
The dog door closes.
...but he's stuck outside!

Fido starts barking.
  BarkRecognizer: Heard a 'Woof'
The dog door opens.

Fido's back inside...
The dog door closes.
```

Да! Дверь теперь закрывается автоматически.

Мозговой штурм

А что произойдет, если Тодд и Джина захотят, чтобы дверь дольше оставалось открытой? Или закрывалась быстрее? Удастся ли вам изменить класс DogDoor таким образом, чтобы промежуток времени до автоматического закрывания двери мог определяться заказчиком?

Иногда изменение требований выявляет скрытые проблемы, о которых вы даже не подозревали.

Изменения происходят постоянно, и ваша система должна улучшаться каждый раз, когда вы над ней работаете.

Возьми в руку карандаш

Сформулируйте свой принцип проектирования!

В этой главе использовался важный принцип проектирования, относящийся к дублированию кода и автоматическому закрыванию двери. Попробуйте обобщить и сформулировать этот принцип:

Принцип проектирования

_____

_____

_____

В этой главе ответ не приводится, но мы еще вернемся к нему позднее. А пока — постарайтесь сделать обоснованное предположение!

Новые Инструменты ООАП

Вы узнали много нового, и теперь пришло время расширить ваш инструментарий ООАП. Эта страница напомнит вам, о чем говорилось в этой главе.

Требования

Хорошие требования помогают привести вашу систему в соответствие с ожиданиями заказчика.

Проследите за тем, чтобы требования охватывали все шаги всех вариантов использования вашей системы.

Варианты использования помогут вам узнать о вашей системе то, о чем забыл упомянуть заказчик.

Анализ вариантов использования выявляет неполные и пропущенные требования, которые вам, возможно, придется добавить в систему.

Требования всегда изменяются (и растут) со временем.

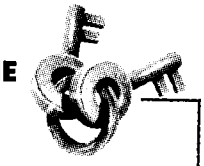
В этой главе вы узнали всего один новый принцип, но очень важный!

Принципы ООАП

Инкапсулируйте то, что изменяется.

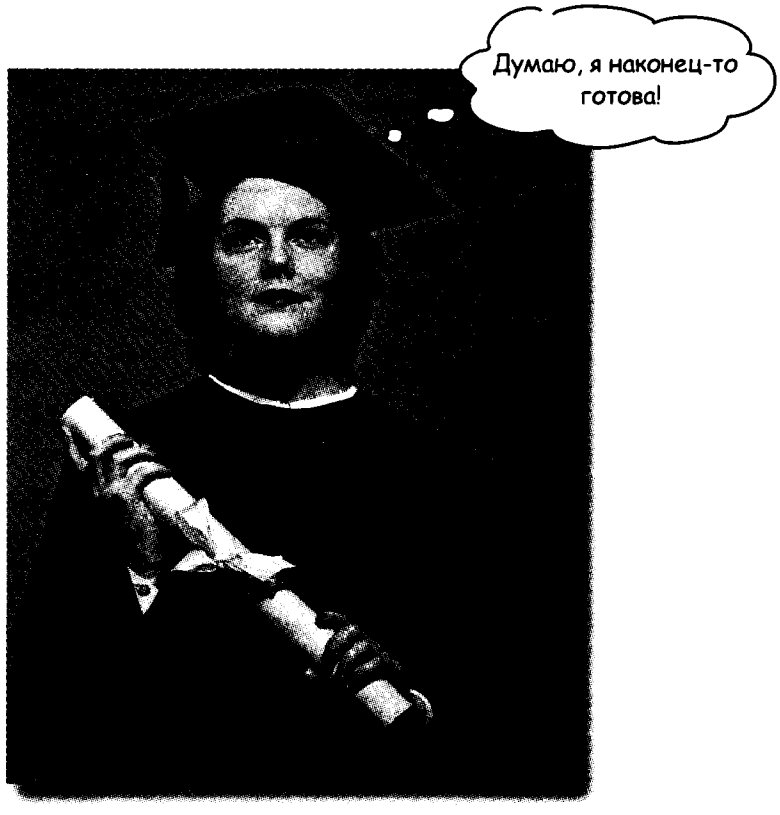
Инкапсуляция помогла нам осознать, что собачья дверь должна сама закрывать себя. Мы отделили поведение двери от остального кода приложения.

КЛЮЧЕВЫЕ МОМЕНТЫ



- Требования всегда **изменяются** в ходе работы над проектом.
- С изменением требований система должна развиваться, чтобы адаптироваться к новым требованиям.
- Если вашей системе потребуется работать по-новому, начните с обновления варианта использования.
- **Сценарием** называется конкретный путь прохождения варианта использования, от начала и до конца.
- Один вариант использования может иметь несколько сценариев — при условии, что все сценарии имеют общую цель.
- **Альтернативные пути** состоят из шагов, которые выполняются только в некоторых случаях или образуют другие пути прохождения варианта использования.
- Если шаг является необязательным для работы системы или образует альтернативный путь в системе, используйте многоуровневую нумерацию вида 3.1, 4.1, 5.1 или 2.1.1, 2.2.1 и 2.3.1.
- Практически всегда следует избегать дублирования кода. Дублирование превращает сопровождение кода в сущий кошмар и обычно свидетельствует о недостатках в проектировании системы.

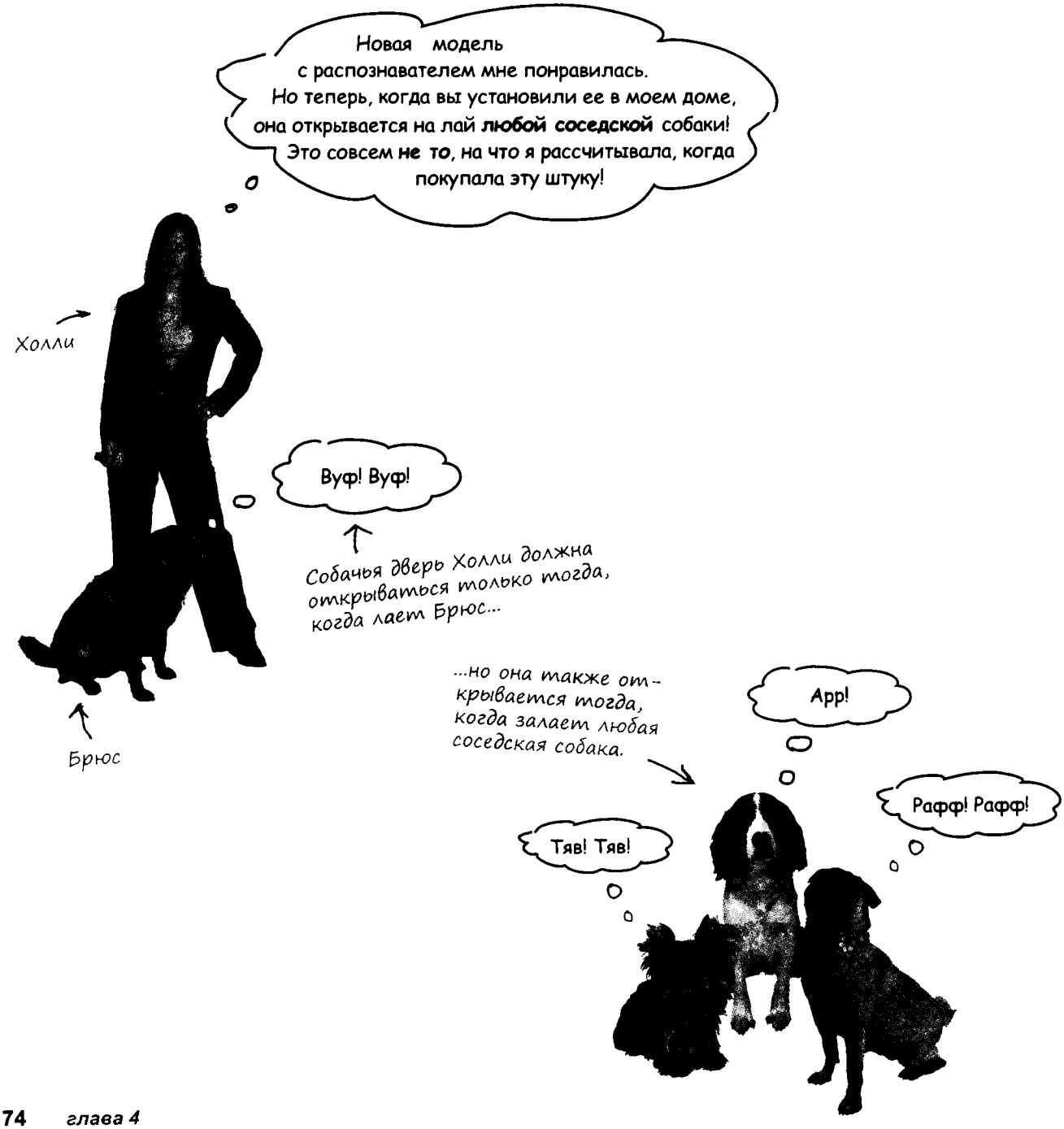
Программы для реального мира



Пора переходить к реальным приложениям. Даже если ваше приложение идеально работает на машине разработки, этого недостаточно; приложения должны работать тогда, когда их используют реальные люди. В этой главе говорится о том, как заставить вашу программу работать в реальном контексте. Вы узнаете, как текстологический анализ преобразует вариант использования, над которым вы работали, в классы и методы, которые делают то, что нужно вашим заказчикам. А когда работа будет завершена, вы сможете сказать: «Да, у меня получилось! Моя программа готова к жизни в реальном мире!»

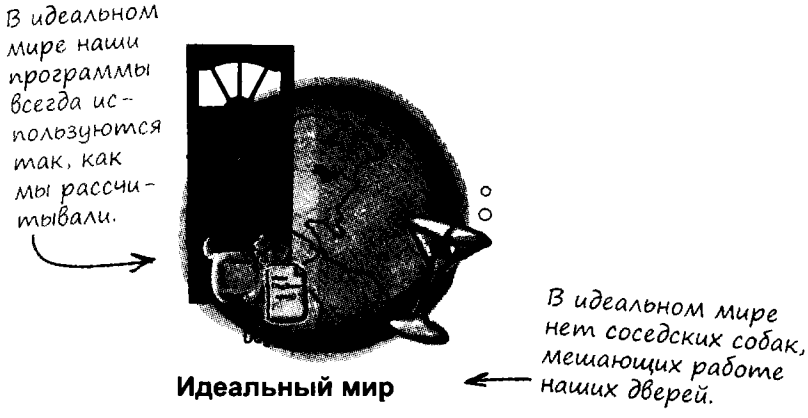
Одна собака, две собаки, три...

У фирмы Дуга дела идут хорошо. Версия собачьей двери, разработанная нами в главе 3, продается нарасхват... Но со временем покупатели начали жаловаться:



Программа работает в определенном контексте

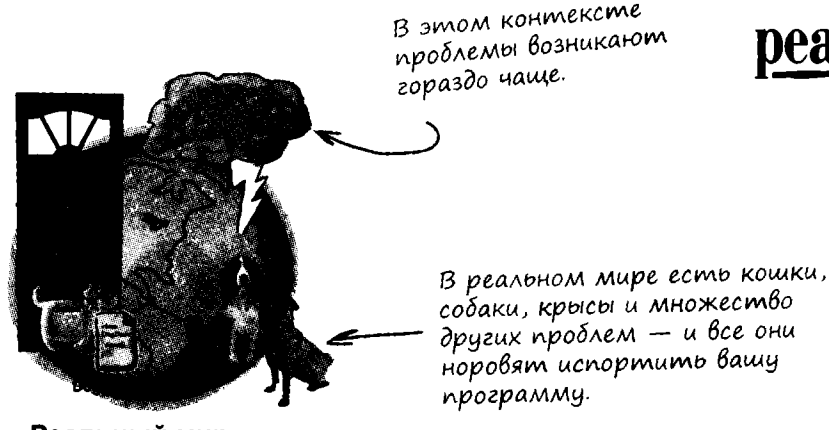
До сих пор мы писали условные программы, работающие «в вакууме», не обращая внимания на контекст, в котором программа выполняется. Иначе говоря, мы представляли свои программы следующим образом:



Идеальный мир

Анализ помогает обеспечить работу вашей системы в контексте реального мира.

Но наши программы должны **работать в реальном**, а не в идеальном мире. Это означает, что мы должны рассматривать свои программы в другом контексте:

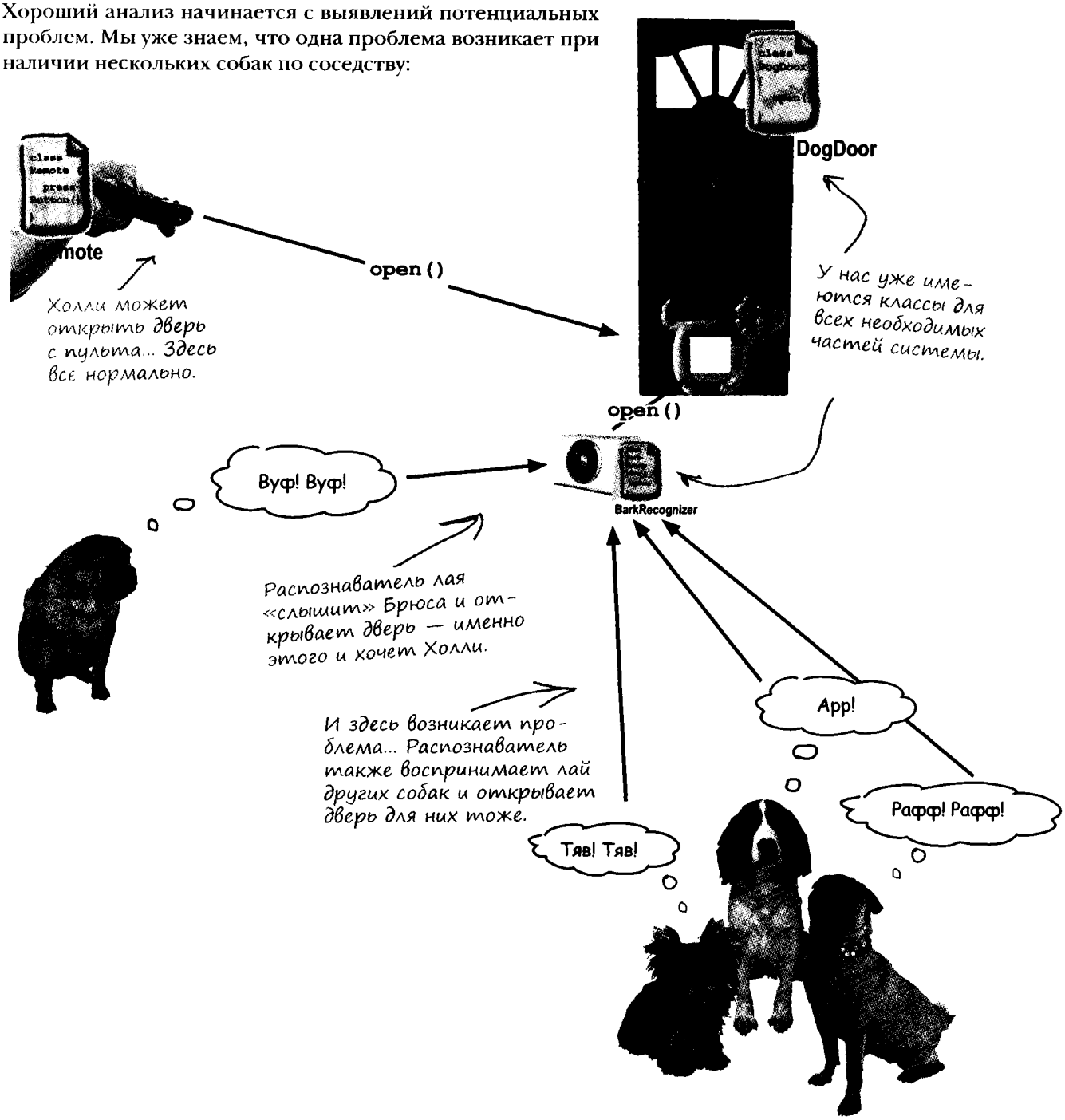


Реальный мир

Как убедиться в том, что ваши программы будут нормально работать и не сломаются от столкновения с реальным миром? В этом вам поможет **анализ**: выявление потенциальных проблем и их решение — *еще до того,* как ваше приложение попадет в реальный мир.

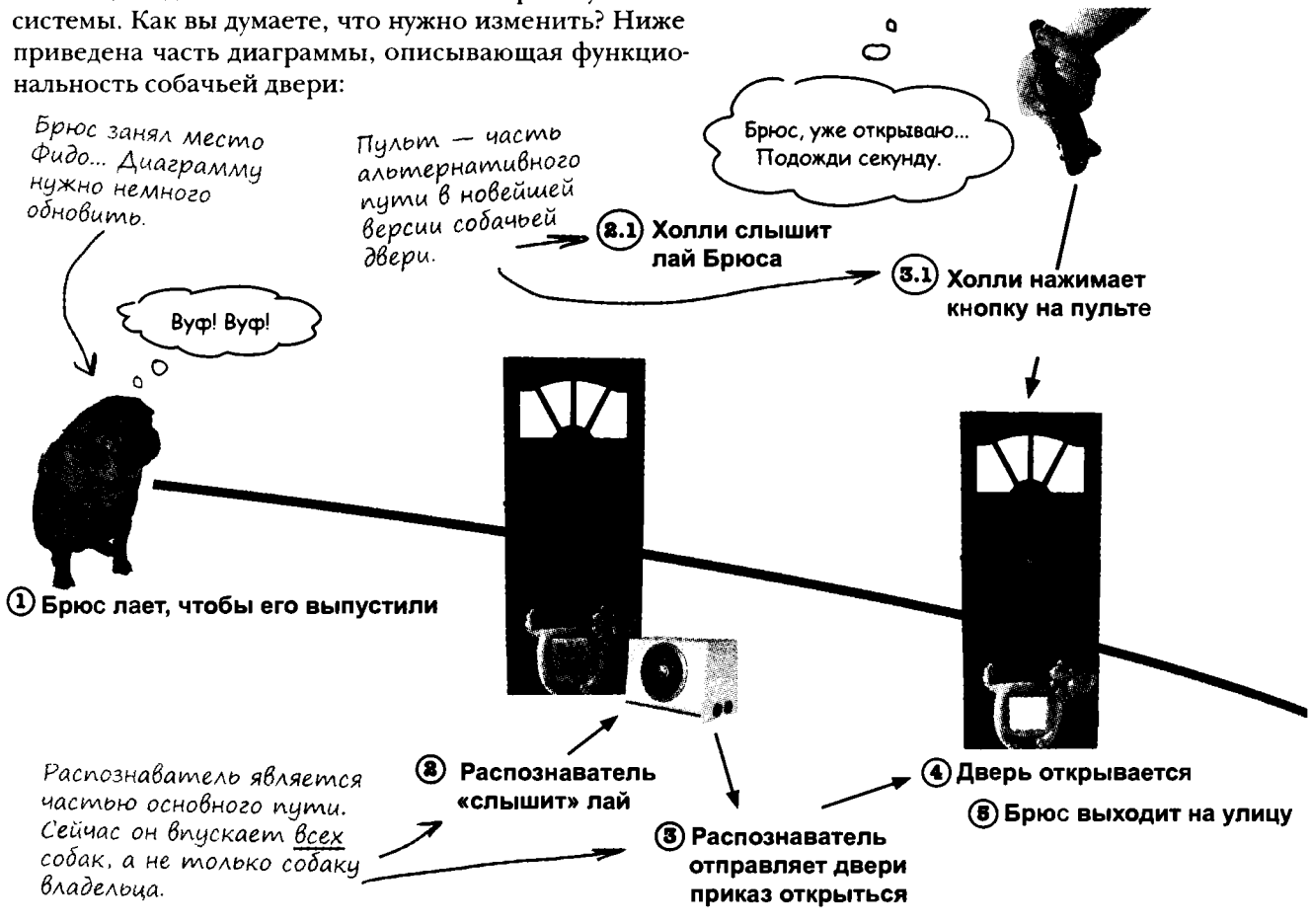
Выявление проблемы

Хороший анализ начинается с выявлений потенциальных проблем. Мы уже знаем, что одна проблема возникает при наличии нескольких собак по соседству:



Планирование решения

Похоже, мы должны внести изменения в работу нашей системы. Как вы думаете, что нужно изменить? Ниже приведена часть диаграммы, описывающая функциональность собачьей двери:



Возьми в руку карандаш

Что не так с этой диаграммой?

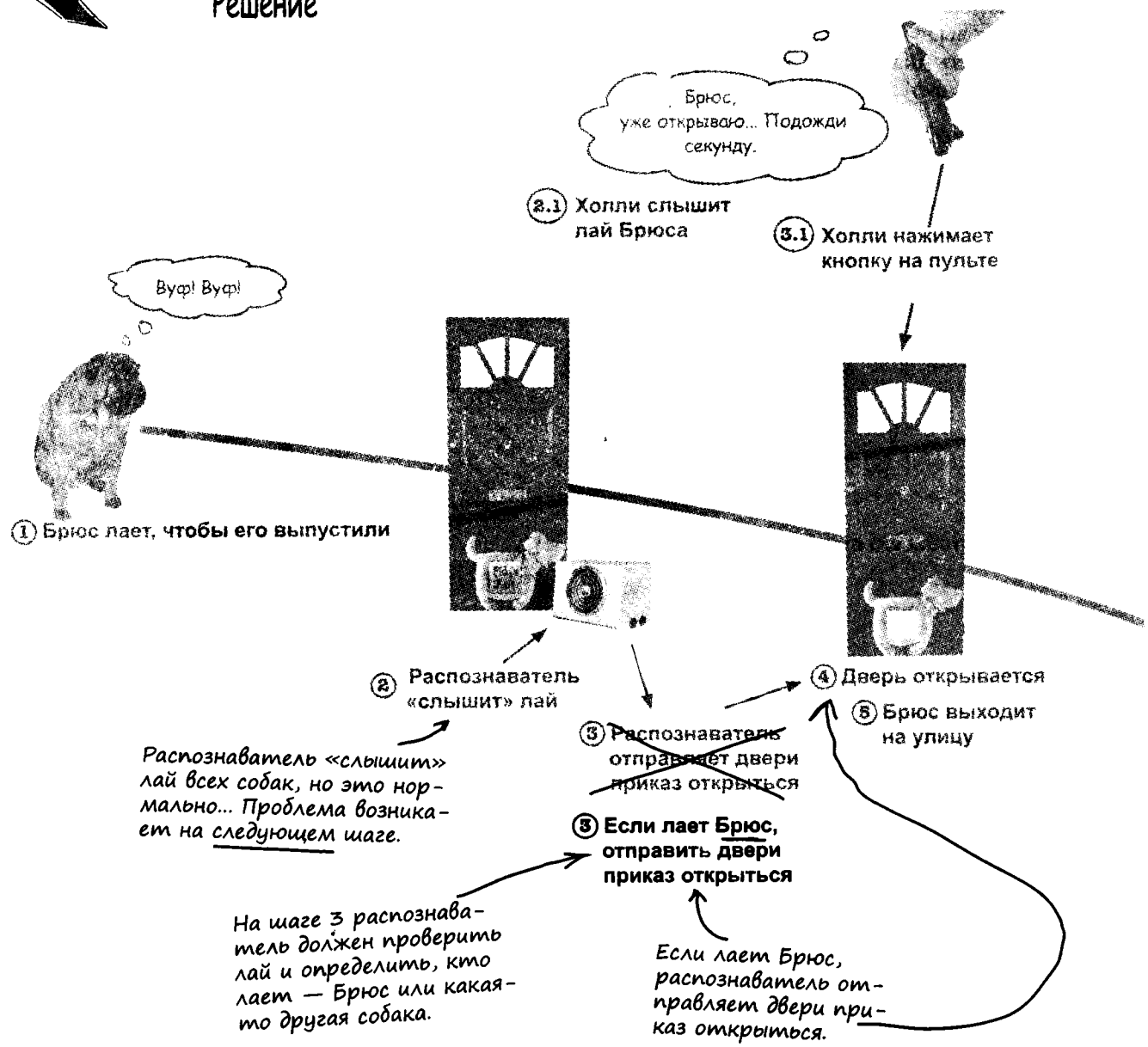
Как бы вы исправили обнаруженный недостаток? Вы можете добавлять новые шаги, удалять и изменять существующие шаги... делайте все, что считаете нужным. Запишите внизу то, что нужно изменить, а затем сделайте соответствующие пометки на диаграмме.

_____

_____

Возьми в руку карандаш
Решение

Что не так с этой диаграммой?



Часто задаваемые вопросы

В: У меня получилось другое решение. Значит ли это, что оно неправильно?

О: Нет — при условии, что ваше решение не позволяет открывать дверь другим собакам, кроме Брюса. Именно поэтому бывает так сложно говорить о программах: задача имеет много решений вместо одного «правильного» решения.

В: В своем решении я преобразовал шаг 4 исходного варианта использования в два шага (вместо простой замены). Что я сделал не так?

О: Ничего. Задача имеет много возможных решений — и каждое из этих решений может быть записано в варианте использования многими разными способами. Если вы используете несколько шагов, но вариант с лаем других собак успешно обрабатывается, ваш вариант использования вполне пригоден.

В: Выходит, варианты использования не так уж точны?

О: Напротив, варианты использования *очень* точны. Если ваш вариант использования не содержит подробного описания того, как должна работать система, вы рискуете упустить какое-нибудь важное требование и столкнуться с недовольством заказчика.

Но варианты использования не обязаны быть формальными; другими словами, ваш вариант использования не обязан быть похожим на наш или чей-то другой. Здесь важно, чтобы вариант использования был понятен для вас и вы могли объяснить его суть коллегам, начальству и заказчикам.

Записывайте свои варианты использования в том виде, который будет понятен вам, вашему руководству и заказчикам.

Анализ и варианты использования помогают объяснить заказчикам, начальству и другим разработчикам, как ваша система работает в контексте реального мира.

Мозговой штурм

В систему собачьей двери необходимо внести одно важное дополнение (кроме того, что показано на с. 178). Что это за дополнение?

Обновление варианта использования

Диаграмма работы собачьей двери изменилась, поэтому мы должны вернуться к варианту использования и обновить его новыми шагами. Затем на нескольких ближайших страницах мы определимся с изменениями, которые потребуется внести в код.

Все упоминания конкретных владельцев и собак исключены, так что теперь этот вариант подходит для всех клиентов Дуга.

Прощай, Фидо. Отныне твое место занимает «хозяйская собака».

Идеальная собачья дверь, версия 3.0

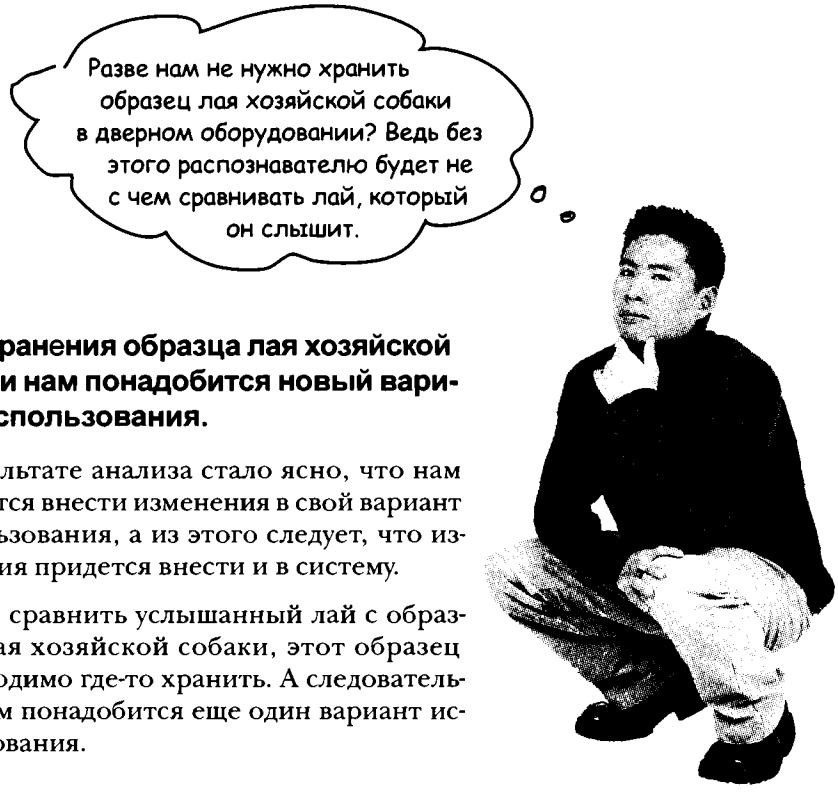
Как должна работать дверь

Основной путь	Альтернативные пути
1. Хозяйская собака лает, чтобы ее выпустили.	
2. Распознаватель «слышит» лай.	2.1. Хозяин слышит лай своей собаки.
	3.1. Хозяин нажимает кнопку на пульте.
3. Если лает хозяйская собака, распознаватель отправляет двери приказ открыться.	
4. Дверь открывается.	
5. Хозяйская собака выходит из дома.	
6. Хозяйская собака делает свои дела.	
6.1. Дверь автоматически закрывается.	
6.2. Хозяйская собака лает, чтобы вернуться в дом.	
6.3. Распознаватель «слышит» лай (повторно).	6.3.1. Хозяин слышит лай своей собаки (повторно).
6.4. Если лает хозяйская собака, распознаватель отправляет двери приказ открыться.	6.4.1. Хозяин нажимает кнопку на пульте.
6.5. Дверь открывается (повторно).	
7. Хозяйская собака заходит в дом.	
8. Дверь автоматически закрывается.	

Место Тодда, Джины или Холли теперь занимает «хозяин».

Этот шаг обновляется: теперь через дверь может входить и выходить только хозяйская собака.

Не забудьте изменить и этот отрезок.



Для хранения образца лая хозяйской собаки нам понадобится новый вариант использования.

В результате анализа стало ясно, что нам придется внести изменения в свой вариант использования, а из этого следует, что изменения придется внести и в систему.

Чтобы сравнить услышанный лай с образцом лая хозяйской собаки, этот образец необходимо где-то хранить. А следовательно, нам понадобится еще один вариант использования.

Возьми в руку карандаш

Добавьте новый вариант использования для хранения образца лая.

Нам понадобится вариант использования для сохранения образца лая хозяйской собаки; мы будем хранить его прямо в двери (специалисты из фирмы Дуга говорят, что с их уровнем технологии это несложно). Воспользуйтесь приведенным ниже шаблоном для создания нового варианта использования для этой задачи.

Идеальная собачья дверь, версия 3.0

Сохранение образца лая

1. _____
2. _____

Этот вариант использования состоит всего из двух шагов и не содержит никаких альтернативных путей.

Так как это наш второй вариант использования, мы снабжаем его соответствующим описанием.

Возьми в руку карандаш
Решение

Добавьте новый вариант использования для хранения образца лая.

Нам понадобится вариант использования для сохранения образца лая хозяйской собаки; мы будем хранить его прямо в двери (специалисты из фирмы Дуга говорят, что с их уровнем технологии это несложно). Воспользуйтесь приведенным ниже шаблоном для создания нового варианта использования для этой задачи.

Мы пока не знаем, как это делается, — это аппаратная задача.

Идеальная собачья дверь, версия 3.0

Сохранение образца лая

1. **Хозяйская собака лает рядом с собачьей дверью.**
2. **Дверь сохраняет образец лая хозяйской собаки.**

А вот это должны сделать мы... Нужно добавить в DogDoor метод для сохранения образца лая хозяйской собаки.

Часто задаваемые вопросы

В: Неужели для сохранения лая хозяйской собаки нужен отдельный вариант использования?

О: Да. Каждый вариант использования должен подробно описывать одну конкретную пользовательскую цель. Для исходного варианта эта цель заключалась в том, чтобы выпустить собаку на улицу и вернуть ее домой. Целью нового варианта использования является сохранение образца собачьего лая. Так как мы имеем дело с двумя разными целями, нам понадобятся два разных варианта использования.

В: Это действительно результат хорошего анализа или просто что-то такое, что мы упустили в двух последних главах?

О: Вероятно и то и другое. Конечно, необходимость сохранения образца лая следовало осознать намного ранее, но ведь анализ для этого и нужен: мы убеждаемся в том, что не забыли ничего, что помогло бы вашей программе работать в реальном контексте.

В: В каком виде будет храниться собачий лай?

О: Хороший вопрос. Сейчас мы займемся им...

Головоломка

Вы знаете, какие классы уже присутствуют в системе, и у вас есть два варианта использования, которые описывают, как должен работать код. Попробуйте определить, как должен измениться ваш код:

Ваша задача:

1. Добавьте все новые объекты, которые, как вы считаете, понадобятся для новой версии собачьей двери.
2. Включите в класс **DogDoor** новый метод, который будет сохранять образец лая, и еще один метод, при помощи которого другие классы смогут обращаться к записанному образцу.
3. Если вам потребуется внести изменения в другие кассы или методы, запишите их на приведенной ниже диаграмме классов.
4. Разместите на диаграмме классов пометки, напоминающие о том, для чего нужны неочевидные атрибуты или операции и как они должны работать.

Мы использовали диаграммы классов еще в главе 1; на них представлены базовые конструкции уровня программного кода в вашем приложении.

Напоминаем: это атрибуты вашего класса, которые обычно соответствуют переменным экземпляров класса...

...а это операции класса, которые обычно соответствуют открытым методам класса.

Remote
door: DogDoor
pressButton()

DogDoor
open: boolean
open()
close()
isOpen(): boolean

Обновите класс DogDoor для поддержки нового варианта использования, описанного на с. 182.

BarkRecognizer
door: DogDoor
recognize(String)

Оборудование Дуга передает этому методу запись услышанного лая собаки.

История двух программистов

Головоломку на с. 183 можно решить многими способами. У Рэнди и Сэма – двух разработчиков, только что принятых в фирму Дуга, – есть на этот счет неплохие идеи. Более того, наградой победителю будет нечто большее, чем гордость программиста; Дуг назначил за разработку лучшей структуры кода приз – новенький Apple MacBook Pro!

17 дюймов чистой мощи Apple и Intel.

Рэнди: чем проще — тем лучше, верно?

Рэнди не тратит время на лишний код. Он думает о том, как сравнивать услышанный лай с записанным образцом:

Звуки лая — это просто строки, так что я сохраню строки образца лая хозяйской собаки в DogDoor и добавлю пару простых методов. Кусок торта!

Рэнди

```
public class DogDoor {

  private boolean open;
  private String allowedBark;

  public DogDoor() {
    open = false;
  }

  public void setAllowedBark(String bark) {
    this.allowedBark = bark;
  }

  public String getAllowedBark() {
    return allowedBark;
  }

  // etc
}
```

Рэнди добавляет в свой класс DogDoor переменную allowedBark.

Этот метод задает образец лая, занимающий центральное место в нашем новом варианте использования.

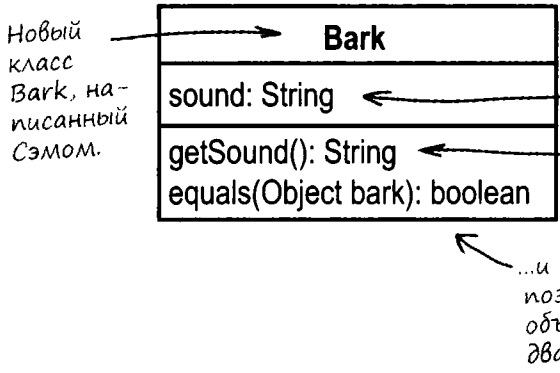
Другие классы могут получить образец лая хозяйской собаки с этим методом.

DogDoor на диаграмме классов Рэнди.

DogDoor
open: boolean **allowedBark: String**
open() close() isOpen(): boolean **setAllowedBark(String)** **getAllowedBark(): String**

Сэм: поклонник объектов

Возможно, по скорости работы Сэм уступает Рэнди, но он предпочитает работать с объектами. Сэм прикидывает, что создание нового класса, представляющего собачий лай, — это именно то, что нужно:



Возьми в руку карандаш

Написать код по имеющейся диаграмме классов проще простого.

Вы уже видели, что диаграммы классов предоставляют подробную информацию об атрибутах и операциях класса. Ваша задача — написать код класса Bark на основании его диаграммы классов. Мы написали часть кода, чтобы немного помочь вам в самом начале.

```
public class ____________ {
  private ___________ _________;

  public _____________(___________ ____________) {
    this._________ = ___________;
  }

  public ________ ___________() {
    ___________ _________;
  }

  ________ _____________ ______________(________ ________) {
    if (________ instanceof _________) {
      Bark otherBark = (_________)_________;
      if (this._________.equalsIgnoreCase(_______._______)) {
        return ___________;
      }
    }
    return ____________;
  }
}
```

Возьми в руку карандаш
Решение

Написать код по имеющейся диаграмме классов проще простого.

Вы должны были написать код класса Bark, предложенного Сэмом, на основании его диаграммы классов. Вот что получилось у нас:

Диаграмма классов для Bark

Bark
sound: String
getSound(): String equals(Object bark): boolean

Сэм, как и Рэнди, использует для хранения образца лая поле типа String...

...но он упаковал образец в объект Bark.

```
public class Bark {
  private String sound;

  public Bark (String sound) {
    this.sound = snund;
  }

  public String getSound() {
    return sound;
  }

  public boolean equals(Object bark) {
    if (bark instanceof Bark) {
      Bark otherBark = (Bark) bark;
      if (this.sound.equalsIgnoreCase(otherBark.sound)) {
        return true;
      }
    }
    return false;
  }
}
```

Сэм планирует, что другие классы будут делегировать сравнение объектов Bark методу equals() класса Bark.

Этот метод проверяет, что объект будет сравниваться с другим объектом Bark...

...а затем сравнивает два объекта Bark.

Сэм: обновление класса DogDoor

Так как Сэм создал новый объект **Bark**, обновление его версии класса **DogDoor** идет по несколько иному пути, чем в версии Рэнди:

(Sam's) DogDoor
open: boolean allowedBark: Bark
open() close() isOpen(): boolean setAllowedBark(Bark) getAllowedBark(): Bark

(Randy's) DogDoor
open: boolean allowedBark: String
open() close() isOpen(): boolean setAllowedBark(String) getAllowedBark(): String

В версии DogDoor, предложенной Сэмом, хранится объект Bark, а не простое значение String.

Операции get и set в проекте Сэма работают с объектами Bark, а не со значениями String.

Сравнение объектов Bark

Остается лишь добавить сравнение объектов Bark в метод **recognize()** класса **BarkRecognizer**.

Рэнди: я просто сравниваю две строки

Когда класс **BarkRecognizer** получает от оборудования сигнал об услышанном лае, он также получает запись и сравнивает ее с образцом, хранящимся в двери:

```
public class BarkRecognizer {

  public void recognize(String bark) {
    System.out.println("   BarkRecognizer: "
      "Heard a '" + bark + "'");
    if (door.getAllowedBark().equals(bark)) {
      door.open();
    } else {
      System.out.println("This dog is " +
        "not allowed.");
    }
  }

  // ...
}
```

При вызове recognize() передается значение String с услышанным лаем.

Сравнение лая, полученного с распознавателя, с образцом, хранящимся в двери.

Сэм: А я делегирую операцию сравнения

Сэм использует объект **Bark**, и все сравнения звуков поручаются этому объекту:

```
public class BarkRecognizer {

  public void recognize(Bark bark) {
    System.out.println("   BarkRecognizer: "
      "Heard a '" + bark.getSound() + "'");
    if (door.getAllowedBark().equals(bark)) {
      door.open();
    } else {
      System.out.println("This dog is not allowed.");
    }
  }

  // ...
}
```

Сэм договаривается с инженерами, чтобы ему передавался объект Bark вместо простого значения String (как у Рэнди).

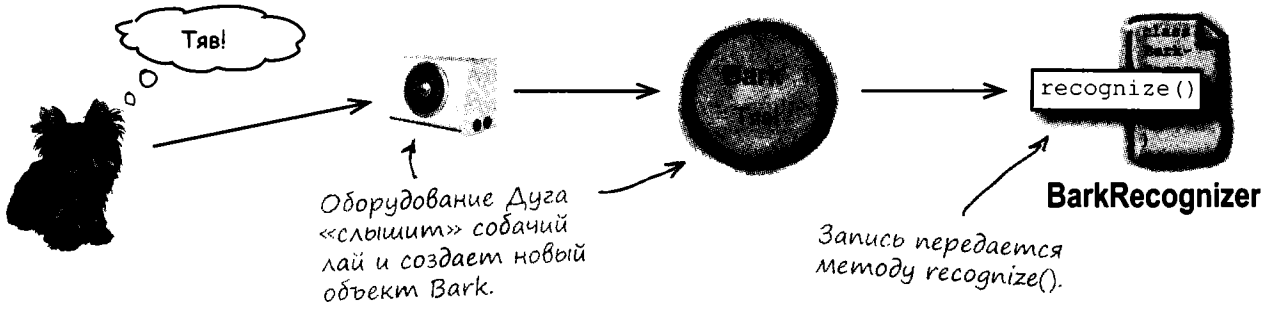
В коде Сэма сравнение выполняется объектом Bark, хранящимся в DogDoor. Его версия BarkRecognizer делегирует сравнение объекту Bark.

Делегирование в версии Сэма: более подробный взгляд

Сэм использует в своих классах **Bark** и **DogDoor** очень похожие приемы. Давайте посмотрим, что там происходит:

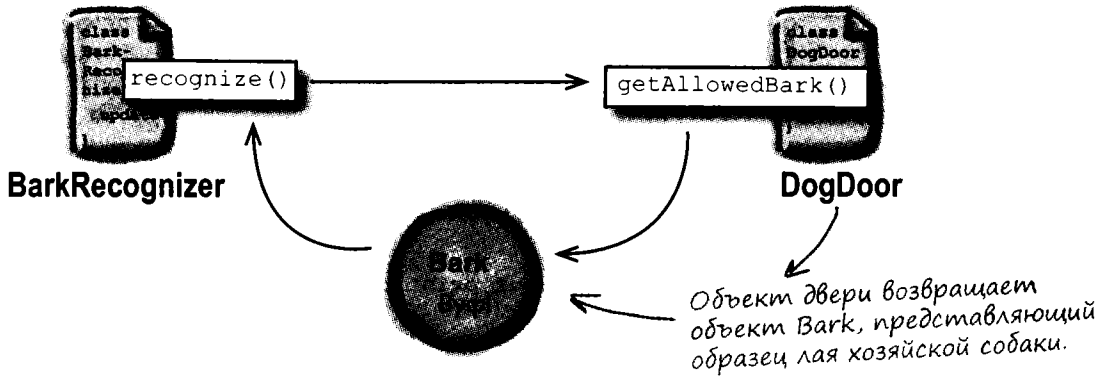
1 BarkRecognizer получает объект Bark для обработки.

Оборудование Дуга «слышит» собачий лай, создает новый объект **Bark**, передает его в метод **recognize()**.



2 BarkRecognizer получает от DogDoor образец лая хозяйской собаки

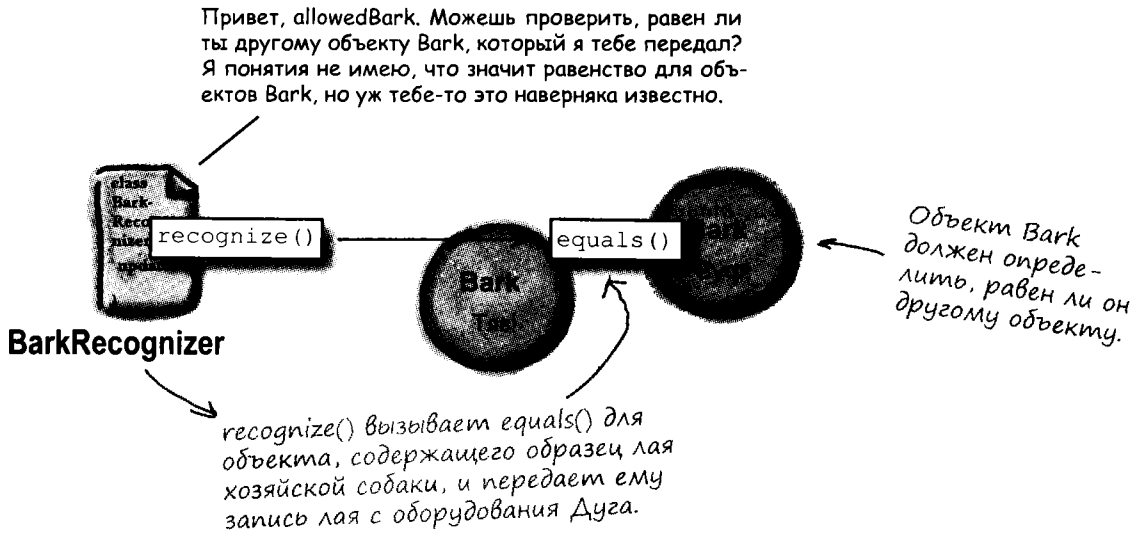
Метод recognize() вызывает метод **getAllowedBark()** объекта, с которым он связан, и получает объект **Bark**, представляющий образец лая хозяйской собаки.



Мы еще вернемся к состязанию Сэма и Рэнди за MacBook Pro, когда поближе познакомимся с делегированием.

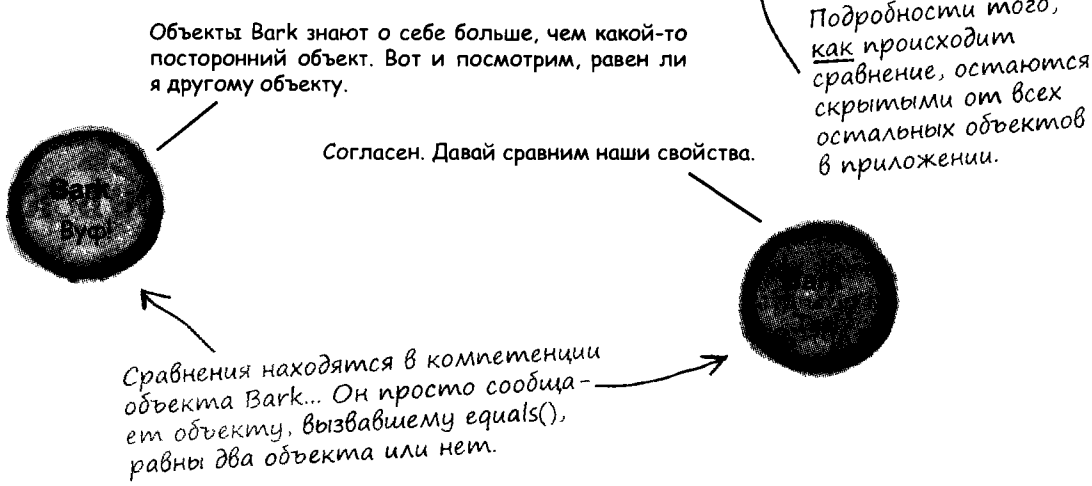
3 BarkRecognizer делегирует сравнение Bark

Метод **`recognize()`** приказывает объекту **`Bark`** хозяйской собаки проверить, равен ли он экземпляру **`Bark`**, полученному от оборудования Дуга. Для проверки используется метод **`Bark.equals()`**.



4 Объект Bark решает, равен ли он объекту, полученному от оборудования Дуга

Объект **`Bark`**, представляющий лай хозяйской собаки, проверяет, равен ли он объекту **`Bark`**, полученному от оборудования Дуга... ***Каким бы способом ни проводилось сравнение.***

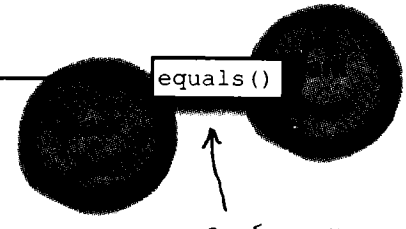


Сила приложений с низкой связностью

В главе 1 говорилось о том, что делегирование помогает обеспечить слабую связанность приложений. Это означает, что объекты остаются независимыми друг от друга; другими словами, изменения в одном объекте не требуют внесения многочисленных изменений в других объектах.

Делегируя сравнение объекту **Bark**, мы абстрагируем информацию о том, какие два объекта Bark должны считаться равными, из класса **BarkRecognizer**. Давайте еще раз рассмотрим код, вызывающий **equals()** для **Bark**:

```
public void recognize(Bark bark) {
  System.out.println("   BarkRecognizer: "
    "Heard a '" + bark.getSound() + "'");
  if (door.getAllowedBark().equals(bark)) {
    door.open();
  } else {
    System.out.println("This dog is not allowed.");
  }
}
```



Подробности работы equals() остаются скрытыми от метода recognize().

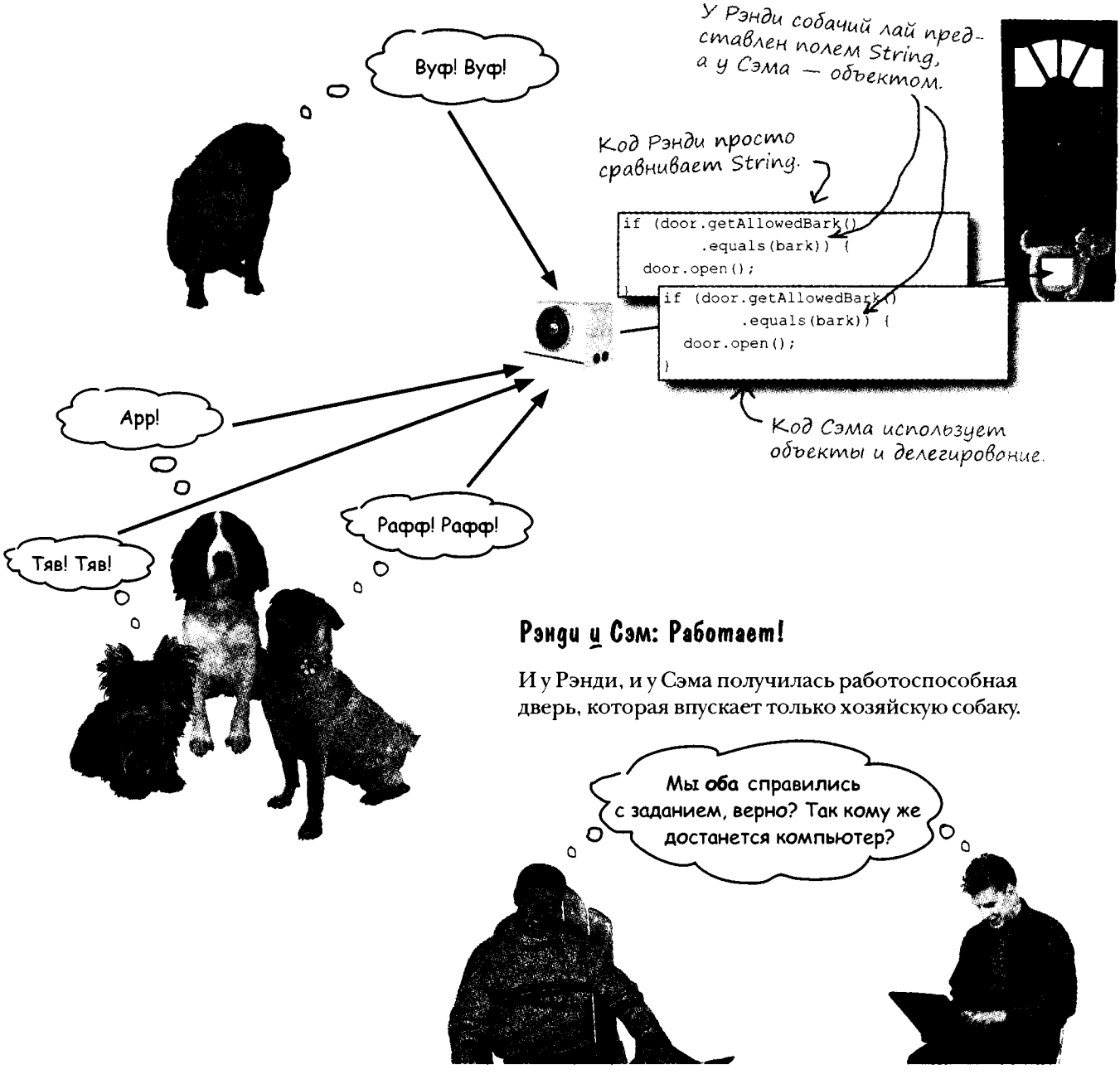
Теперь предположим, что звук собачьего лая будет храниться в Bark в виде файла WAV. Нам придется изменить метод **equals()** в классе **Bark**, чтобы выполнить более сложное сравнение с учетом специфики файлов WAV. Но поскольку метод **recognize()** делегирует сравнение, код **BarkRecognizer** изменять не придется..

Итак, с использованием делегирования и слабой связанности приложений можно изменить реализацию одного объекта (например, **Bark**), и это не приведет к изменению всех остальных объектов приложения. Ваши объекты *изолируются* от изменений реализации в других объектах.

Делегирование изолирует объекты от изменений реализации других объектов вашей программы.

Возвращаемся к Сэму, Рэнди и конкурсу...

Рэнди предлагает быстрое решение, Сэм — более объектно-ориентированное. Посмотрим, как их решения покажут себя в деле.

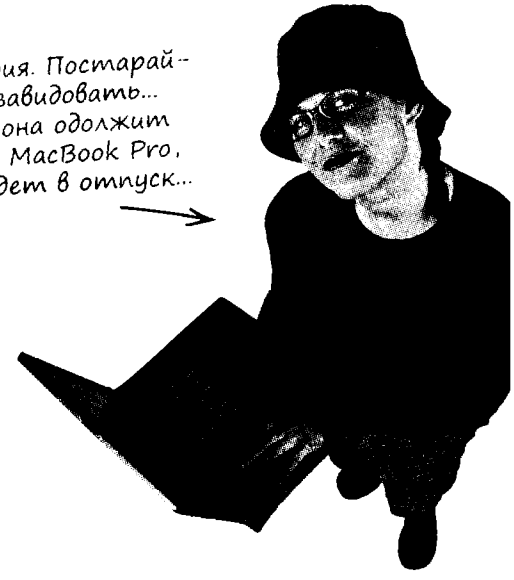


Рэнди и Сэм: Работает!

И у Рэнди, и у Сэма получилась работоспособная дверь, которая впускает только хозяйскую собаку.

MacBook Pro выиграла Мария!

К удивлению Рэнди и Сэма, Дуг объявил, что компьютер выиграла Мария – младший программист, проходящий летнюю практику на фирме.



Рэнди: Да это просто смешно. Мое решение работало! Компьютер должен достаться мне, а не какой-то практикантке!

Сэм: Мое решение тоже работало, и я использовал объекты. Ты не читал «Head First Java»? Нужно применять объектно-ориентированные решения... Компьютер мой!

Мария: Ммм... Парни, не хочу вам мешать, но я не уверена, что ваши решения действительно работали.

Сэм: Что значит «не уверена»? Мы их протестировали. Брюс сказал «Вуф!», и дверь открылась... Но осталась закрытой для других собак. По мне, так вполне работающее решение.

Мария: А вы анализировали свои решения? Ваши двери действительно работали в реальном мире?

Рэнди: О чем ты говоришь? Ты кто, программист или философ? Или это прикол типа «ложки не существует»?

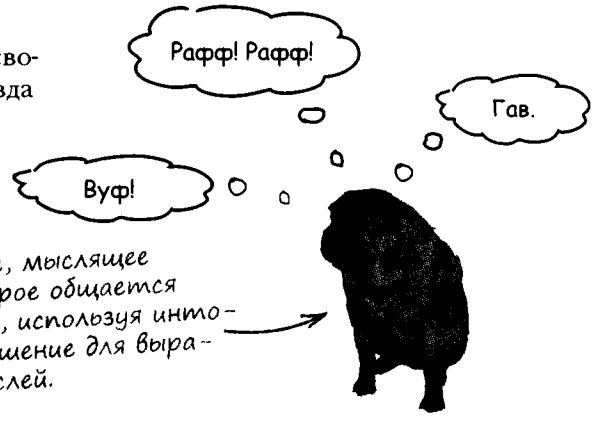
Мария: Нет, вовсе нет. Просто я подумала... А если Брюс скажет что-то другое? Скажем, «Гррр» или «Рафф»?

Сэм: Другой звук? Если он голоден...

Рэнди: ...или взволнован...

Мария: ...или ему очень нужно выйти на улицу по своим делам. В реальном мире такое случается, не правда ли?

Рэнди и Сэм: Пожалуй, мы об этом не подумали...

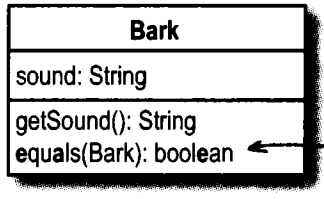


Брюс — сложное, мыслящее существо, которое общается с окружающими, используя интонации и произношение для выражения своих мыслей.

Чем же отличалось решение Марии?

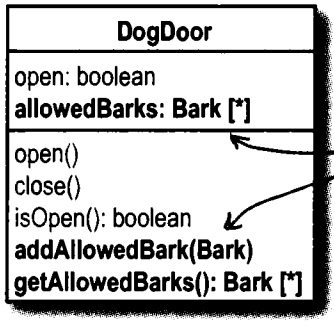
Мария начала примерно с того же, что и Сэм, — она создала объект **Bark**, представляющий собачий лай.

Я же говорил, что объекты и делегирование — это важно!



Мария знала, что ей понадобится делегирование с использованием метода equals(), как в решении Сэма.

Но Мария пошла еще дальше: она решила, что поскольку собака может лаять по-разному, ей понадобятся несколько объектов **Bark**. И тогда при любом варианте лая собака все равно сможет попасть наружу:



А вот здесь Мария пошла по другому пути. Она решила, что собачья дверь должна хранить более одного образца лая, потому что хозяйская собака может лаять по-разному.

Интересуетесь, что это за звездочки? Посмотрите сюда...

UML под увеличительным стеклом

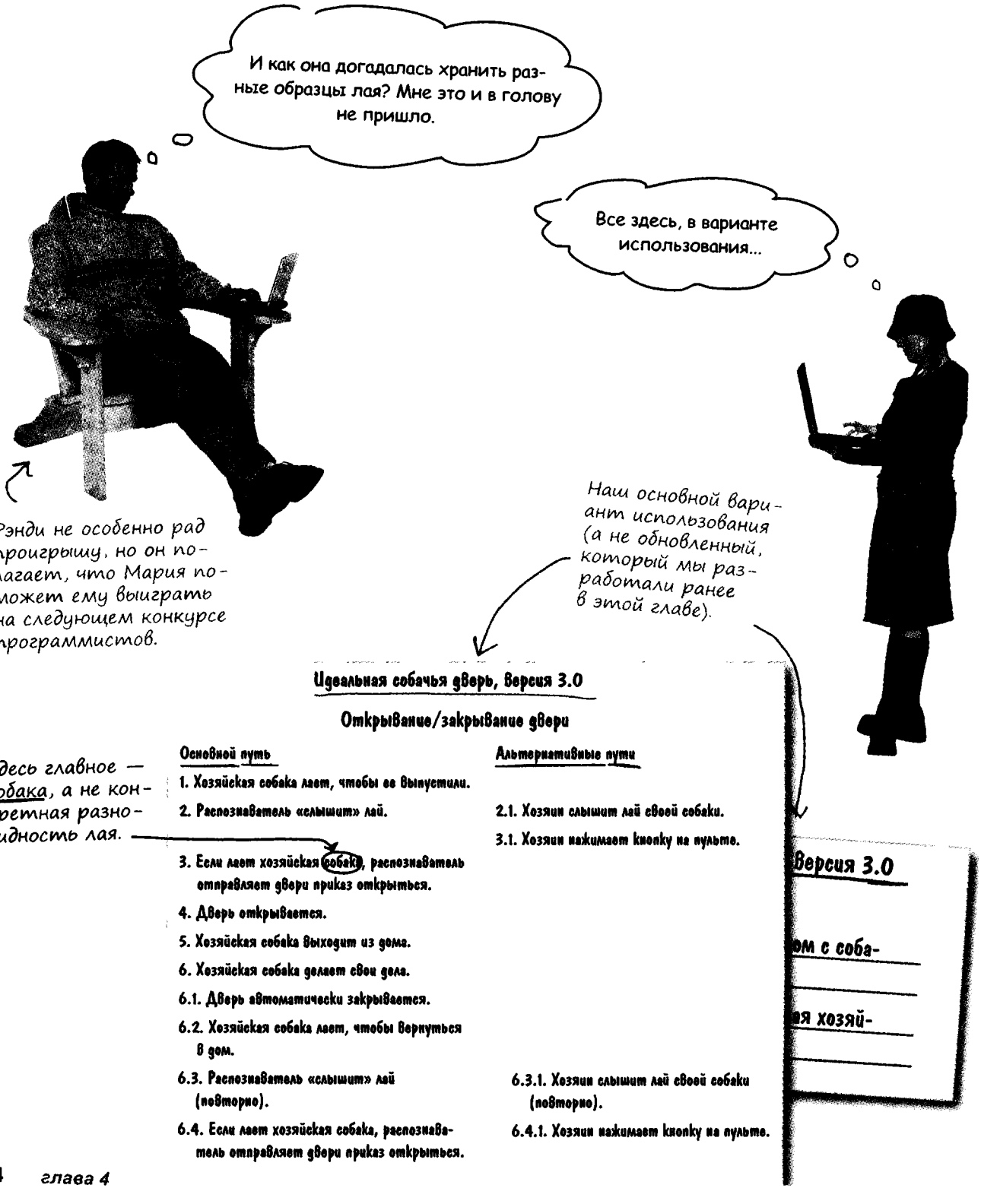
На нашей диаграмме классов появилось нечто новое:

allowedBarks: Bark [*]

Квадратные скобки обозначают **мощность** атрибута, то есть количество значений типа, которые может содержать атрибут.

Атрибут allowedBarks содержит объекты типа Bark.

А звездочка указывает, что allowedBarks может содержать неограниченное количество объектов Bark.


И как она догадалась хранить разные образцы лая? Мне это и в голову не пришло.
Все здесь, в варианте использования...
Рэнди не особенно рад проигрышу, но он полагает, что Мария поможет ему выиграть на следующем конкурсе программистов.
Наш основной вариант использования (а не обновленный, который мы разработали ранее в этой главе).
Здесь главное — собака, а не конкретная разновидность лая.
Идеальная собачья дверь, версия 3.0
Открывание/закрывание двери
Основной путь
1. Хозяйская собака лает, чтобы ее выпустили.
2. Распознаватель «слышит» лай.
3. Если лает хозяйская собака, распознаватель отправляет двери приказ открыться.
4. Дверь открывается.
5. Хозяйская собака выходит из дома.
6. Хозяйская собака делает свои дела.
6.1. Дверь автоматически закрывается.
6.2. Хозяйская собака лает, чтобы вернуться в дом.
6.3. Распознаватель «слышит» лай (повторно).
6.4. Если лает хозяйская собака, распознаватель отправляет двери приказ открыться.
Альтернативные пути
2.1. Хозяин слышит лай своей собаки.
3.1. Хозяин нажимает кнопку на пульте.
6.3.1. Хозяин слышит лай своей собаки (повторно).
6.4.1. Хозяин нажимает кнопку на пульте.
Версия 3.0
ом с соба-
ая хозяй-

Обращайте внимание на существительные в варианте использования

Мария обнаружила нечто очень важное: именам существительным в вариантах использования обычно соответствуют классы, которые вам придется написать и которые будут играть важную роль в системе.

Возьми в руку карандаш

Обведите кружком каждое существительное (человек, место, предмет) в следующем варианте использования. Запишите в нижней части страницы все найденные существительные (каждое в одном экземпляре, дубликаты не нужны). Обязательно выполните упражнение перед тем, как перевернуть страницу!

Идеальная собачья дверь, версия 3.0

Открывание/закрывание двери

Основной путь	Альтернативные пути
1. Хозяйская собака лает, чтобы ее выпустили.	
2. Распознаватель «слышит» лай.	2.1. Хозяин слышит лай своей собаки.
3. Если лает хозяйская собака, распознаватель отправляет двери приказ открыться.	3.1. Хозяин нажимает кнопку на пульте.
4. Дверь открывается.	
5. Хозяйская собака выходит из дома.	
6. Хозяйская собака делает свои дела.	
6.1. Дверь автоматически закрывается.	
6.2. Хозяйская собака лает, чтобы вернуться в дом.	
6.3. Распознаватель «слышит» лай (повторно).	6.3.1. Хозяин слышит лай своей собаки (повторно).
6.4. Если лает хозяйская собака, распознаватель отправляет двери приказ открыться.	6.4.1. Хозяин нажимает кнопку на пульте.
6.5. Дверь открывается (повторно).	
7. Хозяйская собака заходит в дом.	
8. Дверь автоматически закрывается.	

«собака» — существительное (также можно обвести кружком «хозяйскую собаку»).

Запишите существительные, которые вы обвели в варианте использования.

_____ _____ _____

_____ _____ _____

_____ _____ _____

Возьми в руку карандаш
Решение

Обведите кружком каждое существительное (человек, место, предмет) в следующем варианте использования. Запишите в нижней части страницы все найденные существительные (каждое в одном экземпляре, дубликаты не нужны). Обязательно выполните упражнение перед тем, как перевернуть страницу!

Идеальная собачья дверь, версия 3.0

Открывание/закрывание двери

Основной путь

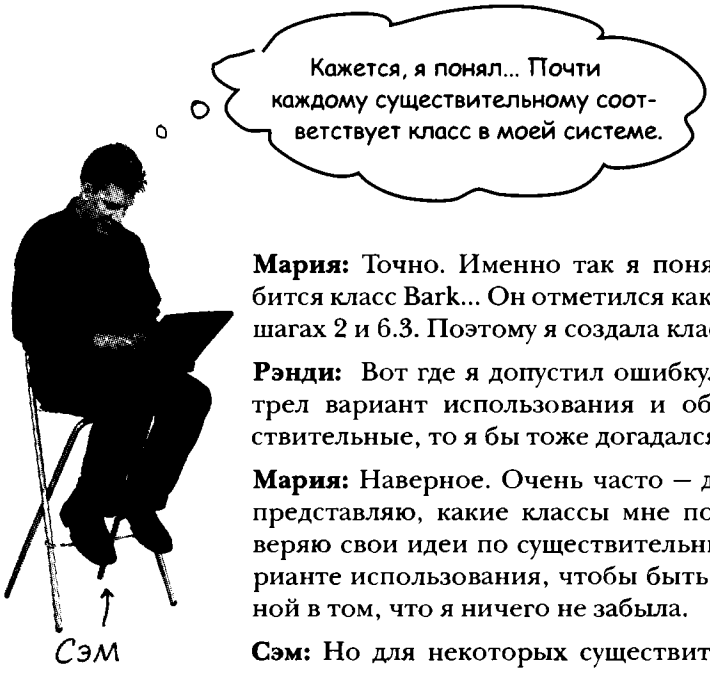
1. Хозяйская собака лает, чтобы ее выпустили.
2. Распознаватель «слышит» лай.
3. Если лает хозяйская собака, распознаватель отправляет двери приказ открыться.
4. Дверь открывается.
5. Хозяйская собака выходит из дома.
6. Хозяйская собака делает свои дела.
6.1. Дверь автоматически закрывается.
6.2. Хозяйская собака лает, чтобы вернуться в дом.
6.3. Распознаватель «слышит» лай (повторно).
6.4. Если лает хозяйская собака, распознаватель отправляет двери приказ открыться.
6.5. Дверь открывается (повторно).
7. Хозяйская собака заходит в дом.
8. Дверь автоматически закрывается.

Альтернативные пути

2.1. Хозяин слышит лай своей собаки.
3.1. Хозяин нажимает кнопку на пульте.
6.3.1. Хозяин слышит лай своей собаки (повторно).
6.4.1. Хозяин нажимает кнопку на пульте.

(хозяйская) собака	распознаватель	собачья дверь
хозяин	приказ	пульт
кнопка	дом/улица	лай

Существительные, выделенные нами в варианте использования.



Мария: Точно. Именно так я поняла, что мне понадобится класс Bark... Он отметился как существительное на шагах 2 и 6.3. Поэтому я создала класс Bark.

Рэнди: Вот где я допустил ошибку... Если бы я просмотрел вариант использования и обвел кружками существительные, то я бы тоже догадался создать класс Bark.

Мария: Наверное. Очень часто — даже когда я хорошо представляю, какие классы мне понадобятся, — я проверяю свои идеи по существительным в варианте использования, чтобы быть уверенной в том, что я ничего не забыла.

Сэм: Но для некоторых существительных классы не нужны — скажем, «хозяин» или «приказ».

Мария: Да, верно... Проектировщик должен обладать здравым смыслом и понимать систему, которую он строит. Помните: классы нужны только для тех частей, которые вы собираетесь представить в структуре системы. Классы для «хозяина» или «приказа» нам не нужны, потому что эти понятия не представлены в структуре нашей системы.

Рэнди: И класс для «кнопки» тоже не нужен, потому что кнопка является частью пульта дистанционного управления, а для него класс уже имеется.

Сэм: Все это очень здорово, но я вот что подумал... Я тоже определил класс Bark, и вариант использования мне для этого не понадобился.

Мария: Да... Но в результате твоя дверь не работала, не так ли?

Сэм: Так, но... Ведь это потому, что ты решила хранить в двери несколько объектов Bark. Какое отношение это имеет к варианту использования?

Поиск существительных (и глаголов) в варианте использования с целью определения классов и методов называется <u>текстологическим анализом</u>.

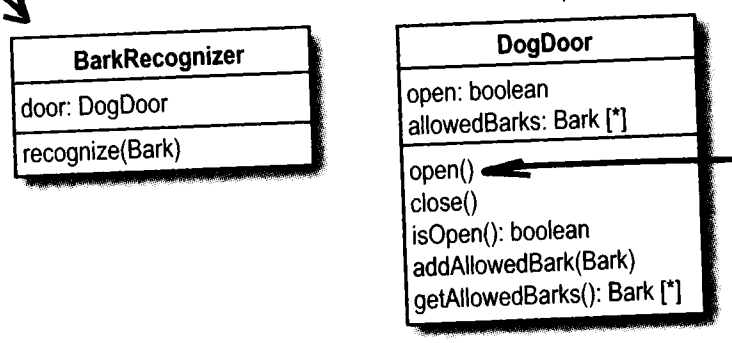
Вариант использования определяет все

Присмотритесь к шагу 3 варианта использования и обратите внимание на то, какие классы в нем используются:

«Хозяйская собака» — тоже существительное, но для него класс не понадобится, потому что собака выполняет функции субъекта и существует вне системы.

3. Если лает хозяйская собака, распознаватель отправляет двери приказ открыться.

Приказ — еще одно существительное без класса — представлен вызовом метода open() для собачьей двери.



Класса Bark здесь нет!

На шаге 3 используются классы распознавателя лая (BarkRecognizer) и собачьей двери (DogDoor)... Но не класс Bark!

3. Если лай хозяйской собаки совпадает с лаем, услышанным распознавателем, дверь открывается.

Шаг 3 из варианта использования, который Рэнди написал для своей двери. В его шаге 3 «лай» является существительным.

Шаг 3 в варианте использования Рэнди очень похож на шаг 3 нашего варианта... Но в нем центральное место занимает существительное «лай», а не «хозяйская собака». Выходит, Рэнди прав? И весь текстологический анализ рассыпается, если в нем используются другие слова?

Как вы думаете?

ПОДСКАЗКА: Присмотритесь внимательнее к шагу 3 в формулировке Рэнди. Работает ли описываемая им система точно так же, как система на с. 198?

Эти две формулировки не эквивалентны...

Похоже, шаг 3 в формулировке Рэнди немного отличается от нашего исходного шага 3... В чем же ошибся Рэнди?

Это шаг 3 из исходного варианта использования, который мы написали в главе 3.

3. Если лает хозяйская собака, распознаватель отправляет двери приказ открыться.

А это шаг 3 из варианта использования, написанного Рэнди для той же собачьей двери.

3. Если лай хозяйской собаки совпадает с лаем, услышанным распознавателем, дверь открывается.

Центр внимания: хозяйская собака

В исходной формулировке шага 3 центральное место занимает хозяйская собака... *какие бы звуки она ни издавала.* Следовательно, если сегодня хозяйская собака громко лает «Гав!», а завтра тихо лает «Рафф!», система в любом случае ее впустит. Это объясняется тем, что центральное место в нашей формулировке занимает *собака*, а не конкретная разновидность лая.

Центр внимания: лай хозяйской собаки

Вариант использования Рэнди сосредоточен на лае хозяйской собаки... А если собака сможет издавать несколько разных звуков? И если две собаки могут лаять похожим образом? Этот шаг очень похож на исходный шаг 3, но *на самом деле они не эквивалентны!*

Часто задаваемые вопросы

В: Итак, если я пишу варианты использования, мои программы будут работать так, как положено?

О: Безусловно, варианты использования становятся хорошей отправной точкой для написания хороших программ. Тем не менее это отнюдь не все. Вспомните, что анализ помогает определить классы на основании варианта использования. В следующей главе мы обсудим принципы проектирования, применяемые при написании этих классов.

В: Прежде я никогда не применял варианты использования и у меня никогда не возникало проблем. А теперь вы говорите, что варианты использования необходимы для написания хороших программ?

О: Вовсе нет. Многие программисты, хорошо справляющиеся со своей работой, даже не знают, что такое вариант использования. Но если вы хотите, чтобы заказчики чаще оставались довольны вашими программами и ваш код работал правильно с минимумом переработки, варианты использования помогут правильно сформулировать требования... до того, как вы совершите досадную ошибку перед своим начальником или заказчиком.

В: Похоже, вся эта возня с существительными и анализом достаточно сложна, а я не силен в грамматике. Что мне делать?

О: Грамматика — не главное. Записывайте свои варианты использования на том языке, на котором вы говорите и пишете. Найдите в своем варианте использования «предметные понятия» — обычно они являются существительными. Для каждого существительного подумайте, нужен ли вам класс для его представления, — и это станет хорошей отправной точкой для анализа системы в реальном контексте.

В: А если я совершу ошибку, как Рэнди, и использую существительное там, где этого делать не следует?

О: Ошибка Рэнди на шаге 3 его варианта использования не имеет никакого отношения к грамматике. Рэнди не подумал над вариантом использования и тем, как его система должна работать в реальном мире. Вместо того чтобы сосредоточиться на том, как выпустить собаку из дома, он занялся конкретной разновидностью лая. Он сосредоточился не на том!

Записав свой вариант использования, перечитайте его и убедитесь, что вы его хорошо понимаете. Возможно, вам даже стоит дать почитать его своим коллегам или друзьям — так вы сможете быть уверены в том, что вариант будет работать в реальном мире, а не только в контролируемой среде.

Хороший вариант использования четко и точно, на понятном языке объясняет, что должна делать система.

Когда вариант использования будет написан, текстологический анализ поможет легко и быстро определить классы в вашей системе.

Теперь я вижу, в чем заключалась ошибка Рэнди: он ориентировался на лай, а не на собаку владельца. Но даже в правильном варианте использования нет объекта Dog. Тогда для чего все это было нужно, если наш анализ не указывает, какие классы нужно написать и использовать?

Текстологический анализ помогает определить, на чем следует сосредоточиться, а не какие классы следует создавать.

Хотя в нашем приложении нет класса **Dog**, текстологический анализ дает важную информацию о том, как должна работать наша система: она должна впускать и выпускать хозяйскую собаку из дома *независимо от того, как она лает.* Другими словами, наш анализ помог нам понять, на чем следует сосредоточиться... И это *не* конкретная разновидность лая.

Когда это становится понятно, логично подумать о том, что должна делать собака. Всегда ли собака лает одинаково? Здесь-то Мария и нашла свое реалистичное решение: ведь если собака может лаять по-разному, а цель заключается в том, чтобы выпустить собаку из дома, дверь должна хранить *все* возможные варианты лая, а не только один из них. Но Мария никогда не осознала бы этого без анализа своего варианта использования.

Возьми в руку карандаш

Почему в приложении нет класса Dog?

Когда мы искали существительные в варианте использования, среди них неоднократно встречалась «хозяйская собака». Но Мария решила не создавать объект Dog. Почему? Запишите три причины, которые, по вашему мнению, объясняют, почему Мария не стала создавать класс Dog в своей системе.

1. _____

2. _____

3. _____

Ответы на с. 207.

Помните: существительные очень важны!

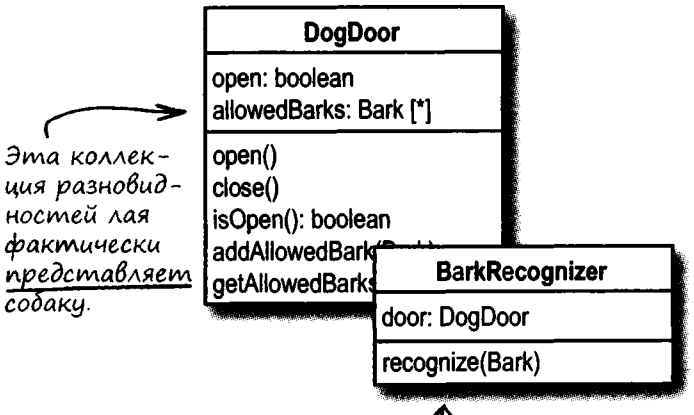
Даже если существительные в варианте использования не преобразуются в классы систем, они все равно важны для того, чтобы ваша система работала так, как задумано.

В этом варианте использования «хозяйская собака» — существительное, но это не класс...

3. Если лает хозяйская собака, распознаватель отправляет двери приказ открыться.

...и хотя на этом шаге «лает» не является существительным, в системе имеется класс Bark.

Существительные указывают, на чем следует сосредоточиться. Если на этом шаге центром внимания будет собака, мы поймем, что необходимо сделать так, чтобы собака могла входить и выходить из двери — с одним вариантом лая или с несколькими.



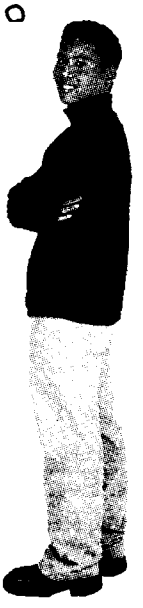
Эта коллекция разновидностей лая фактически представляет собаку.

И хотя этот метод получает только одну разновидность лая, его задача — определить, какая собака залаяла. Для этого он перебирает все допустимые разновидности лая, хранимые в двери, и определяет, принадлежит ли этот лай хозяйской собаке.

Обращайте внимание на существительные в вариантах использования, даже если они не имеют прямых аналогов среди классов вашей системы.

Подумайте, как имеющиеся классы могут обеспечить поведение, описанное вашим вариантом использования.

Если существительные в варианте использования обычно соответствуют классам системы, то **глаголы**, вероятно, соответствуют методам. Логично, не так ли?



Глаголы в варианте использования (обычно) соответствуют методам объектов системы.

Как вы уже видели, существительные в варианте использования обычно становятся хорошей отправной точкой для определения классов, которые могут потребоваться в вашей системе. А по глаголам обычно можно определить, какие методы должны содержать объекты, представленные этими классами.

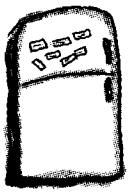
Идеальная собачья дверь, версия 3.0

Открывание/закрывание двери

Основной путь	Альтернативные пути
1. Хозяйская собака лает, чтобы ее выпустили.	
2. Распознаватель «слышит» лай.	2.1. Хозяин слышит лай своей собаки.
3. Если лает хозяйская собака, распознаватель отправляет двери приказ открыться.	3.1. Хозяин нажимает кнопку на пульте.
4. Дверь открывается.	
5. Хозяйская собака выходит из дома.	
6. Хозяйская собака делает свои дела.	
6.1. Дверь автоматически закрывается.	
6.2. Хозяйская собака лает, чтобы вернуться в дом.	
6.3. Распознаватель «слышит» лай (повторно).	6.3.1. Хозяин слышит лай своей собаки (повторно).
6.4. Если лает хозяйская собака, распознаватель отправляет двери приказ открыться.	6.4.1. Хозяин нажимает кнопку на пульте.
6.5. Дверь открывается (повторно).	
7. Хозяйская собака заходит в дом.	
8. Дверь автоматически закрывается.	

Класс DogDoor должен содержать методы open() и close() для поддержки этих глаголов.

Другой фрагмент с глаголом: «нажимает кнопку». Наш класс Remote содержит метод pressButton(), который идеально соответствует этому глаголу.



Развлечения с магнитами

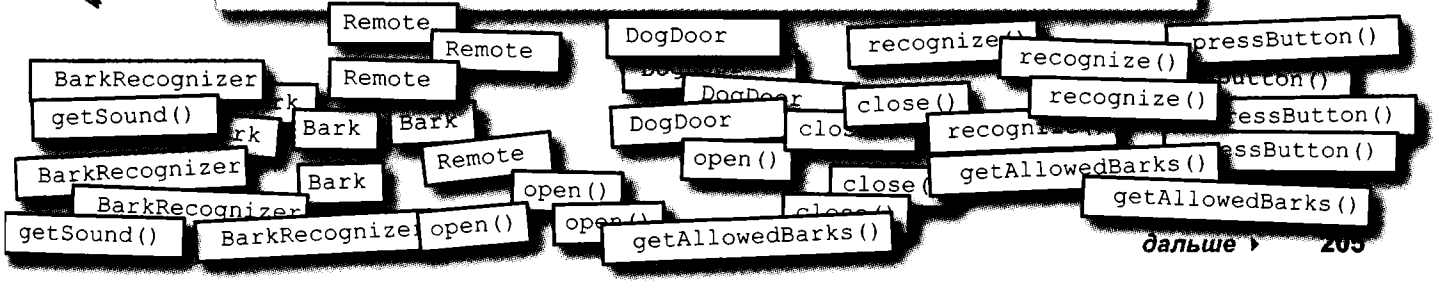
Пора заняться текстологическим анализом более основательно. Перед вами вариант использования для двери, которую мы разрабатываем. В нижней части страницы сложены магниты для большинства классов и методов, включенных в нашу систему. Найдите среди магнитов классов соответствия для существительных, а среди магнитов методов — для глаголов из нашего варианта использования. Обратите внимание, насколько близко методы соответствуют глаголам.

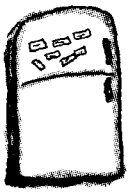
Идеальная собачья дверь, версия 3.0

Открывание/закрывание двери

Основной путь	Альтернативные пути
1. Хозяйская собака лает, чтобы ее выпустили.	
2. Распознаватель «слышит» лай.	2.1. Хозяин слышит лай своей собаки.
3. Если лает хозяйская собака, распознаватель отправляет двери приказ открыться.	3.1. Хозяин нажимает кнопку на пульте.
4. Дверь открывается.	
5. Хозяйская собака выходит из дома.	
6. Хозяйская собака делает свои дела.	
6.1. Дверь автоматически закрывается.	
6.2. Хозяйская собака лает, чтобы вернуться в дом.	
6.3. Распознаватель «слышит» лай (повторно).	6.3.1. Хозяин слышит лай своей собаки (повторно).
6.4. Если лает хозяйская собака, распознаватель отправляет двери приказ открыться.	6.4.1. Хозяин нажимает кнопку на пульте.
6.5. Дверь открывается (повторно).	
7. Хозяйская собака заходит в дом.	
8. Дверь автоматически закрывается.	

Здесь накопилось довольно много классов и методов, так что не торопитесь.





Развлечения с магнитами. Решение

Пора заняться текстологическим анализом более основательно. Перед вами вариант использования для двери, которую мы разрабатываем. В нижней части страницы сложены магниты для большинства классов и методов, включенных в нашу систему. Найдите среди магнитов классов соответствия для существительных, а среди магнитов методов — для глаголов из нашего варианта использования. Обратите внимание, насколько близко методы соответствуют глаголам.

Идеальная собачья дверь, версия 3.0

Открывание/закрывание двери

Основной путь

1. Хозяйская собака лает, чтобы ее выпустили.
2. BarkRecogni recognize() Bark
3. Если getAllowedBarks() узнава- Bar DogDoor open() крыть- ся.
4. DogDoor open()
5. Хозяйская собака выходит из дома.
6. Хозяйская собака делает свои дела.
   6.1. Д DogDoor close() тся.
   6.2. Хозяйская собака лает, чтобы вернуться в дом.
   6.3. BarkRecogni recognize() Bark лаю getAllowedBarks() BarkRec door приказ откр DogDoor open()
   6.5. DogDoor open()).
7. Хозяйская собака заходит в дом.
8. Д DogDoor close() тся.

Альтернативные пути

2.1. Хозяин слышит лай своей собаки.

3.1. Хозя pressButton() Remote

6.3.1. Хозяин слышит лай своей собаки (повторно).

6.4.1. Хозяин pressButton() Remote

Обратите внимание: большинство шагов без магнитов соответствует внешним событиям, происходящим за пределами системы.

Вариант использования с магнитами выглядит вполне разумно! И это хороший признак того, что классы и методы делают именно то, что им положено делать, а система будет успешно работать.

Возьми в руку карандаш
Решение

Почему в приложении нет класса Dog?

Когда мы искали существительные в варианте использования, среди них неоднократно встречалась «хозяйская собака». Но Мария решила не создавать объект Dog. Почему? Запишите три причины, которые, по вашему мнению, объясняют, почему Мария не стала создавать класс Dog в своей системе.

1. Собака существует за пределами системы, а в системе такие внешние концепции обычно не представляются.
2. Собака не является программным объектом (да и не должна им быть)... Обычно живые существа не представляются классами, если только система не должна хранить информацию о них в течение продолжительного времени.
3. Даже если бы в нашей структуре присутствовал класс Dog, он никак не помог бы работе системы. Например, «хранение» ссылки на собаку в двери просто не имеет смысла.

Иногда такие объекты создаются, но обычно это делается в ситуациях, когда система должна с ними взаимодействовать. В нашем случае взаимодействие с собакой не требуется.

В объекте DogDoor можно сохранить ссылку на класс Dog, но как сохранить собаку в двери в реальном мире? Не забудьте, что приемы, работающие в программировании, не всегда работают в реальной жизни. Ваши приложения должны быть совместимы с реальным миром!

В приложениях часто используются такие классы, как User или Manager, но они представляют не живые существа, а роли в системе, данные кредитных карт или адреса. Собака под эти схемы не подходит.

Часто задаваемые вопросы

В: Значит, существительные в вариантах использования заменяются классами, а глаголы — методами?

О: Почти так. Если говорить точнее, существительные становятся кандидатами для создания классов... Не каждое существительное преобразуется в класс. Например, существительное «хозяин» встречается в варианте использования (см. шаги 2.1 и 3.1), но класс для него не создается. Таким образом, хотя «хозяин» является кандидатом в классы, он не становится классом в итоговой системе.

По тому же принципу глаголы являются кандидатами для операций. Например, в варианте встречается глагольное выражение «делает свои дела», но нам и в голову не придет писать метод pee() или poop(). Надеемся, в этом вы с нами согласитесь! И все же текстологический анализ станет хорошей отправной точкой для определения классов и методов, которые стоит реализовать в системе.

В: Похоже, существительные, находящиеся за пределами системы, не преобразуются в классы. Это всегда так?

О: В большинстве случаев. Единственное типичное исключение встречается при внешних взаимодействиях системы — например, при наличии данных состояния или поведения, с которыми система должна периодически работать.

Так, в системе с собачьей дверью нам не нужен класс, представляющий хозяина, потому что класс Remote предоставляет всю функциональность, относящуюся к хозяину. Но при необходимости отслеживать состояние хозяина (скажем, спит он или бодрствует) нам пришлось бы создать класс Owner.

От качественного анализа к хорошим классам...

Диаграмма классов Марии

Remote
- pressButton()

BarkRecognizer
- recognize(Bark)

Куда исчез атрибут door класса Remote?

door 1 — door 1

На диаграмме классов Марии определенно происходит что-то новое.

DogDoor
- open: boolean
- open()
- close()
- isOpen(): boolean
- addAllowedBark(Bark)
- getAllowedBarks(): Bark [*]

Почему атрибут allowedBarks переместился отсюда...

...сюда?

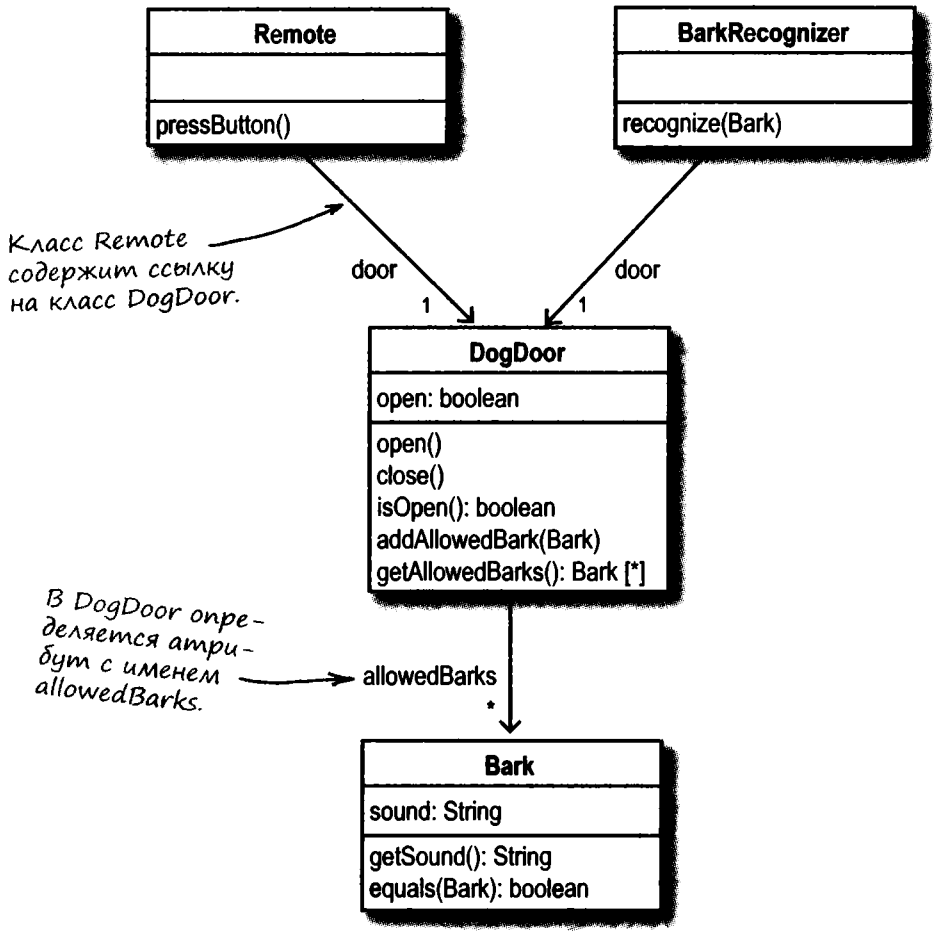
Напоминаем: звездочка означает, что getAllowedBarks() может вернуть несколько объектов Bark.

allowedBarks *

Bark
- sound: String
- getSound(): String
- equals(Bark): boolean

UML-диаграмма

Мария явно увлеклась своими диаграммами UML... Удастся ли вам разобраться в том, что она сделала? На следующей диаграмме пометьте все, что она добавила, и попробуйте определить, что означают все эти линии, числа и новые слова. Мы сделали несколько пометок, чтобы вам было проще взяться за дело.



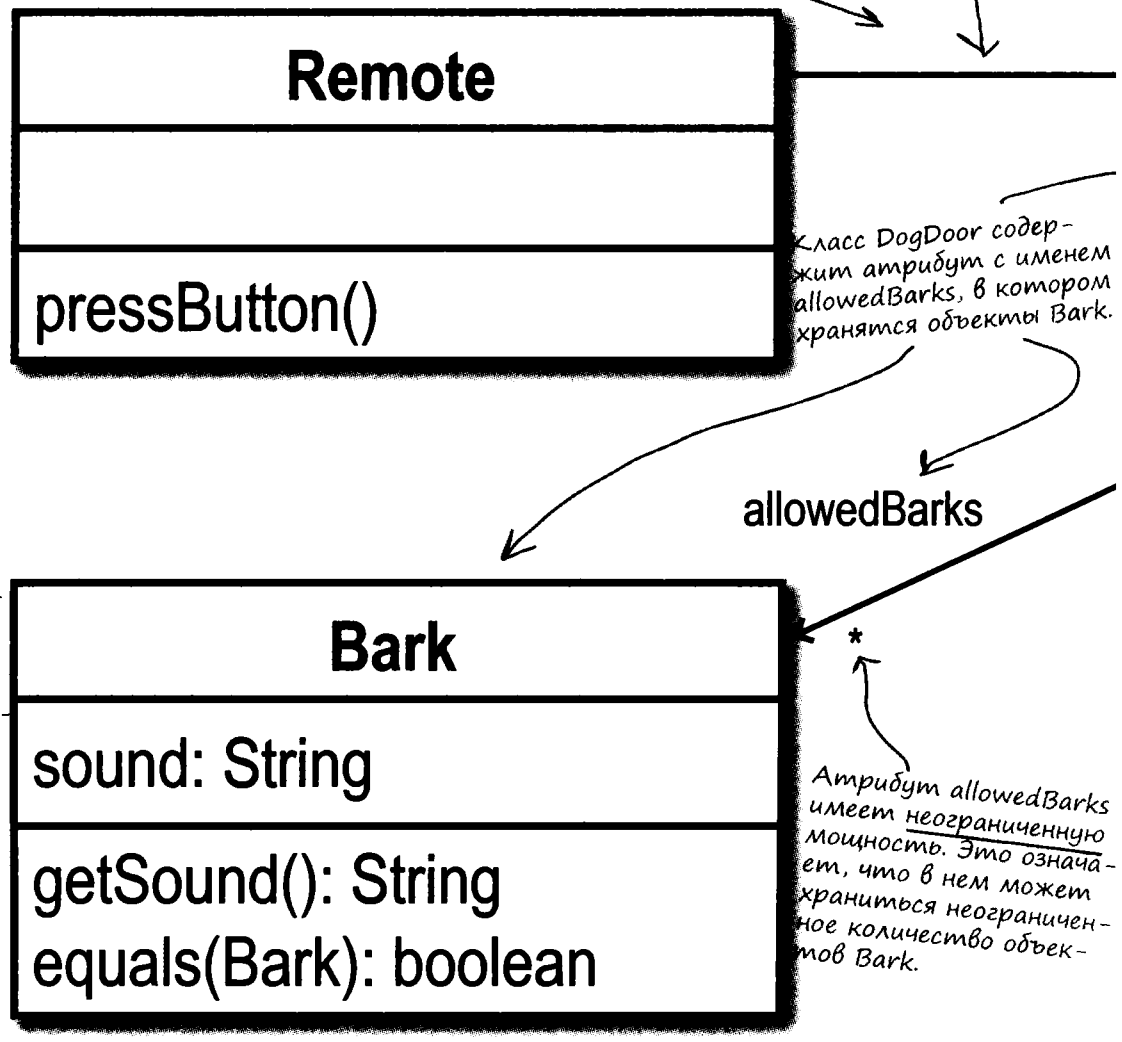
Ответы на с. 212.

Подробнее о диаграммах классов

Возможности диаграмм классов не ограничиваются прямоугольниками и текстом. Линии и стрелки способны значительно расширить информацию, передаваемую диаграммой классов.

Сплошная линия, соединяющая один класс с другим, называется ассоциацией. Она означает, что класс ассоциирован с другим классом — посредством включения ссылки, расширения, наследования и т. д.

Эта линия соединяет класс Remote (исходный класс) с классом DogDoor (целевой класс). Она означает, что класс Remote содержит атрибут типа DogDoor.



Когда для представления атрибутов используются ассоциации, представляемые этими ассоциациями атрибуты обычно не записываются в разделе атрибутов класса. Именно поэтому мы не видим в классе Remote атрибут door.

Класс DogDoor содержит атрибут с именем allowedBarks, в котором хранятся объекты Bark.

Атрибут allowedBarks имеет неограниченную мощность. Это означает, что в нем может храниться неограниченное количество объектов Bark.

Имя атрибута в исходном классе записывается здесь, на целевом конце линии. Таким образом, класс Remote содержит атрибут с именем door и типом DogDoor.

Это число определяет мощность ассоциации, то есть количество значений целевого типа, хранящихся в атрибуте исходного класса. В нашем случае атрибут door содержит одно значение DogDoor.

door 1

DogDoor

open: boolean

open()
close()
isOpen(): boolean
addAllowedBark(Bark)
getAllowedBarks(): Bark [*]

Сравните с диаграммой Марии на с. 208. Классы находятся в разных местах, но это ТА ЖЕ САМАЯ диаграмма классов. Расположение классов на диаграмме несущественно.

Ответ на с. 213.

Возьми в руку карандаш

Руководствуясь приведенной выше диаграммой классов, какие типы вы бы предложили использовать для атрибута **allowedBarks** класса **DogDoor** class? Запишите свои предложения:

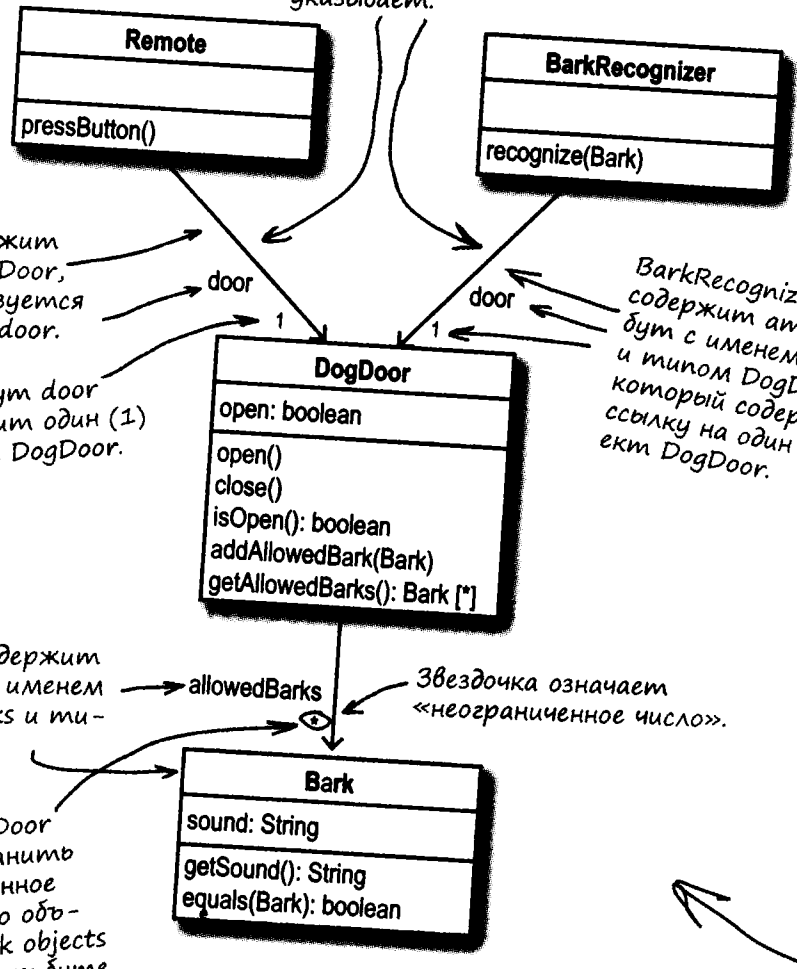
_____

UML-диаграмма. Решение

Мария явно увлеклась своими диаграммами UML... Удастся ли вам разобраться в том, что она сделала?

На первый взгляд класс Remote не имеет атрибутов... Но когда один класс содержит ссылку на другой класс, эта связь представляет атрибут. Следовательно, класс Remote все же содержит один атрибут.

Линия проводится от класса, содержащего ссылку, к классу объекта, на который ссылка указывает.



Класс Remote содержит ссылку на класс DogDoor, для которой используется атрибут с именем door.

Атрибут door содержит один (1) объект DogDoor.

BarkRecognizer содержит атрибут с именем door и типом DogDoor, который содержит ссылку на один объект DogDoor.

DogDoor содержит атрибут с именем allowedBarks и типом Bark.

Звездочка означает «неограниченное число».

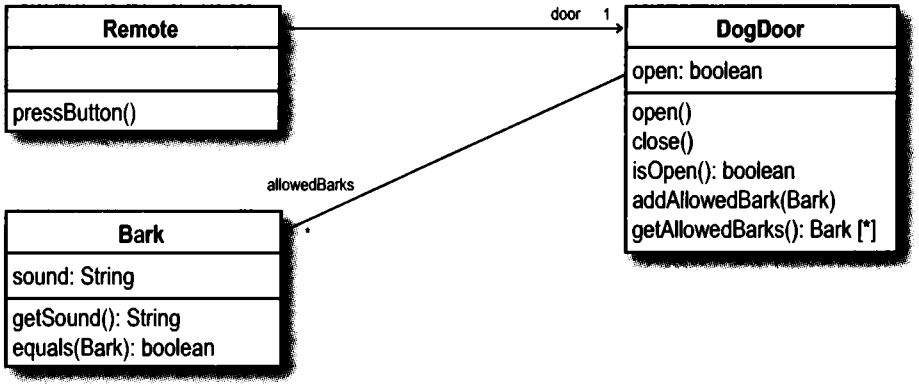
Класс DogDoor может хранить неограниченное количество объектов Bark objects в своем атрибуте allowedBarks.

Возьми в руку карандаш
Решение

Руководствуясь приведенной выше диаграммой классов, какие типы вы бы предложили использовать для атрибута **allowedBarks** класса **DogDoor** class? Запишите свои предложения:

List, Array, Vector и т. д.

Здесь можно записать любой тип, поддерживающий множественные значения... Подойдет почти любой из классов коллекций Java.



Обратите внимание: несмотря на другое размещение элементов, обе эти диаграммы содержат одни и те же классы и ассоциации.

Я до сих пор не пойму, для чего нужны эти диаграммы классов...

Рэнди: Может, я и упустил создание класса Bark, но мое решение было не таким уж плохим, и я не тратил время на рисование квадратиков со стрелочками.

Мария: Ты никогда не слышал, что картинка стоит тысячи слов? Когда у меня появилась диаграмма классов, я стала достаточно четко представлять, как будет работать моя система.

Рэнди: Да, пожалуй, я понимаю... Но и я достаточно четко представлял, как будет работать моя система. Просто картина была у меня в голове, а не на бумаге.

Сэм: Кажется, я начинаю понимать смысл всей этой возни с UML, Рэнди. Я хочу сказать, что когда у тебя есть вариант использования, естественно заняться анализом и преобразовать существительные в классы. А с диаграммой тебе не придется так долго думать над тем, что должно стать классом, а что нет.

Мария: Совершенно верно! Кому захочется написать семейство классов, а потом обнаружить, что что-то было сделано не так? Если у меня есть варианты использования и диаграммы классов, в случае ошибки я просто вычеркиваю неправильные строки и перерисовываю диаграмму.

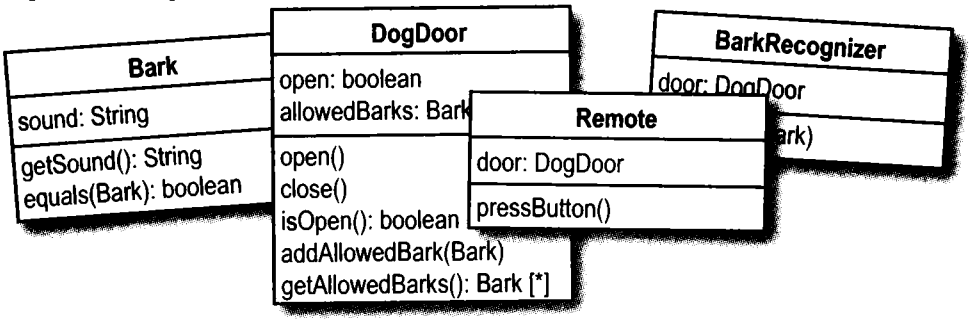
Помните, как ранее мы говорили о том, что ООАП раз за разом помогает создавать хорошие программы? Это один из примеров того, как ООАП способствует уменьшению количества ошибок в коде.

Рэнди: Да, это правда. На переписывание кода уйдет куда больше времени, чем на переделку варианта использования или диаграммы классов...

Мария: И если тебе когда-нибудь придется работать с напарником, то объяснить ему видение системы, находящееся у тебя в голове, будет не так просто, верно?

Сэм: Пожалуй, она права, Рэнди. Я видел твои рисунки, когда ты пытаешься объяснять свои идеи... Там ничего не разобрать!

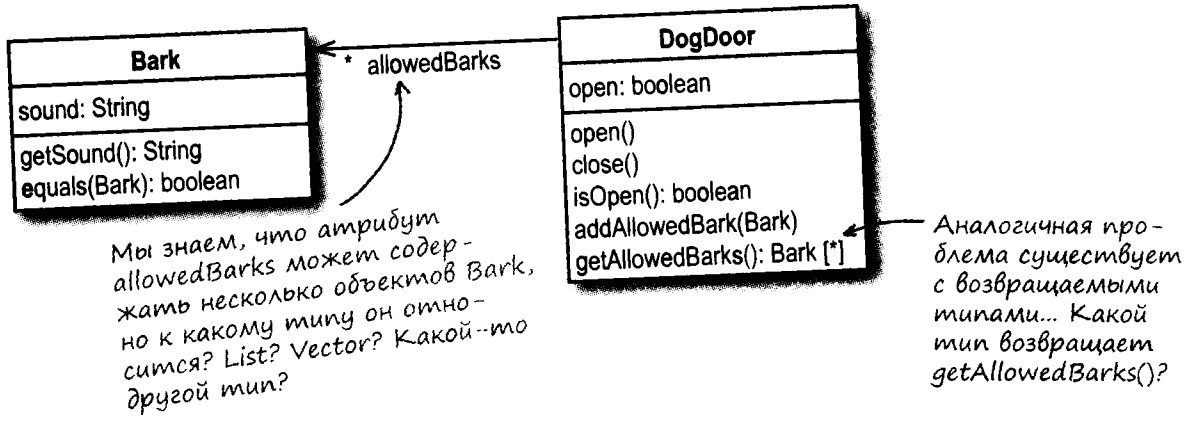
Рэнди: Не стану спорить. Мне кажется, что в диаграмме классов еще не все отражено. Например, как код сравнивает образцы лая и выясняет, должна ли открыться дверь?



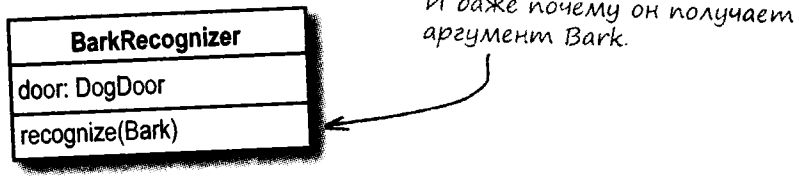
Диаграммы классов — это еще не все

Диаграммы классов дают общее представление о системе и ее структуре вашим коллегам и другим программистам. Однако существует немало информации о классе, которая на них *не отображена.*

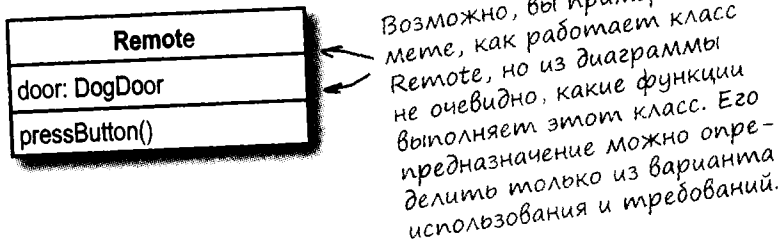
Диаграммы классов предоставляют ограниченную информацию о типах

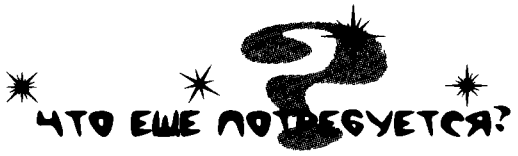


Диаграммы классов не поясняют, как следует программировать методы



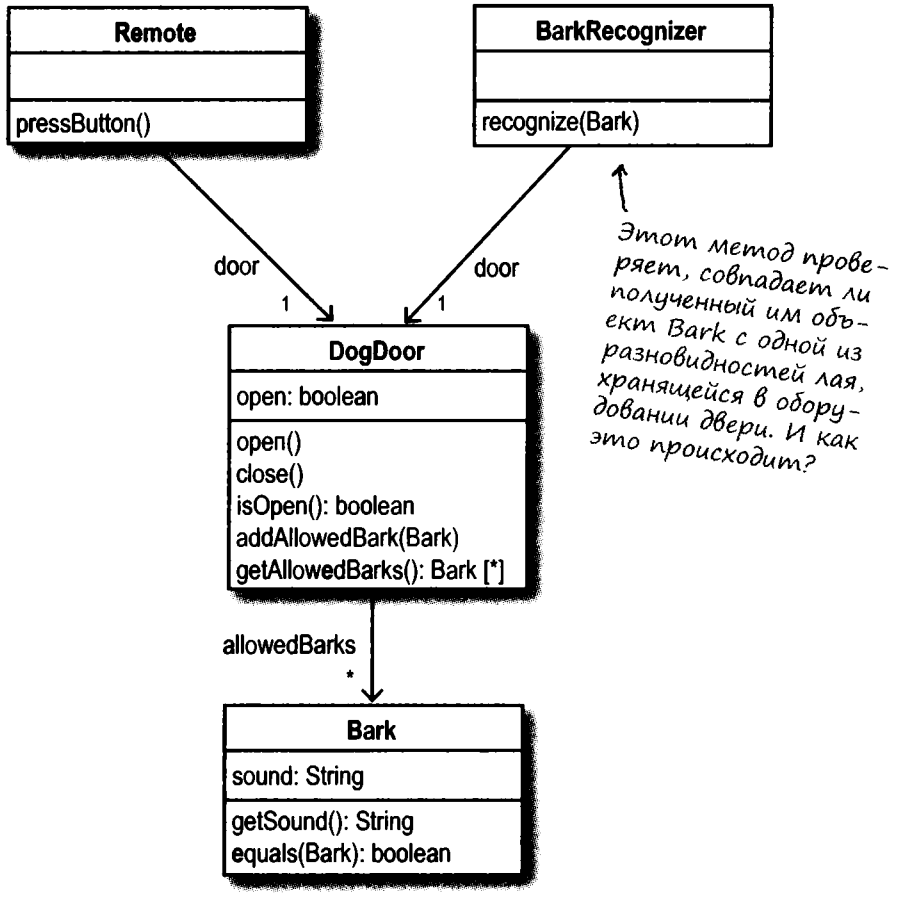
Диаграммы классов дают лишь общее представление о системе





Диаграммы классов прекрасно подходят для моделирования классов, которые необходимо создать, но они не дают всей информации, необходимой для программирования системы. Вы уже видели, что диаграмма классов собачьей двери ничего не сообщает о возвращаемых типах; что еще из того, что может понадобиться для программирования двери, неясно из этой диаграммы?

Пометьте на диаграмме внизу, какая еще информация может потребоваться для программирования. Чтобы вам было проще взяться за дело, мы добавили пометку о сравнении образцов лая.



Ответ на с. 218

И как теперь работает метод recognize()?

Мария определила, что ее класс **BarkRecognizer** должен уметь сравнивать полученный лай с несколькими записанными образцами, но из ее диаграммы классов не видно, как должна выглядеть фактическая реализация метода **recognize()**.

Придется обратиться к коду Марии. Ее метод **recognize()** класса **BarkRecognizer** выглядит так:

```
public void recognize(Bark bark) {
  System.out.println("   BarkRecognizer: Heard a '" +
      bark.getSound() + "'");
  List allowedBarks = door.getAllowedBarks();
  for (Iterator i = allowedBarks.iterator(); i.hasNext(); ) {
    Bark allowedBark = (Bark)i.next();
    if (allowedBark.equals(bark)) {
      door.open();
      return;
    }
  }
  System.out.println("This dog is not allowed.");
}
```

Итератор — объект Java, предназначенный для перебора всех элементов списка.

Мария получает полный список объектов Bark от объекта двери.

Каждый элемент, полученный от итератора, преобразуется в объект Bark.

Как и в коде Сэма, Мария делегирует сравнения Bark объекту Bark.

При обнаружении совпадения перебор прекращается.

Текстологический анализ помог Марии определить, что в ее объекте BarkRecognizer центральное место должна занимать собака, а не ее лай.

door.getAllowedBarks()

door.getAllowedBark()

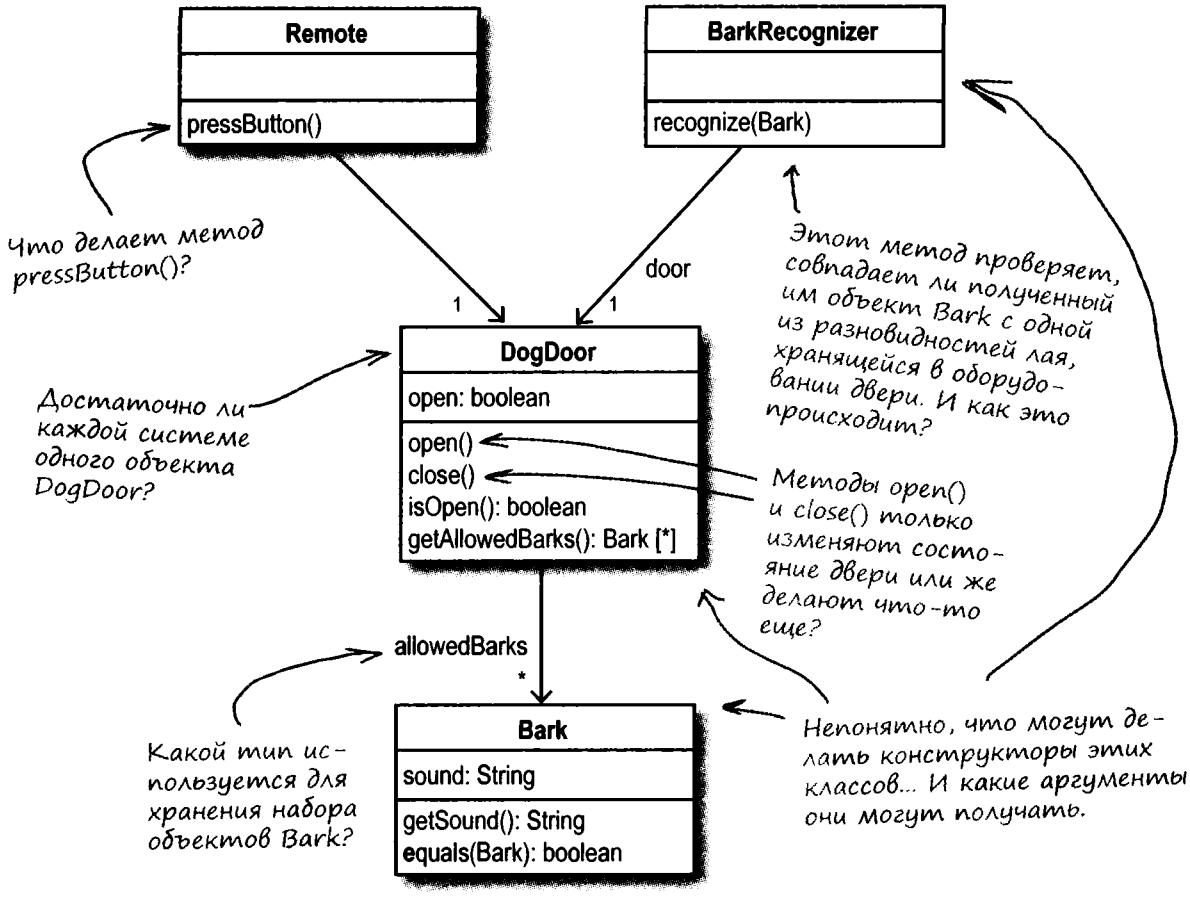
Этот метод представляет собаку: все разновидности лая, которые она может издавать.

А этот метод ориентируется на одну разновидность лая... А не на саму собаку.

Что еще потребуется?

Решение

Пометьте на диаграмме внизу, какая еще информация может потребоваться для программирования.



* Это лишь некоторые вопросы, возникшие у нас при виде диаграммы. Ваше решение может выглядеть совершенно иначе. Здесь важно найти что-то такое, что нельзя понять из диаграммы классов.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Анализ помогает обеспечить работу программного продукта в контексте реального мира, а не в идеальной среде.
- Варианты использования должны быть понятны вам, вашему руководству, заказчикам и другим программистам.
- Вариант использования записывается в любом формате, наиболее удобном для вас и других людей, которые будут с ним работать.
- Хороший вариант использования точно описывает, что делает система, но не указывает, как должна решаться эта задача.
- Каждый вариант использования должен быть сосредоточен на одной цели заказчика. Если целей несколько, необходимо написать несколько вариантов.
- Диаграммы классов дают представление системы и ее программных конструкций «с высоты птичьего полета».
- Атрибуты на диаграмме классов обычно соответствуют переменным класса.
- Операции на диаграмме классов обычно представляют методы ваших классов.
- На диаграммах классов не показаны многие подробности — конструкторы классов, часть информации о типах, цель операций класса.
- Текстологический анализ помогает преобразовать вариант использования в классы, атрибуты и операции уровня программного кода.
- Существительные в вариантах использования являются кандидатами для классов системы, а глаголы — кандидатами для методов классов системы.

Головоломка

Спорим, вы ожидали найти здесь весь код, который я написала? Какая жалость... При переносе файлов на мой новый MacBook Pro почти весь код системы собачьей двери был поврежден. Сможете ли вы восстановить его?

Старый компьютер Марии испортил почти весь код, написанный ею для собачьей двери, — кроме класса **DogDoorSimulator.java**, приведенного на следующей странице. У нас остались только фрагменты ее решения из этой главы, ее диаграммы классов, а также то, что вы узнали об анализе, требованиях и ОО-программировании. Ваша очередь стать героем...

Проблема:

Нужно запрограммировать приложение собачьей двери, чтобы оно соответствовало запросам всех новых клиентов Дуга (а это множество потенциальных продаж!), особенно тех, у которых в окрестностях живет несколько собак. Дверь должна работать в соответствии с вариантами использования этой главы, описывающими систему.

Ваша задача:

1. Начните с воссоздания приложения в том виде, в каком оно было описано в главе 3. Если вы хотите поскорее взяться за дело, код можно скачать с сайта Head First Labs.
2. Скопируйте или загрузите файл **DogDoorSimulator.java**, приведенный на следующей странице. Это единственный файл, переживший катастрофу на старом компьютере Марии.
3. Убедитесь в том, что ваш код соответствует диаграмме классов Марии на с. 220.
4. Приступайте к программированию! Для начала все классы дложны успешно компилироваться, чтобы вы могли заняться тестированием.
5. При помощи класса **DogDoorSimulator** проверьте, все ли работает так, как должно.
6. Продолжайте анализ и программирование до тех пор, пока результат тестового класса не совпадет с результатом на следующей странице. Не сдавайтесь!
7. А когда, по вашему мнению, ваше приложение заработает, сравните свой код с нашим кодом на сайте Head First Labs. Не торопитесь, мы подождем.

ГДЕ МОЕ ОПРЕДЕЛЕНИЕ?

В UML и вариантах использования много терминов, похожих на уже знакомые вам термины из области программирования, но не эквивалентные им. Ниже перечислены некоторые термины из области ООАП и их определения. Соедините каждый термин с его определением и наведите порядок.

АНАЛИЗ СУЩЕСТВИТЕЛЬНЫХ	Перечисление всех конструкций уровня программного кода с атрибутами и операциями.
МОЩНОСТЬ	Термин UML, обычно соответствующий методу одного из ваших классов.
АТРИБУТ	Помогает определить кандидатов в методы для объектов вашей системы.
ДИАГРАММА КЛАССОВ	Визуально представляет связь одного класса с другим (часто посредством атрибута).
ОПЕРАЦИЯ	Аналог переменной в классе.
АССОЦИАЦИЯ	Определяет количество значений некоторого типа, которые могут храниться в атрибуте класса.
АНАЛИЗ ГЛАГОЛОВ	Применяется с вариантами использования для определения набора классов, входящих в систему.

```
public class DogDoorSimulator {

  public static void main(String[] args) {
    DogDoor door = new DogDoor();
    door.addAllowedBark(new Bark("rowlf"));
    door.addAllowedBark(new Bark("rooowlf"));
    door.addAllowedBark(new Bark("rawlf"));
    door.addAllowedBark(new Bark("woof"));
    BarkRecognizer recognizer = new BarkRecognizer(door);
    Remote remote = new Remote(door);

    // Имитация получения лая оборудованием
    System.out.println("Bruce starts barking.");
    recognizer.recognize(new Bark("rowlf"));

    System.out.println("\nBruce has gone outside...");

    try {
      Thread.currentThread().sleep(10000);
    } catch (InterruptedException e) { }

    System.out.println("\nBruce's all done...");
    System.out.println("...but he's stuck outside!");

    // Имитация получения постороннего лая (не Брюса!)
    Bark smallDogBark = new Bark("yip");
    System.out.println("A small dog starts barking.");
    recognizer.recognize(smallDogBark);

    try {
      Thread.currentThread().sleep(5000);
    } catch (InterruptedException e) { }

    // Имитация повторного получения лая
    System.out.println("Bruce starts barking.");
    recognizer.recognize(new Bark("rooowlf"));

    System.out.println("\nBruce's back inside...")
  }
}
```

DogDoorSimulator.java

Тестовый класс со старого компьютера Марии. Используйте его в своем тестировании собачьей двери.

```
File Edit Window Help HollyLovesBruce
%java DogDoorSimulator
Bruce starts barking.
   BarkRecognizer: Heard a 'rowlf'
The dog door opens.

Bruce has gone outside...
The dog door closes.

Bruce's all done...
...but he's stuck outside!
Bitsie starts barking.
   BarkRecognizer: Heard a 'yip'
This dog is not allowed.
Bruce starts barking.
   BarkRecognizer: Heard a 'rooowlf'
The dog door opens.

Bruce's back inside...
The dog door closes.
```

То, что нужно: дверь работает для Брюса, но не для других собак.

ГДЕ МОЕ ОПРЕДЕЛЕНИЕ?

РЕШЕНИЕ

В UML и вариантах использования много терминов, похожих на уже знакомые вам термины из области программирования, но не эквивалентные им. Ниже перечислены некоторые термины из области ООАП и их определения. Соедините каждый термин с его определением и наведите порядок.

- АНАЛИЗ СУЩЕСТВИТЕЛЬНЫХ
- МОЩНОСТЬ
- АТРИБУТ
- ДИАГРАММА КЛАССОВ
- ОПЕРАЦИЯ
- АССОЦИАЦИЯ
- АНАЛИЗ ГЛАГОЛОВ

- Перечисление всех конструкций уровня программного кода с атрибутами и операциями.
- Термин UML, обычно соответствующий методу одного из ваших классов.
- Помогает определить кандидатов в методы для объектов вашей системы.
- Визуально представляет связь одного класса с другим (часто посредством атрибута).
- Аналог переменной в классе.
- Определяет количество значений некоторого типа, которые могут храниться в атрибуте класса.
- Применяется с вариантами использования для определения набора классов, входящих в систему.

Качественное проектирование =

5 (часть 1) гибкость программного продукта

Все течет, все меняется

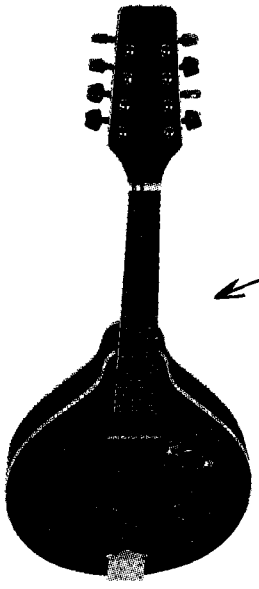
Изменения неизбежны. Какой бы замечательной ни была ваша программа сегодня, завтра ее, скорее всего, придется изменять. И чем труднее вносить поправки в программу, тем сложнее реагировать на изменяющиеся потребности заказчика. В этой главе мы вернемся к старому знакомому, попробуем улучшить существующую программу и увидим, как маленькие изменения создают большие трудности. А при этом мы обнаружим проблему настолько значительную, что для ее решения потребуется глава ИЗ ДВУХ ЧАСТЕЙ!

Струнные инструменты

Фирма «~~Гитары~~ Рика» расширяется

Рику только что удалось продать целых три гитары местной рок-группе. Его дела идут лучше, чем когда-либо, а программа, которую вы написали для него в главе 1, стала краеугольным камнем бизнеса Рика.

Твоя программа великолепна! Я продаю гитары направо и налево. Однако круг моих клиентов расширился, и я хочу продавать еще и мандолины. От покупателей просто не будет отбоя!



Мандолины очень похожи на гитары... Наверное, обеспечить их поддержку будет нетрудно, верно?

Давайте проверим структуру кода на прочность

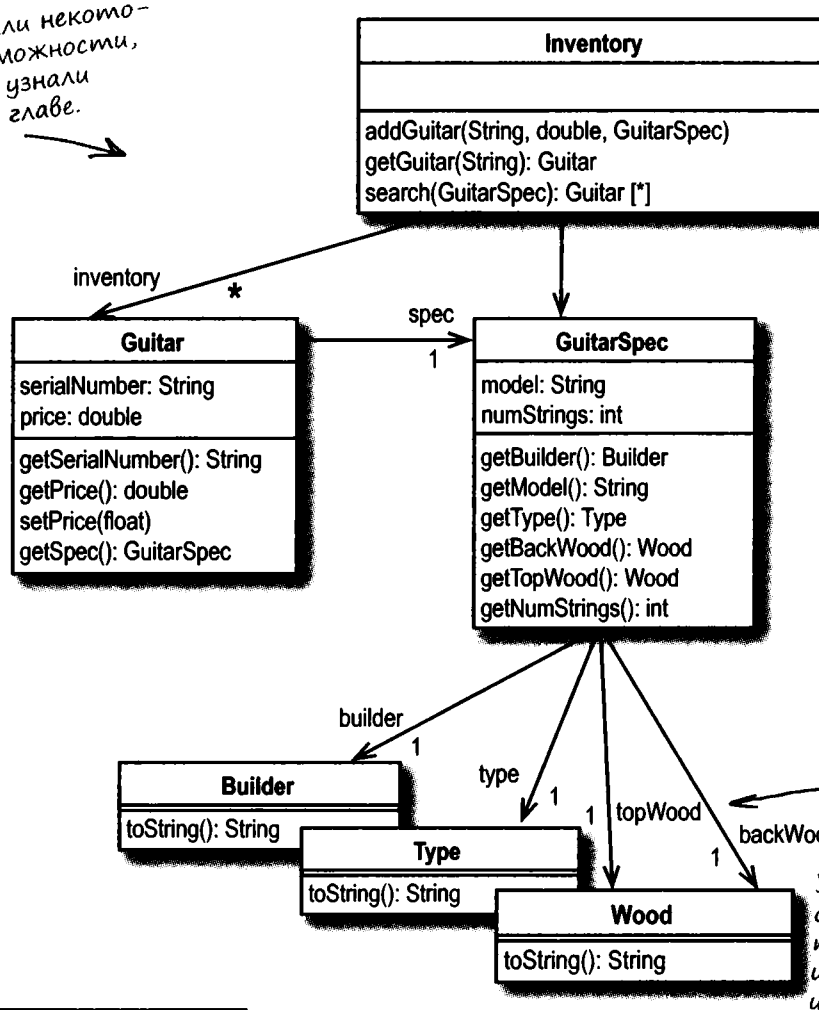
Мы много говорили о том, что качественный анализ и проектирование играют важнейшую роль для построения программ, пригодных для повторного использования и расширения... А теперь, похоже, нам придется доказать это Рику. Давайте попробуем изменить приложение, чтобы оно поддерживало не только гитары, но и мандолины.

Возьми в руку карандаш

Добавьте мандолины в поисковую программу Рика.

Ниже приведена полная диаграмма классов приложения Рика в том виде, в каком мы оставили ее в конце главы 1. Дополните эту диаграмму, чтобы Рик мог продавать новый товар, а ваше приложение помогало ему находить мандолины, соответствующие требованиям покупателя (так же, как с гитарами).

Мы использовали некоторые новые возможности, о которых вы узнали в предыдущей главе.



Большинство свойств исключено из прямоугольника класса, вместо них используются ассоциации.

Учтите, что свойства можно записывать на любой из сторон ассоциации... «Единственно правильной записи» не существует; используйте любой удобный способ.

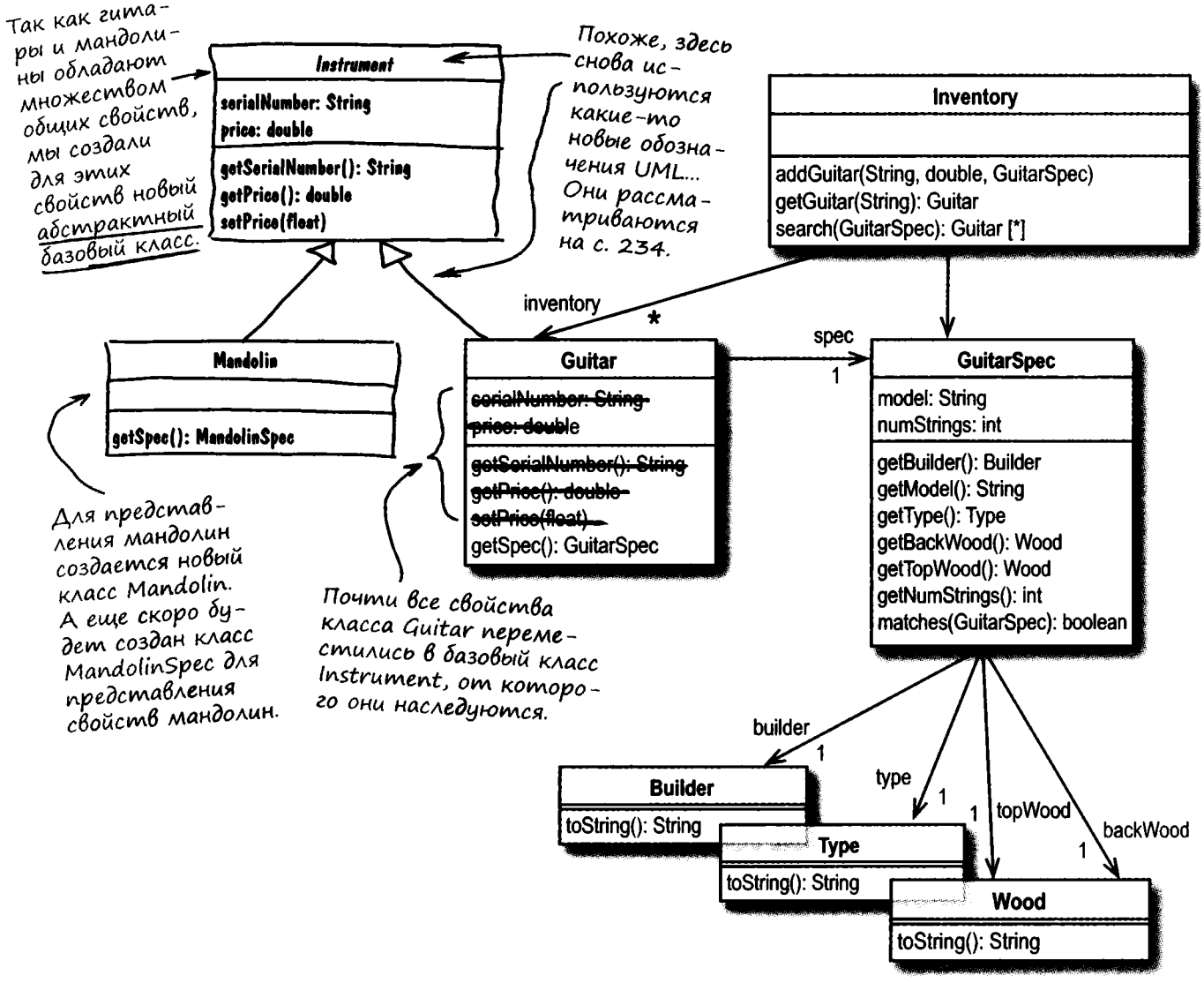
ПОДСКАЗКА: Если вы у гитары и мандолины что-нибудь общее? И если есть, почему бы не использовать наследование?

Возьми в руку карандаш

Решение

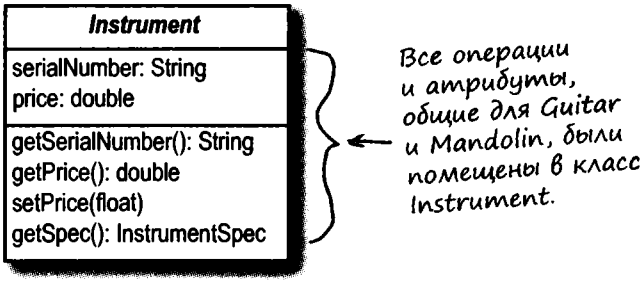
Добавьте мандолины в поисковую программу Рика.

Ниже приведена полная диаграмма классов приложения Рика в том виде, в каком мы оставили ее в конце главы 1. Дополните эту диаграмму, чтобы Рик мог продавать новый товар, а ваше приложение помогало ему находить мандолины, соответствующие требованиям покупателя (так же, как с гитарами).

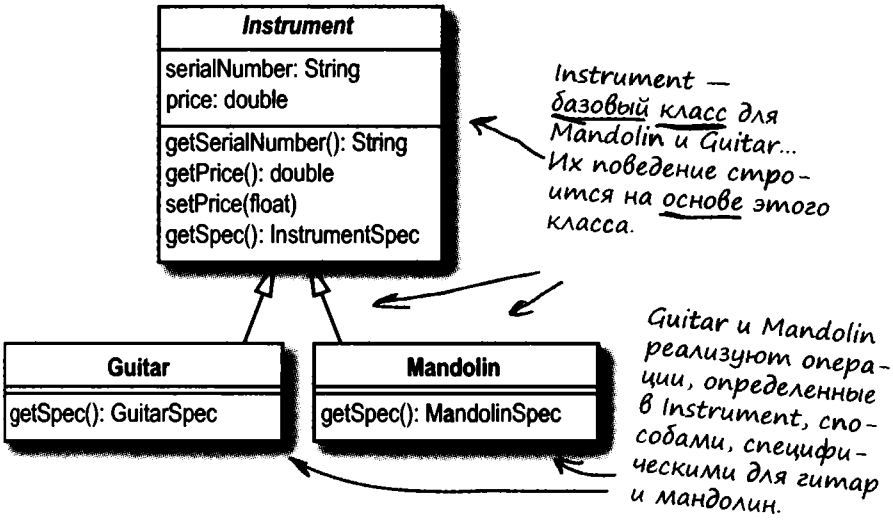


Вы обратили внимание на абстрактный базовый класс?

Присмотримся повнимательнее к созданному классу **Instrument**:



Класс **Instrument** является абстрактным; это означает, что создать экземпляр **Instrument** в программе невозможно. Необходимо определить субклассы, производные от **Instrument**, как сделано с **Mandolin** и **Guitar**:



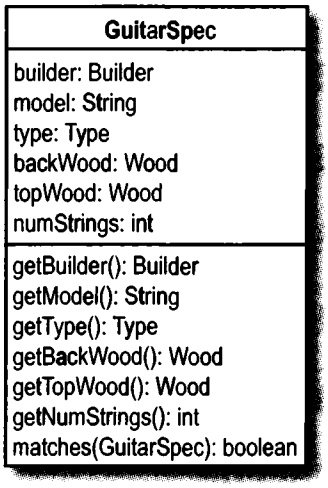
Класс **Instrument** объявлен абстрактным, потому что он лишь хранит подмножество свойств классов реальных инструментов (таких, как **Guitar** и **Mandolin**). Абстрактный класс определяет базовое поведение, но реализация этого поведения добавляется в субклассах абстрактного класса. **Instrument** — всего лишь обобщенный класс, который представляет фактические классы реализации.

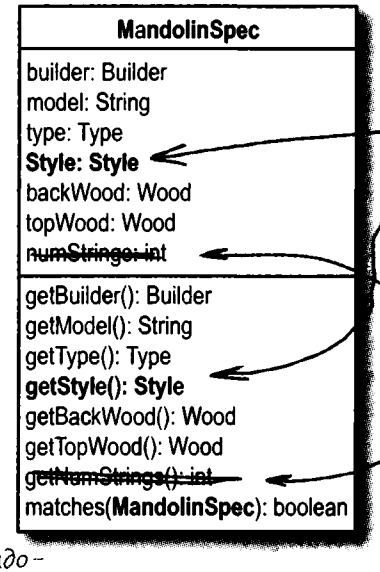
Абстрактные классы могут рассматриваться как условные «представители» конкретных классов реализации.

Абстрактный класс определяет поведение, а субклассы это поведение реализуют.

Нам также понадобится класс MandolinSpec

У мандолин и гитар много общего, но у мандолин имеются свои отличительные особенности... В нашем приложении эти особенности будут отражены в классе **MandolinSpec**:

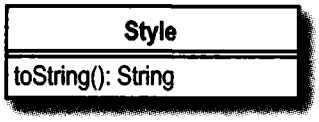




Существует несколько разновидностей (стилей) мандолин, например стиль «A» или стиль «F».

У мандолины 4 пары струн (всего 8), поэтому свойство numStrings не потребуется.

По аналогии с перечисляемыми типами для дерева и изготовителя определяем новый тип для стиля.



Если вы ничего не знаете о мандолинах или выбрали другие свойства в классеMandolinSpec — ничего страшного. Главное — понять, что для представления мандолин и их набора свойств понадобится новый класс. А если вы использовали абстрактный класс или интерфейс Instrument — еще лучше!

Классы спецификаций очень похожи. Может быть, стоит использовать абстрактный базовый класс?

МОЗГОВОЙ ШТУРМ

Что вы думаете о такой структуре? Будет ли она делать то, что нужно? Насколько она гибка? Будет ли программа, спроектированная подобным образом, простой в расширении и сопровождении?

_____

_____

Часто задаваемые вопросы

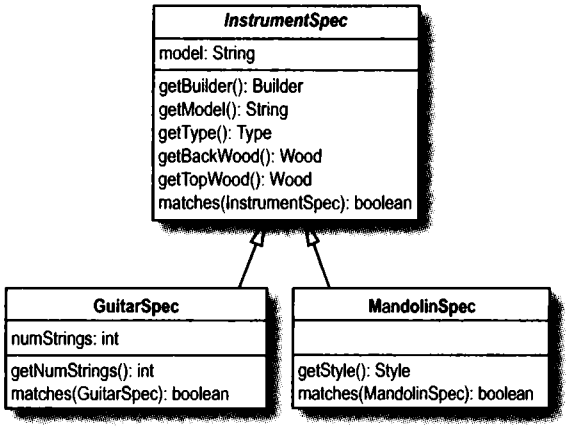
В: Мы сделали класс Instrument абстрактным, потому что абстрагировали в него свойства, общие для Guitar и Mandolin?

О: Нет, мы сделали класс `Instrument` абстрактным потому, что в настоящее время в системе Рика не существует понятия «инструмент». Этот класс всего лишь предоставляет место для хранения свойств, существующих как в классе `Guitar`, так и в `Mandolin`. А поскольку инструмент в настоящее время не обладает никаким поведением за пределами своих субклассов, по сути, он всего лишь определяет общий набор атрибутов и свойств, которые должны быть реализованы всеми инструментами.

Итак, мы абстрагировали свойства, общие для обоих типов инструментов, но это не означает, что класс `Instrument` должен быть абстрактным. Ничто не мешает нам позднее преобразовать класс `Instrument` в конкретный, если это будет оправдано логикой проектирования...

В: Почему бы не сделать то же самое с GuitarSpec и MandolinSpec? Похоже, эти классы содержат множество общих атрибутов и операций, как и Guitar с Mandolin.

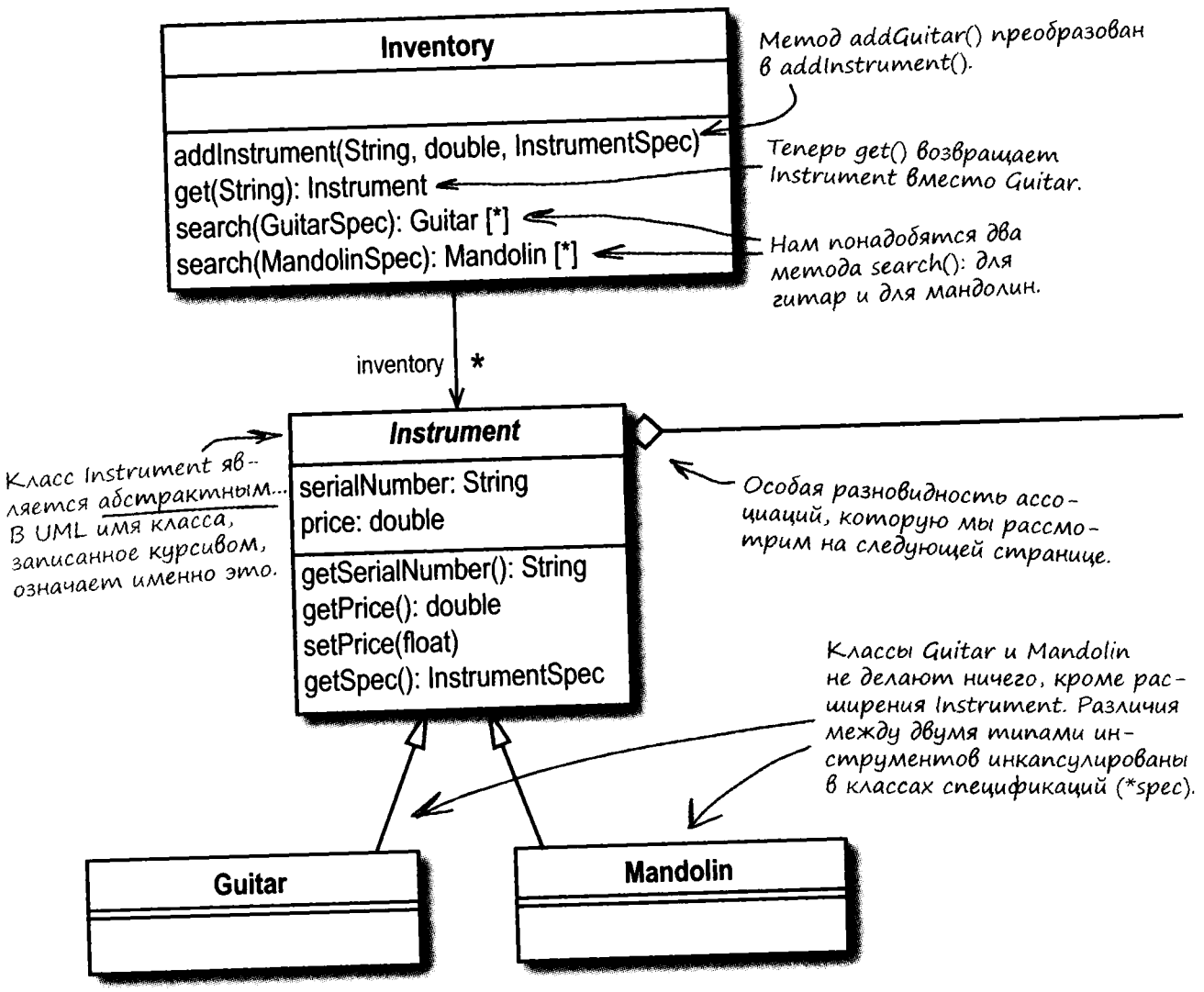
О: Отличная идея! Мы создадим еще один абстрактный базовый класс с именем `InstrumentSpec`, от которого будут наследовать `GuitarSpec` и `MandolinSpec`:



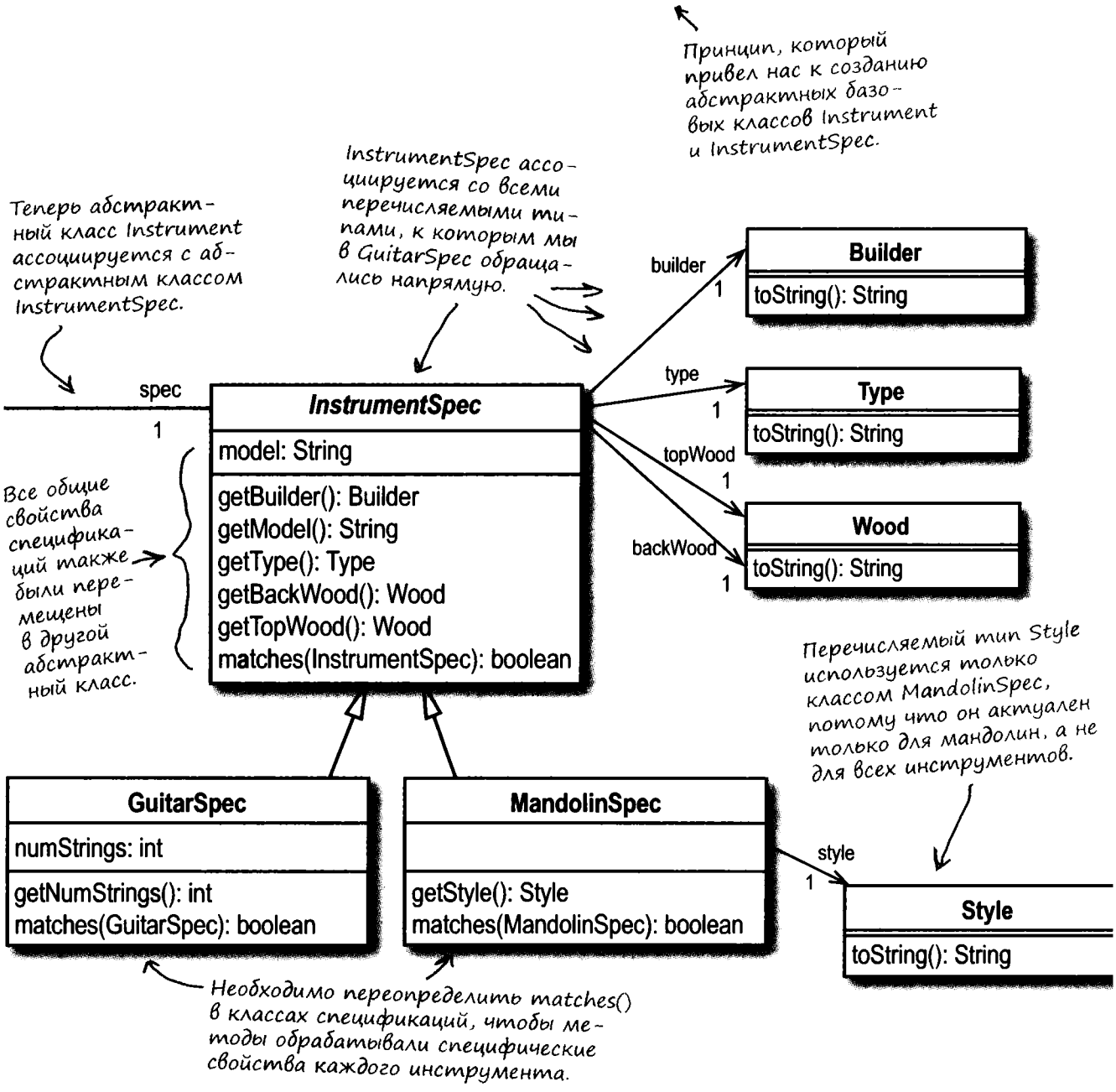
Давайте соберем все воедино...

Новое приложение Рика

Похоже, наша работа по проектированию в главе 1 окупилась; чтобы добавить поддержку мандолин в поисковую программу Рика, нам понадобилось менее 10 страниц. Итоговая диаграмма классов выглядит так:



Когда в двух и более местах обнаруживается общее поведение, постарайтесь абстрагировать это поведение в отдельный класс.



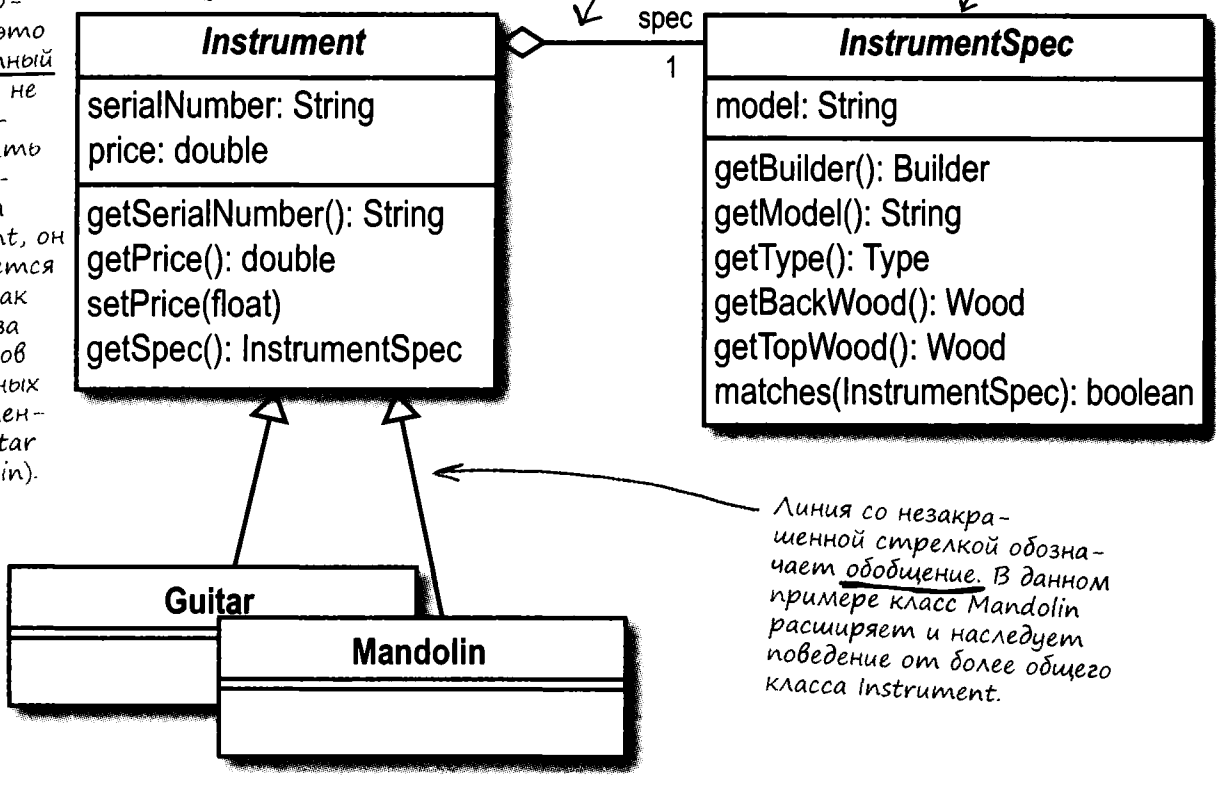
Как устроены диаграммы классов (снова)

Итак, с появлением абстрактных классов, субклассов и новой разновидности ассоциаций пришло время обновить наши навыки построения диаграмм классов и UML.

Линия с ромбиком обозначает агрегирование — особую разновидность ассоциаций, при которой один объект включает в себя другой объект. В данном случае Instrument включает в себя InstrumentSpec.

Больше курсива: InstrumentSpec тоже абстрактный.

Если имя класса выделено курсивом — это абстрактный класс. Мы не собираемся создавать экземпляры класса Instrument, он используется только как общая база для классов конкретных инструментов (Guitar и Mandolin).



Линия со незакрашенной стрелкой обозначает обобщение. В данном примере класс Mandolin расширяет и наследует поведение от более общего класса Instrument.

Загните уголок страницы, чтобы быстро вернуться к ней, если вы забудете какие-либо UML-обозначения.

Шпаргалка по UML

Как называется в Java	Как называется в UML	Как изображается в UML
Абстрактный класс	Абстрактный класс	*Курсивное имя класса*
Отношение	Ассоциация	——→
Наследование	Обобщение	——▷
Агрегирование	Агрегирование	——◇

Часто задаваемые вопросы

В: И сколько еще условных знаков и обозначений мне придется запомнить, чтобы использовать UML?

О: В UML существует множество условных знаков и обозначений, но вы сами решаете, сколько из них использовать в своей работе (не говоря уже о запоминании). Многие разработчики ограничиваются малым подмножеством того, что вы уже узнали, и их это полностью устраивает (как и их заказчиков и начальство). Другие предпочитают изучить UML во всех тонкостях и использовать весь инструментарий UML. Решайте сами: если вам удается адекватно передать смысл своего решения, значит, вы правильно используете UML.

Напишем код новой поисковой программы Puka

Начнем с создания нового класса **Instrument**, который будет абстрактным. В этот класс включаются свойства, общие для инструментов:

Мы использовали агрегатную форму ассоциации, потому что каждый объект Instrument содержит переменные serialNumber и price, а также экземпляр InstrumentSpec.

```
public abstract class Instrument {

  private String serialNumber;
  private double price;
  private InstrumentSpec spec;

  public Instrument(String serialNumber, double price,
                    InstrumentSpec spec) {
    this.serialNumber = serialNumber;
    this.price = price;
    this.spec = spec;
  }

  // Методы get и set для серийного номера и цены

  public InstrumentSpec getSpec() {
    return spec;
  }
}
```

Instrument — абстрактный класс... В программе создаются экземпляры только его субклассов (например, Guitar).

Большая часть кода проста и похожа на старый класс Guitar.

Instrument
serialNumber: String
price: double
getSerialNumber(): String
getPrice(): double
setPrice(float)
getSpec(): InstrumentSpec

Instrument.java

Затем необходимо переработать **Guitar.java** и создать класс для мандолин. Оба класса расширяют **Instrument** для получения общих свойств инструментов, а затем определяют собственные конструкторы с правильным типом класса spec:

Класс инструмента должен расширять Instrument и предоставлять конструктор, получающий правильный тип объекта spec.

```
public class Guitar extends Instrument {

  public Guitar(String serialNumber, double price,
                GuitarSpec spec) {
    super(serialNumber, price, spec);
  }
}
```

```
public class Mandolin extends Instrument {

umber, double price,
c) {
pec);
```

Guitar

Guitar.java

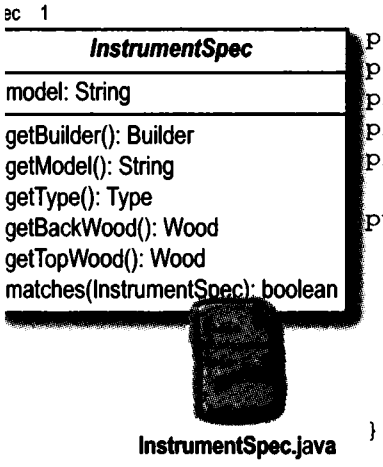
Mandolin

Mandolin.java

Класс Mandolin почти не отличается от Guitar, просто он получает в конструкторе MandolinSpec вместо GuitarSpec.

Создание абстрактного класса для спецификаций инструментов

Разобравшись с классами инструментов, можно переходить к классу спецификаций. Так как многие инструменты обладают одинаковыми спецификациями, мы создадим еще один абстрактный класс **InstrumentSpec**:




```
public abstract class InstrumentSpec {

  private Builder builder;
  private String model;
  private Type type;
  private Wood backWood;
  private Wood topWood;

  public InstrumentSpec(Builder builder, String model, Type type,
                        Wood backWood, Wood topWood) {
    this.builder = builder;
    this.model = model;
    this.type = type;
    this.backWood = backWood;
    this.topWood = topWood;
  }

  // Все методы get для изготовителя, модели, типа и т. д.

  public boolean matches(InstrumentSpec otherSpec) {
    if (builder != otherSpec.builder)
      return false;
    if ((model != null) && (!model.equals("")) &&
        (!model.equals(otherSpec.model)))
      return false;
    if (type != otherSpec.type)
      return false;
    if (backWood != otherSpec.backWood)
      return false;
    if (topWood != otherSpec.topWood)
      return false;
    return true;
  }
}
```

Класс InstrumentSpec, как и Instrument, является абстрактным, а для каждой разновидности инструментов создается отдельный субкласс.

Очень похоже на старый конструктор Guitar...

...но из класса выведены свойства, отсутствующие у некоторых инструментов (например, numStrings и style).

Эта версия matches() работает вполне привычным образом: она сравнивает все свойства текущего экземпляра со свойствами другого экземпляра. Впрочем, в субклассах ее нужно будет переопределить.

А теперь запрограммируем GuitarSpec...

С готовым классом **InstrumentSpec** запрограммировать класс **GuitarSpec** будет несложно:

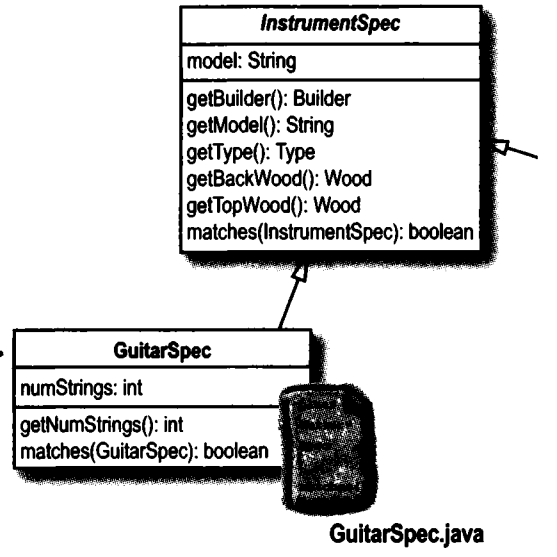
GuitarSpec расширяет InstrumentSpec (по аналогии с тем, как Guitar расширяет Instrument).

```
public class GuitarSpec extends InstrumentSpec {

  private int numStrings;

  public GuitarSpec(Builder builder, String model, Type type,
                    int numStrings, Wood backWood, Wood topWood) {
    super(builder, model, type, backWood, topWood);
    this.numStrings = numStrings;
  }

  public int getNumStrings() {
    return numStrings;
  }

  // Переопределение метода matches() суперкласса
  public boolean matches(InstrumentSpec otherSpec) {
    if (!super.matches(otherSpec))
      return false;
    if (!(otherSpec instanceof GuitarSpec))
      return false;
    GuitarSpec spec = (GuitarSpec)otherSpec;
    if (numStrings != spec.numStrings)
      return false;
    return true;
  }
}
```

Свойство numStrings имеется только у гитар, в суперклассе Instrument его быть не должно.

Конструктор добавляет специфические свойства гитар к данным, которые уже хранятся в базовом классе InstrumentSpec.

matches() использует метод matches() суперкласса, а затем дополнительно проверяет правильность переданного типа и свойства, специфические для гитар.



GuitarSpec теперь наследует большую часть поведения от InstrumentSpec, так что код GuitarSpec стал существенно компактнее по сравнению с кодом из главы 1.

GuitarSpec.java

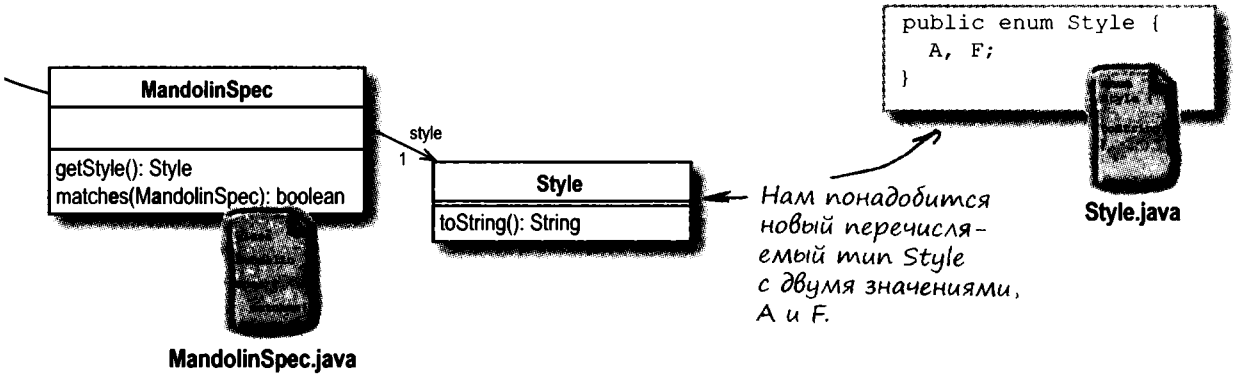
...и метода MandolinSpec

После **GuitarSpec** разобраться в методе **MandolinSpec** будет несложно. Эти два метода очень похожи, если не считать добавления переменной для стиля мандолины (например, «A» или «F») и слегка отличающегося метода **matches()**:

```
public class MandolinSpec extends InstrumentSpec {

  private Style style;

  public MandolinSpec(Builder builder, String model, Type type,
                      Style style, Wood backWood, Wood topWood) {
    super(builder, model, type, backWood, topWood);
    this.style = style;
  }

  public Style getStyle() {
    return style;
  }

  // Переопределение метода matches() суперкласса
  public boolean matches(InstrumentSpec otherSpec) {
    if (!super.matches(otherSpec))
      return false;
    if (!(otherSpec instanceof MandolinSpec))
      return false;
    MandolinSpec spec = (MandolinSpec)otherSpec;
    if (!style.equals(spec.style))
      return false;
    return true;
  }
}
```

Свойство Style имеется только у мандолин, поэтому оно не было вынесено в базовый класс InstrumentSpec.

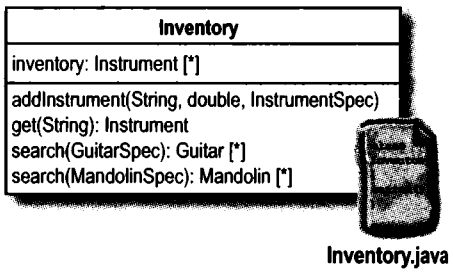
MandolinSpec, как и GuitarSpec, использует свой суперкласс для проверки основных свойств, а затем осуществляет преобразование к типу MandolinSpec и сравнивает свойства, специфические для мандолин.



Нам понадобится новый перечисляемый тип Style с двумя значениями, A и F.

Завершение работы над поисковой программой

Осталось обновить класс **Inventory**, чтобы он работал с разными типами инструментов вместо одного класса **Guitar**:



Теперь каталог содержит инструменты разных типов, а не только гитары.

```
public class Inventory {

  private List inventory;

  public Inventory() {
    inventory = new LinkedList();
  }

  public void addInstrument(String serialNumber, double price,
                            InstrumentSpec spec) {
    Instrument instrument = null;
    if (spec instanceof GuitarSpec) {
      instrument = new Guitar(serialNumber, price, (GuitarSpec)spec);
    } else if (spec instanceof MandolinSpec) {
      instrument = new Mandolin(serialNumber, price, (MandolinSpec)spec);
    }
    inventory.add(instrument);
  }

  public Instrument get(String serialNumber) {
    for (Iterator i = inventory.iterator(); i.hasNext(); ) {
      Instrument instrument = (Instrument)i.next();
      if (instrument.getSerialNumber().equals(serialNumber))
        return instrument;
      }
    }
    return null;
  }

  // search(GuitarSpec) работает так же, как прежде

  public List search(MandolinSpec searchSpec) {
    List matchingMandolins = new LinkedList();
    for (Iterator i = inventory.iterator(); i.hasNext(); ) {
      Mandolin mandolin = (Mandolin)i.next();
      if (mandolin.getSpec().matches(searchSpec))
        matchingMandolins.add(mandolin);
    }
    return matchingMandolins;
  }
}
```

Используя классы Instrument и InstrumentSpec, мы можем преобразовать addGuitar() в более общий метод, создающий инструмент любого типа.

Хмм... А вот это неудобно. Так как класс Instrument является абстрактным, создать его экземпляр напрямую мы не можем, поэтому перед созданием инструмента приходится выполнять дополнительную работу.

Еще одно место, в котором использование абстрактного базового класса делает структуру более гибкой.

Нам понадобится еще один метод search() для мандолин.

Сейчас все готово к тому, чтобы опробовать обновленное приложение Рика. Посмотрите, удастся ли вам самостоятельно обновить FindGuitarTester, и проверьте работу приложения с измененной структурой.

Часто задаваемые вопросы

В: Классы Guitar и Mandolin содержат только конструктор. Это выглядит довольно странно. Действительно ли нужно создавать субкласс для каждой разновидности инструментов только для этого?

О: Нужно... По крайней мере сейчас. В противном случае как вы собираетесь отличать мандолину от гитары? Тип инструмента, с которым вы работаете, можно определить только одним способом: проверкой типа класса. Кроме того, эти субклассы позволяют нам проверить правильность типа переданного объекта спецификации в конструкторе. Например, нельзя создать объект `Guitar` и передать его конструктору `MandolinSpec`.

В: Но с абстрактным классом Instrument метод addInstrument() в Inventory.java выглядит просто ужасно!

О: Речь идет о методе `addInstrument()` на с. 240, не так ли? Да, с абстрактным классом `Instrument` придется написать немного дополнительного кода. И все же это небольшая цена за предотвращение создания экземпляров `Instrument`, которые не существуют в реальном мире.

Все это указывает на недостатки проектирования. Когда в структуре кода приложения что-то выглядит нелогично, проблему стоит изучить поглубже... Чем мы сейчас и займемся.

В: А здесь возможны только крайности? В смысле, даже если в системе не существует концепции «инструмента», который бы не являлся гитарой, мандолиной или еще чем-нибудь, мне все равно кажется, что с нашей структурой что-то не в порядке. Разве не так?

О: Возможно, в этом есть здравое зерно. Похоже, для одних частей нашего кода конкретный класс `Instrument` был бы полезен, а для других — нет. Иногда это означает, что вы должны принять то или иное решение и смириться с неудобствами. А может, происходит что-то такое, о чем мы не подумали...

В: Для чего нужны две разные версии search()? Нельзя ли объединить их в один метод, получающий InstrumentSpec?

О: Класс `InstrumentSpec` является абстрактным, как и класс `Instrument`, поэтому клиенты Рика должны передать методу `search()` в `Inventory` объект `GuitarSpec` или `MandolinSpec`. А поскольку спецификация может совпасть только с другими спецификациями того же типа инструментов, ситуация, при которой список подходящих инструментов включает мандолины и гитары, в принципе невозможна. Следовательно, объединение двух методов `search()` не принесет никаких функциональных преимуществ; более того, все будет выглядеть так, словно метод возвращает мандолины вместе с гитарами (так как возвращаемым типом `search()` будет `Instrument [*]`), хотя этого никогда не происходит.

Да, это уже на что-то похоже! Использование абстрактных классов помогло избежать дублирования кода, а свойства инструментов были инкапсулированы в классах спецификаций.

Мы внесли ПРИНЦИПИАЛЬНЫЕ усовершенствования в приложение Рика

Мы не просто добавили в приложение Рика поддержку мандолин: абстрагирование общих свойств и поведения в классах **`Instrument`** и **`InstrumentSpec`** сократило зависимость классов приложения друг от друга. Это существенное усовершенствование структуры кода.

Не знаю... Мне кажется, что проблем еще хватает — это и почти пустые классы Guitar и Mandolin, и addInstrument() с этим жутким кодом проверки. И мы должны не обращать на это внимания?

Хорошие программы строятся не сразу

Наряду с принципиальными усовершенствованиями мы обнаружили некоторые проблемы в структуре кода приложения. И это нормально... При внесении крупных изменений почти всегда выявляются новые проблемы.

Итак, теперь нужно взять усовершенствованное приложение Рика и посмотреть, нельзя ли его сделать еще лучше... Чтобы превратить обычную программу в ХОРОШУЮ.

3 шага к хорошей программе (еще разок)

Можно ли назвать поисковую программу Рика **хорошей программой**?

Помните три свойства, о которых мы упоминали, говоря о том, как пишутся хорошие программы? Давайте снова вспомним их и посмотрим, в какой степени они применимы к обновленной версии поисковой программы Рика.

1. Делает ли новая поисковая программа то, что она должна делать?

2. Используются ли в ней проверенные ОО-принципы (например, инкапсуляция), которые сокращают дублирование кода и упрощают расширение программы?

3. Легко ли повторно использовать код приложения? Не приводит ли изменение в одной части приложения к многочисленным изменениям в других частях? Обладает ли приложение слабой связностью?

Хорошие программы раз за разом? Не могу даже представить, что такое возможно!

Обязательно напишите ответы на эти вопросы и только потом переверните страницу и сравните их с нашими.

3 шага к хорошей программе (еще разок). Решение

Можно ли назвать поисковую программу Рика **хорошей программой**?

Помните три свойства, о которых мы упоминали, говоря о том, как пишутся хорошие программы? Давайте снова вспомним их и посмотрим, в какой степени они применимы к обновленной версии поисковой программы Рика.

1. Делает ли новая поисковая программа то, что она должна делать?

   Безусловно. Программа находит гитары и мандолины, хотя и не одновременно. Так что, скорее всего, программа делает то, чего от нее ждут. Хотя лучше спросить Рика...

2. Используются ли в ней проверенные ОО-принципы (например, инкапсуляция), которые сокращают дублирование кода и упрощают расширение программы?

   Мы использовали инкапсуляцию при создании классов InstrumentSpec и наследование — при разработке абстрактных суперклассов Instrument и InstrumentSpec. И все же добавление новых типов инструментов требует значительной работы...

3. Легко ли повторно использовать код приложения? Не приводит ли изменение в одной части приложения к многочисленным изменениям в других частях? Обладает ли приложение слабой связностью?

   Использовать отдельные части приложения Рика будет сложновато. Все компоненты сильно связаны друг с другом, а InstrumentSpec фактически является частью Instrument (мы уже говорили об агрегировании).

Похоже, работа еще не закончена... Но я уверена, что в итоге все будет просто замечательно.

Если некоторые ваши ответы отличаются от наших, это нормально... Главное — чтобы вы задумались и поняли, почему мы дали именно такие ответы.

А мне нравится то, что у тебя получилось с поисковой программой! И раз уж на то пошло, я решил включить в свой ассортимент бас-гитары, банджо и гитары «добро» (такие, со встроенным резонатором)... Да, и как насчет скрипок?

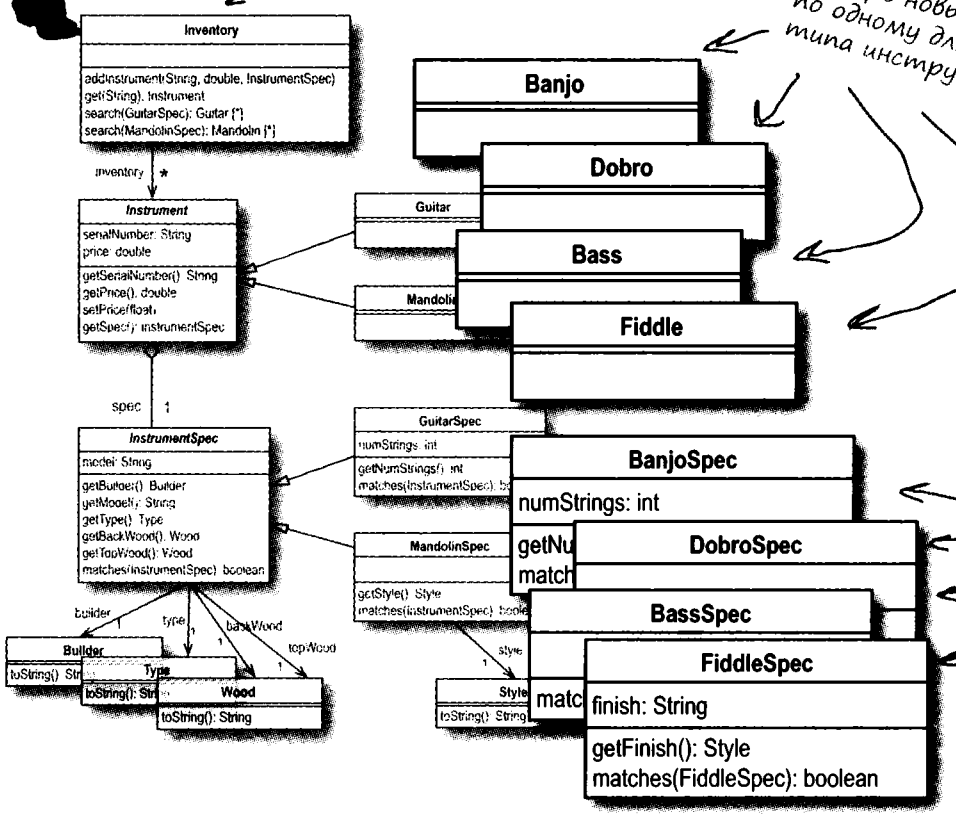
Проверим программу Рика на прочность.

Один из лучших способов проверить, хорошо ли спроектирована программа, — попытаться ИЗМЕНИТЬ ее.

Если изменение программы создает трудности, то, скорее всего, ее структура нуждается в улучшении. Посмотрим, что нужно сделать для добавления пары новых инструментов в приложение Рика:

Нам снова придется изменять Inventory и добавлять поддержку новых типов инструментов. Не самая приятная перспектива...

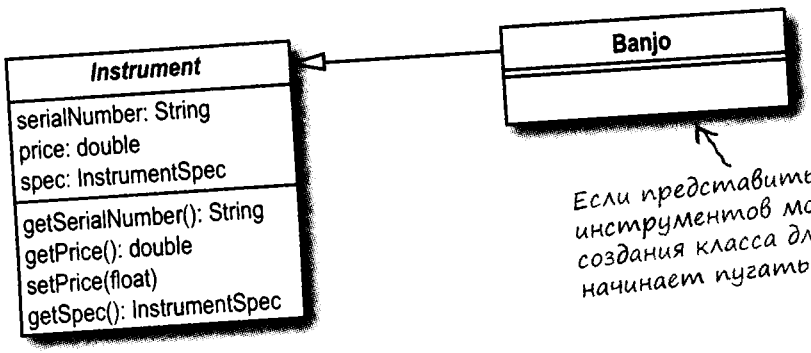
Для четырех новых типов инструментов придется создать четыре новых класса, по одному для каждого типа инструмента.



Также нам потребуются четыре новых объекта *spec, каждый из которых будет добавлять свой набор специфических свойств конкретного инструмента.

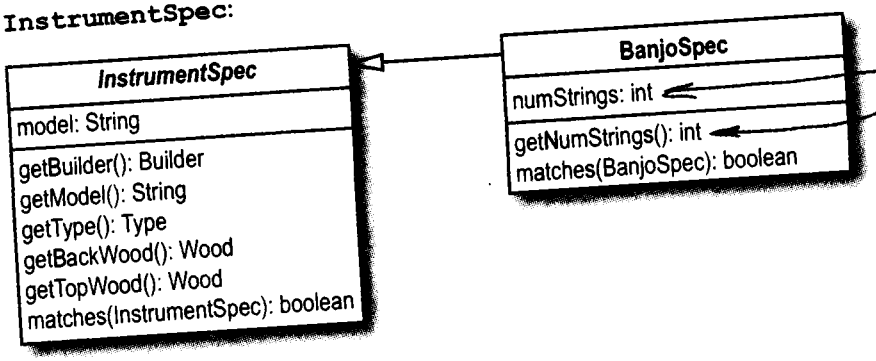
Оказывается, добавлять новые инструменты непросто!

Если определять качество проектирования по простоте изменения продукта, мы столкнулись с серьезными проблемами. Каждый раз, когда в программу добавляется новый инструмент, нам придется создавать очередной субкласс **Instrument**:



Если представить, сколько разновидностей инструментов может продавать Рик, идея создания класса для каждого инструмента начинает пугать.

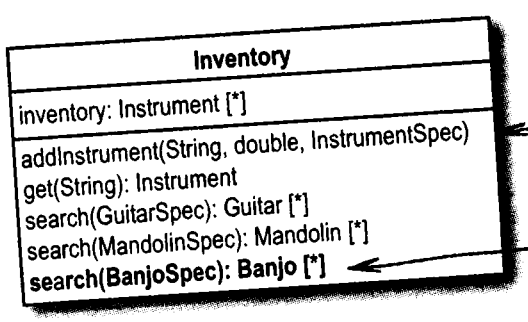
А потом понадобился бы еще и новый субкласс **InstrumentSpec**:



И здесь стало появляться дублирование кода... У банджо, как и у гитар, есть свойство numStrings, но оно имеет недостаточно общий характер для перемещения в суперкласс Instrument.

Но настоящие проблемы начинаются при обновлении методов класса **Inventory** для поддержки нового типа инструментов:

Если вы забыли о проблемах с addInstrument(), вернитесь к странице 240.



Помните все эти instanceof и if/else в addInstrument()? С каждым новым типом инструментов все становится еще хуже.

Ситуация усугубляется. А новая версия с поддержкой банджо нужна прямо сейчас.

И что нам делать?

Похоже, нам еще придется изрядно потрудиться над тем, чтобы превратить приложение Рика в хорошую программу, простую в изменении и расширении. Но это не означает, что проделанная нами работа была лишней... Часто для выявления неочевидных проблем приходится работать над усовершенствованием структуры кода. После применения некоторых ОО-принципов к приложению Рика нам удалось обнаружить недостатки, от которых необходимо избавиться, если мы не хотим потратить несколько лет жизни на написание новых классов для банджо и скрипок (кому захочется этим заниматься?)

Но прежде чем приступать к следующей фазе работы над приложением Рика, необходимо кое-что узнать. Поэтому мы без лишних церемоний отложим приложение Рика в сторону и включим телевизор...

ОО-КАТАСТРОФА!

Любимая телевикторина Объектвиля

ОО-КАТАСТРОФА!

Любимая телевикторина Объектвиля

Исключение риска	Знаменитые проектировщики	Программные конструкции	Сопровождение и повторное использование	Программные неврозы
$100	$100	$100	$100	$100
$200				$200
$30	$300	$300	$300	$300
	$400	$400	$400	$400

Добро пожаловать на ОО-КАТАСТРОФА — любимую телевикторину Объектвиля. Сегодня зрителей ждет множество ответов из области ООП. Надеюсь, вы готовы задавать **правильные** вопросы.

не бывает глупых ответов

О: Может, это и не очевидно, но мы сейчас продолжаем работать над приложением Рика. Чтобы код был гибким и хорошо подходил для повторного использования, нам понадобятся некоторые нетривиальные принципы ООП. Мы хотели дать вам возможность познакомиться с этими принципами до того, как вы начнете применять их для решения непростой задачи.

В: Почему мы развлекаемся? Разве мы не должны исправлять приложение Рика?

О: Найти вопросы, соответствующие ответам, будет непросто, но вы обязательно справитесь. Не торопитесь, очень важно, чтобы вы постарались найти их самостоятельно и только потом переворачивали страницу в поиске подробной информации по каждому вопросу и задействованным принципам ООП. Мы верим, что вы станете настоящим профессионалом.

В: Если это новые принципы, то как я могу определить подходящие вопросы? Вы не слишком многого хотите?

ОО-КАТАСТРОФА!

Любимая телевикторина Объектвиля

Исключение риска	Знаменитые проектировщики	Программные конструкции	Сопровождение и повторное использование	Программные неврозы
$100	$100	$100	$100	$100
$200	$200		$200	$200
$300	$300			
$400	$400			

Эта программная конструкция играет двойную роль: она определяет [illegible] присущее [illegible] тип [illegible] клас[illegible]

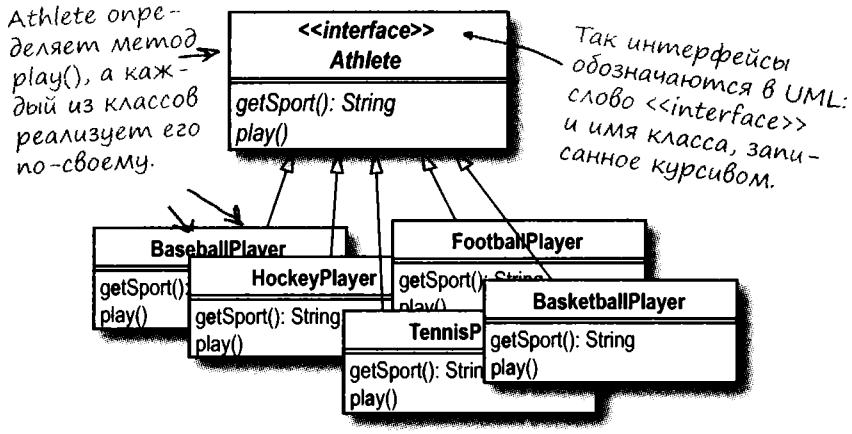
Напишите вопрос, который соответствует приведенному ответу.

«Что такое _____?»

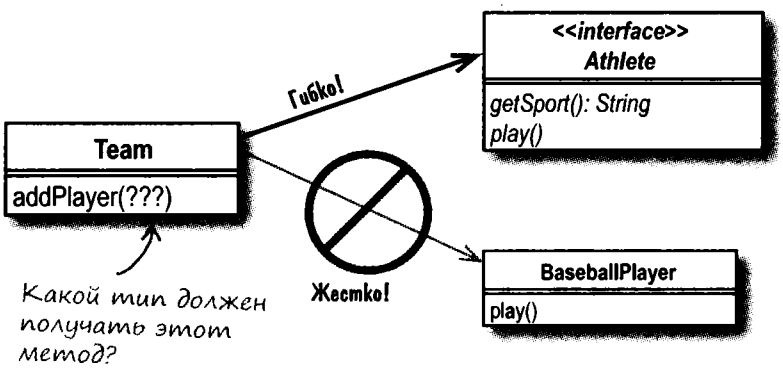
«Что такое ИНТЕРФЕЙС?»

Ну как, угадали? Вы должны были задать этот вопрос для получения ответа на с. 251.

Предположим, у вас имеется приложение, в котором определен интерфейс, и множество субклассов, наследующих общее поведение от этого интерфейса:



Каждый раз, когда вы пишете код, взаимодействующий с этими классами, у вас есть два варианта. Ваш код может взаимодействовать непосредственно с субклассом (например, **FootballPlayer**) или с интерфейсом (**Athlete**). В подобных ситуациях всегда следует программировать для интерфейса, а не для реализации.



Почему это так важно? Потому что приложение становится более гибким. Вместо работы с одним конкретным субклассом (например, **BaseballPlayer**) вы сможете работать с общим интерфейсом **Athlete**. Следовательно, ваш код сможет работать с любым субклассом **Athlete** (скажем, **HockeyPlayer** или **TennisPlayer**) и даже с субклассами, которые еще не были определены (как насчет **CricketPlayer**?).

Программирование для интерфейса (а не для реализации) упрощает расширение ваших программ.

При программировании для интерфейса ваш код будет работать со всеми субклассами этого интерфейса, даже с теми, которые еще не созданы.

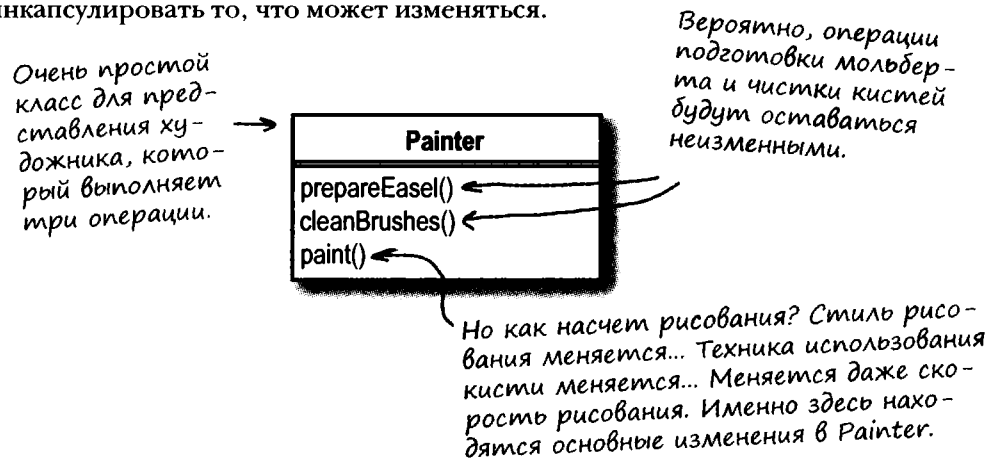
Исключение риска
Знаменитые проектировщики
Программные конструкции
Сопровождение и повторное использование
Программные неврозы
$100
$100
$100
$100
$100
$200
$200
$200
$200
$300
$400
$400

«Что такое_____?»

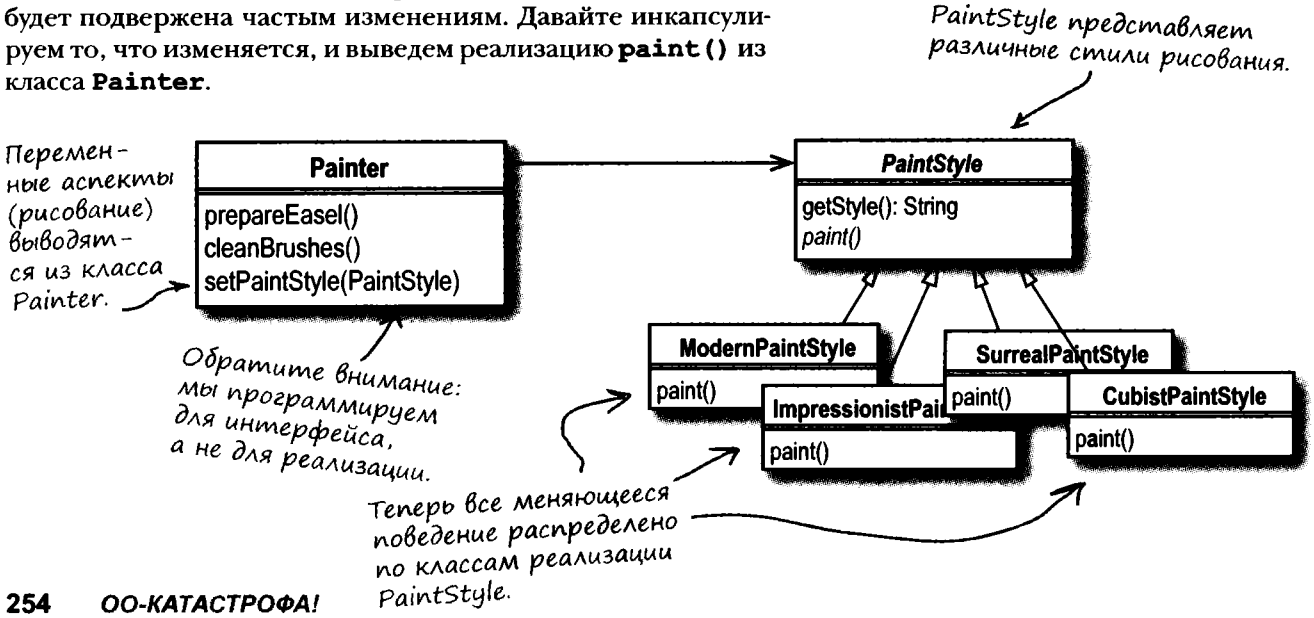
«Что такое ИНКАПСУЛЯЦИЯ?»

Мы уже немало поговорили об инкапсуляции в контексте борьбы с дублированием кода. Однако ее ценность не сводится к предотвращению копирования/вставки кода. Инкапсуляция помогает защитить классы от необязательных изменений.

Любое поведение, которое с большой вероятностью изменится, желательно отделить от частей приложения, которые вряд ли будут часто меняться. Другими словами, старайтесь инкапсулировать то, что может изменяться.



Похоже, класс **Painter** содержит два метода, которые будут оставаться неизменными, но реализация метода **paint()** будет подвержена частым изменениям. Давайте инкапсулируем то, что изменяется, и выведем реализацию **paint()** из класса **Painter**.



Исключение риска
Знаменитые проектировщики
Программные конструкции
Сопровождение и повторное использование
Программные неврозы
$100
$100
$100
$100
$100
$200
$200
$200
$200
$300
$300
$300
$300
$400
Каждый класс должен, насколько возможно, иметь только одну причину для этого.

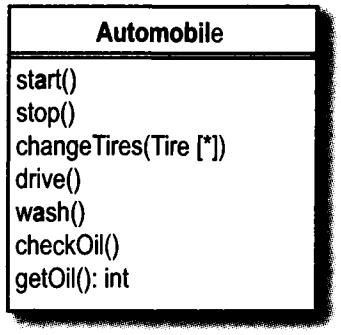
«Что такое _____?»

«Что такое ИЗМЕНЕНИЕ?»

Мы уже знаем, что единственным постоянным фактором в программировании является ИЗМЕНЕНИЕ. Плохо спроектированная программа разваливается при первых признаках изменений, а хорошая программа легко справляется с ними.

Самый простой способ обеспечить устойчивость программы к изменениям — проследить за тем, чтобы у ***каждого класса была только одна причина для изменений.*** Иначе говоря, вы сводите к минимуму вероятность изменения класса, сокращая количество причин, которые могут ***привести*** к его изменению.

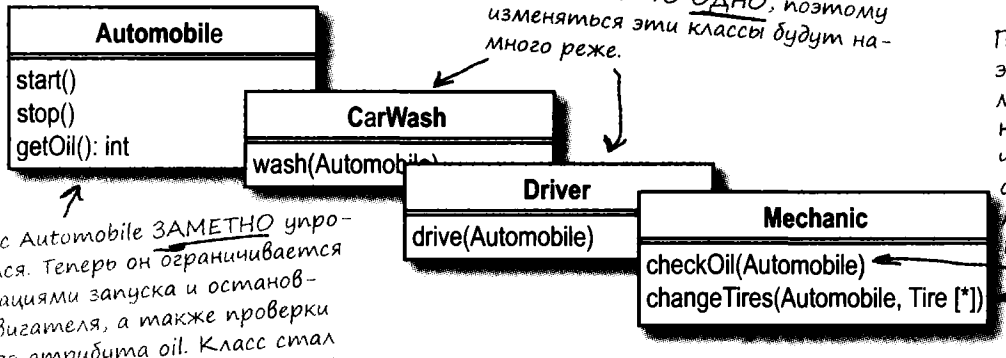
Взгляните на методы этого класса: они выполняют запуск и остановку двигателя, смену шин, управление машиной, мытье и даже проверку и замену масла.



Существует МНОЖЕСТВО причин, которые могут привести к изменению класса. Если механик изменит процедуру проверки масла, или водитель поведет машину более агрессивно, и даже при заезде на другую автомойку — код придется изменять.

Если класс имеет более одной причины для изменения, то скорее всего, он ***пытается делать слишком много всего сразу.*** Попробуйте разбить его функциональность на несколько классов, чтобы ***каждый класс делал что-то одно***, а следовательно, имел только одну причину для изменения.

Каждый из классов Driver и CarWash делает ЧТО-ТО ОДНО, поэтому изменяться эти классы будут намного реже.

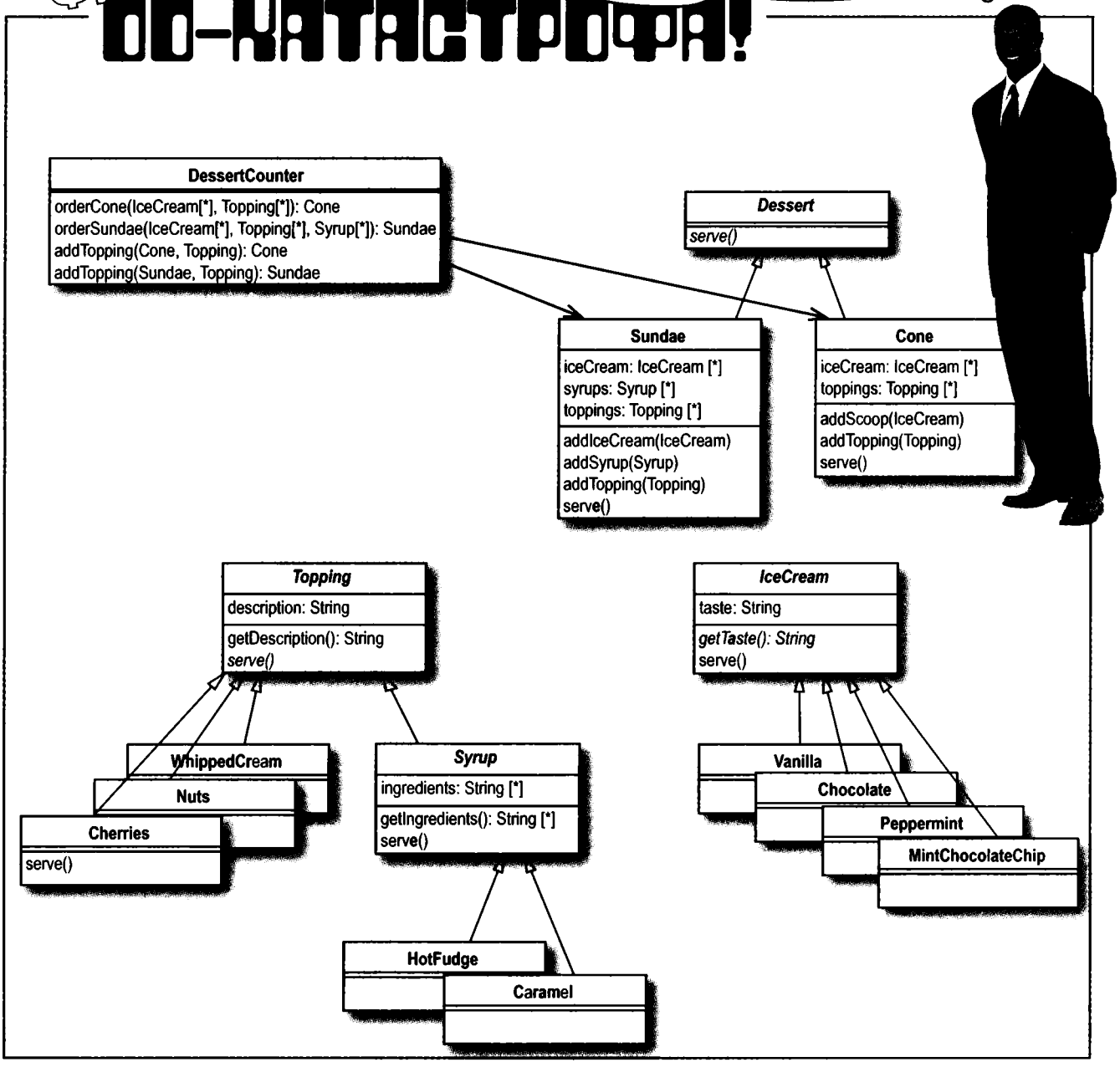


При желании даже эти две функции можно разбить на более мелкие части и разделить функциональность механика на два класса поведения.

Класс Automobile ЗАМЕТНО упростился. Теперь он ограничивается операциями запуска и остановки двигателя, а также проверки своего атрибута oil. Класс стал более устойчивым к изменениям!

Пока вы справляетесь неплохо, но мы приближаемся к ФИНАЛУ. Ниже приведена диаграмма классов приложения, которое трудно назвать гибким. Как следует изменить эту структуру, чтобы избежать ОО-катастрофы? Для этого вам понадобятся все принципы, о которых мы говорили ранее. Не торопитесь! Желаю удачи вам!

Финал ОО-КАТАСТРОФА!



Финал ОО-КАТАСТРОФА!

Ответы

DessertCounter программируется для реализаций интерфейса Dessert. Мы можем сократить два метода order до одного метода orderDessert(), который будет возвращать Dessert.

DessertCounter
- orderCone(IceCream[*], Topping[*]): Cone
- orderSundae(IceCream[*], Topping[*], Syrup[*]): Sundae
- addTopping(Cone, Topping): Cone
- addTopping(Sundae, Topping): Sundae

Dessert
- serve()

Sundae
- iceCream: IceCream [*]
- syrups: Syrup [*]
- toppings: Topping [*]
- addIceCream(IceCream)
- addSyrup(Syrup)
- addTopping(Topping)
- serve()

Cone
- iceCream: IceCream [*]
- toppings: Topping [*]
- addScoop(IceCream)
- addTopping(Topping)
- serve()

У DessertCounter есть не один повод для изменений: это изменение процесса заказа и добавление топпингов к Cone и Sundae. Топпинги должны добавляться непосредственно в классах Dessert, а не здесь.

Syrup — реализация Topping... Метод, добавляющий конкретную разновидность топпинга, не нужен, — это программирование для реализации.

Слишком МНОГО развелось реализаций serve(). Мы должны попытаться инкапсулировать то, что изменяется, и разместить весь код в одном месте. В этом случае при изменении процесса сервировки нам не придется изменять ВСЕ эти классы.

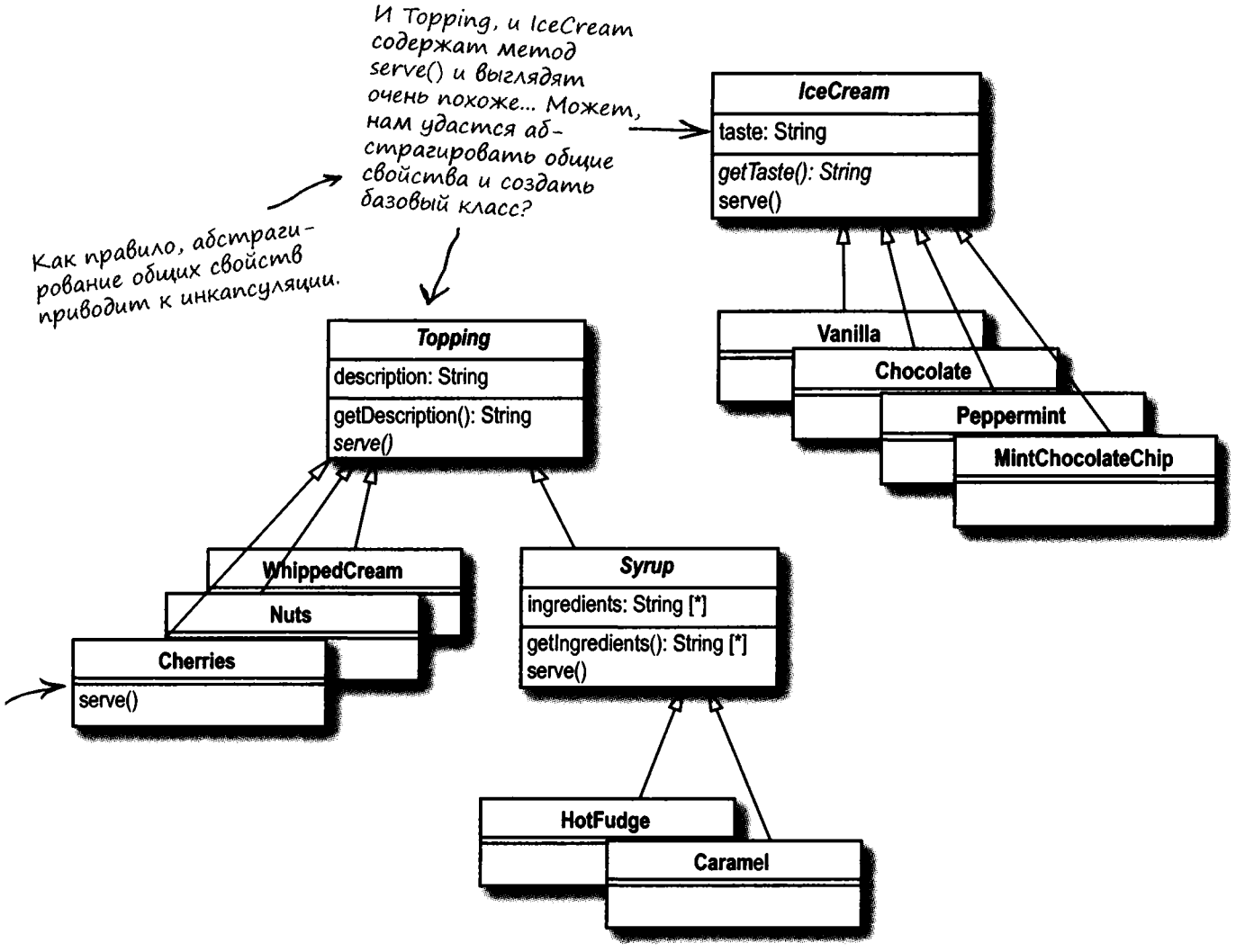
не бывает глупых ответов

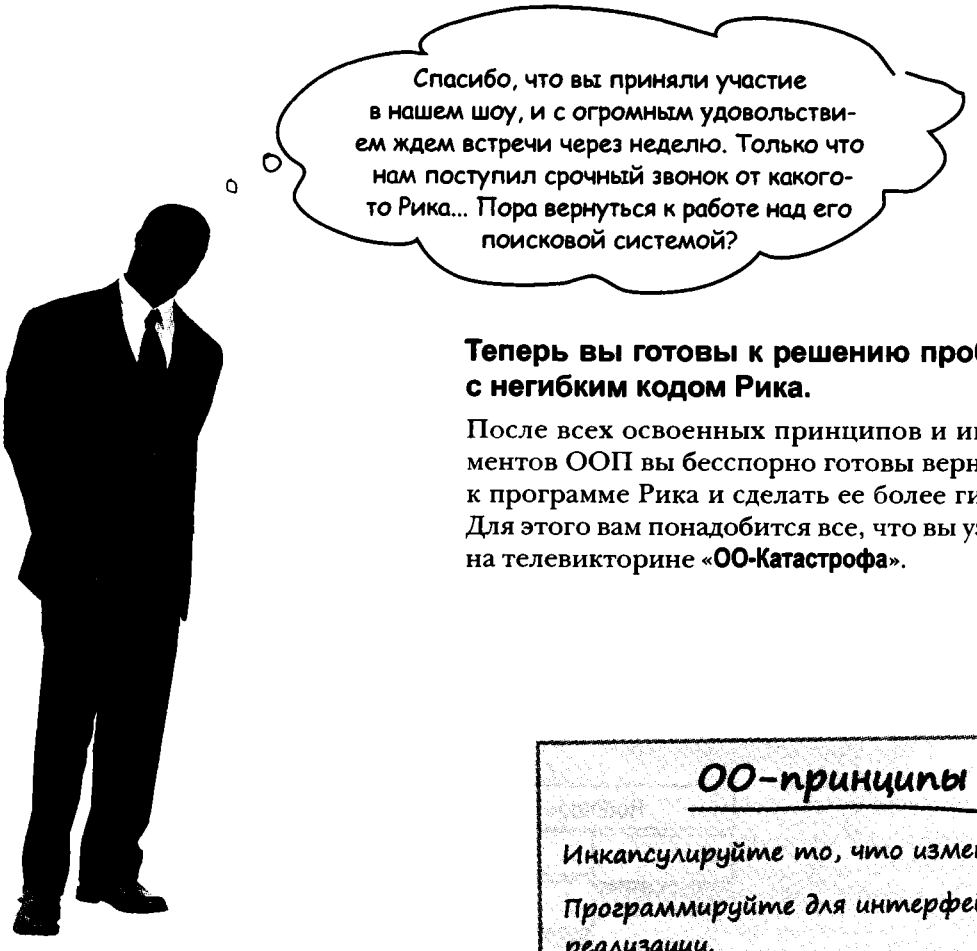
О: Мы уже несколько раз видели, что при дублировании кода следует использовать возможности инкапсуляции. В нашем случае разумно предположить, что сервировка **Sundae** не должна сильно отличаться от сервировки **Cone**.

Вы можете создать новый класс **DessertService** и поместить метод **serve()** в этот класс. В дальнейшем классы **Dessert**, **IceCream** и **Topping** будут обращаться к **DessertService.serve()**. Если метод **serve()** изменится, переписывать код придется только в одном месте: **DessertService**.

Таким образом, мы инкапсулировали все то, что может измениться (код метода **serve()**), и позаботились о том, чтобы у каждого класса была только единственная причина для изменений. Двойная выгода!

В: Как вы догадались, что методы serve() нужно инкапсулировать из разных классов?


И Topping, и IceCream содержат метод serve() и выглядят очень похоже... Может, нам удастся абстрагировать общие свойства и создать базовый класс?
Как правило, абстрагирование общих свойств приводит к инкапсуляции.
IceCream
taste: String
getTaste(): String
serve()
Topping
description: String
getDescription(): String
serve()
Vanilla
Chocolate
Peppermint
MintChocolateChip
WhippedCream
Nuts
Cherries
serve()
Syrup
ingredients: String [*]
getIngredients(): String [*]
serve()
HotFudge
Caramel



Теперь вы готовы к решению проблем с негибким кодом Рика.

После всех освоенных принципов и инструментов ООП вы бесспорно готовы вернуться к программе Рика и сделать ее более гибкой. Для этого вам понадобится все, что вы узнали на телевикторине «**ОО-Катастрофа**».

ОО-принципы

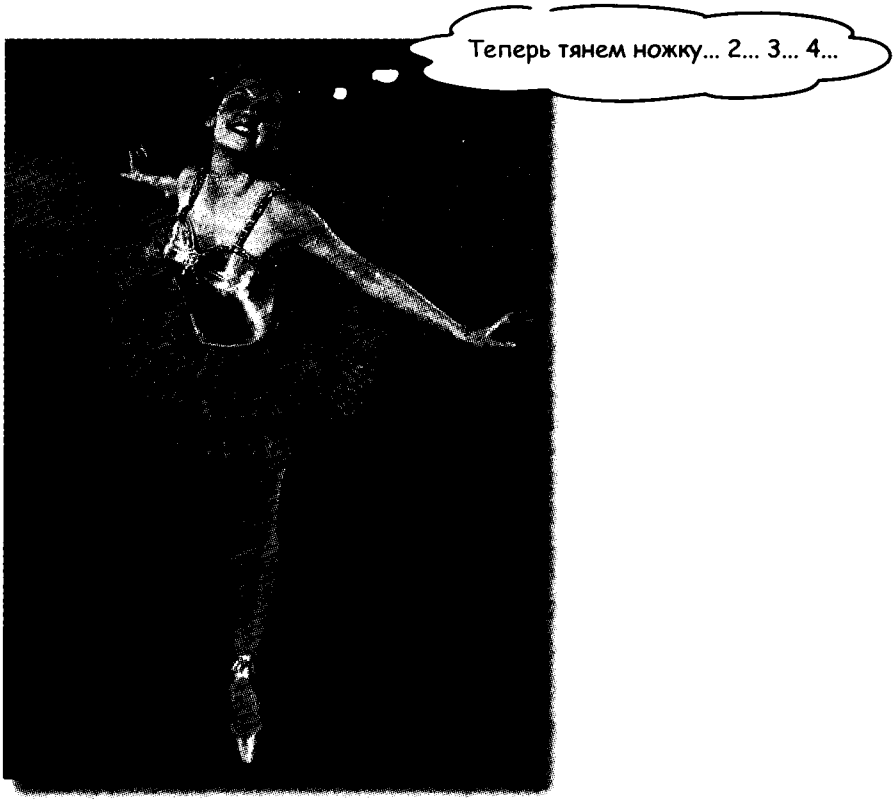
Инкапсулируйте то, что изменяется.

Программируйте для интерфейса, а не для реализации.

Каждый класс в приложении должен иметь только одну причину для изменения.

Эти три принципа ЧРЕЗВЫЧАЙНО ВАЖНЫ! Запомните их. Мы еще неоднократно будем возвращаться к ним в будущих главах.

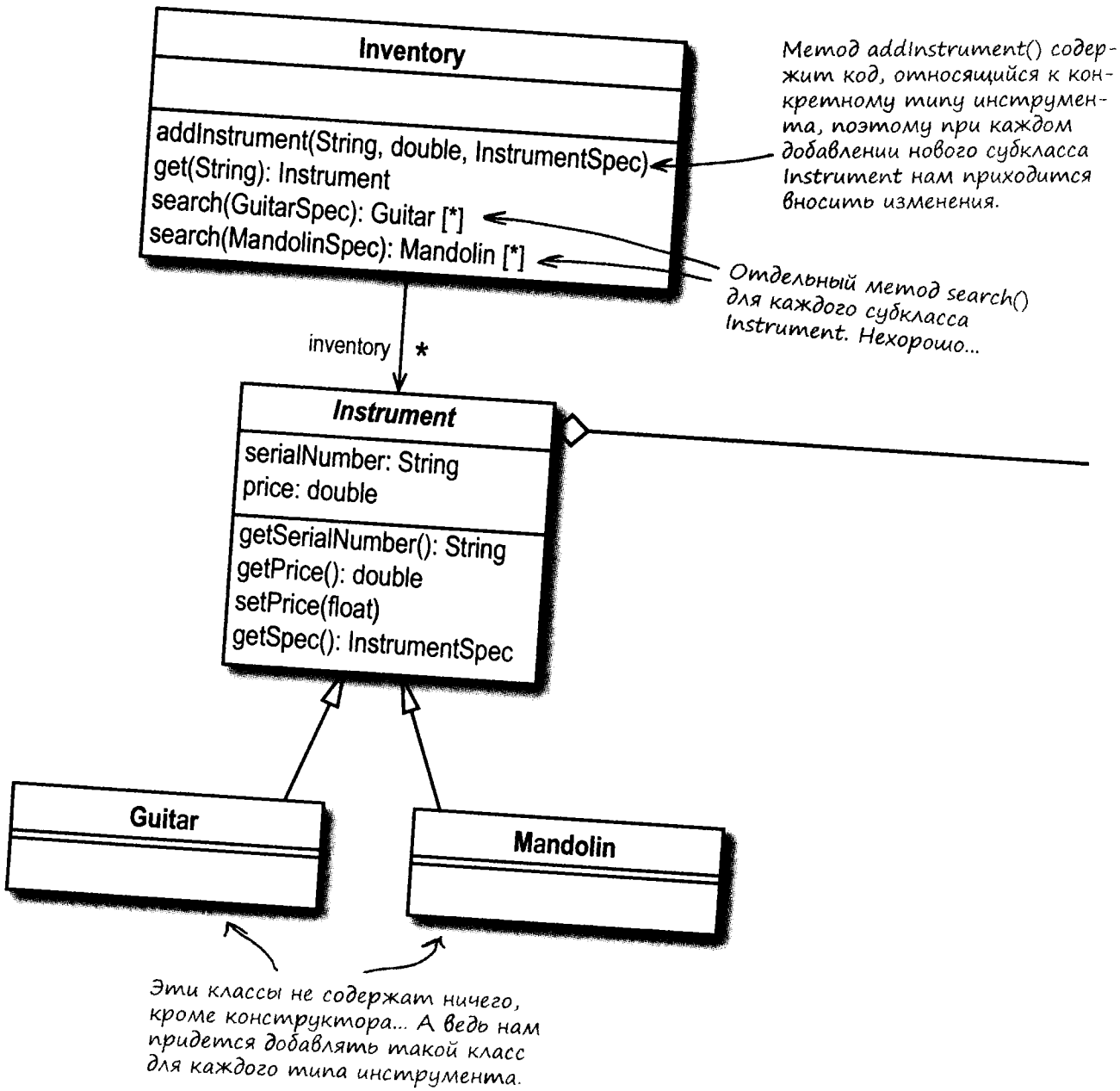
Зарядка для программ

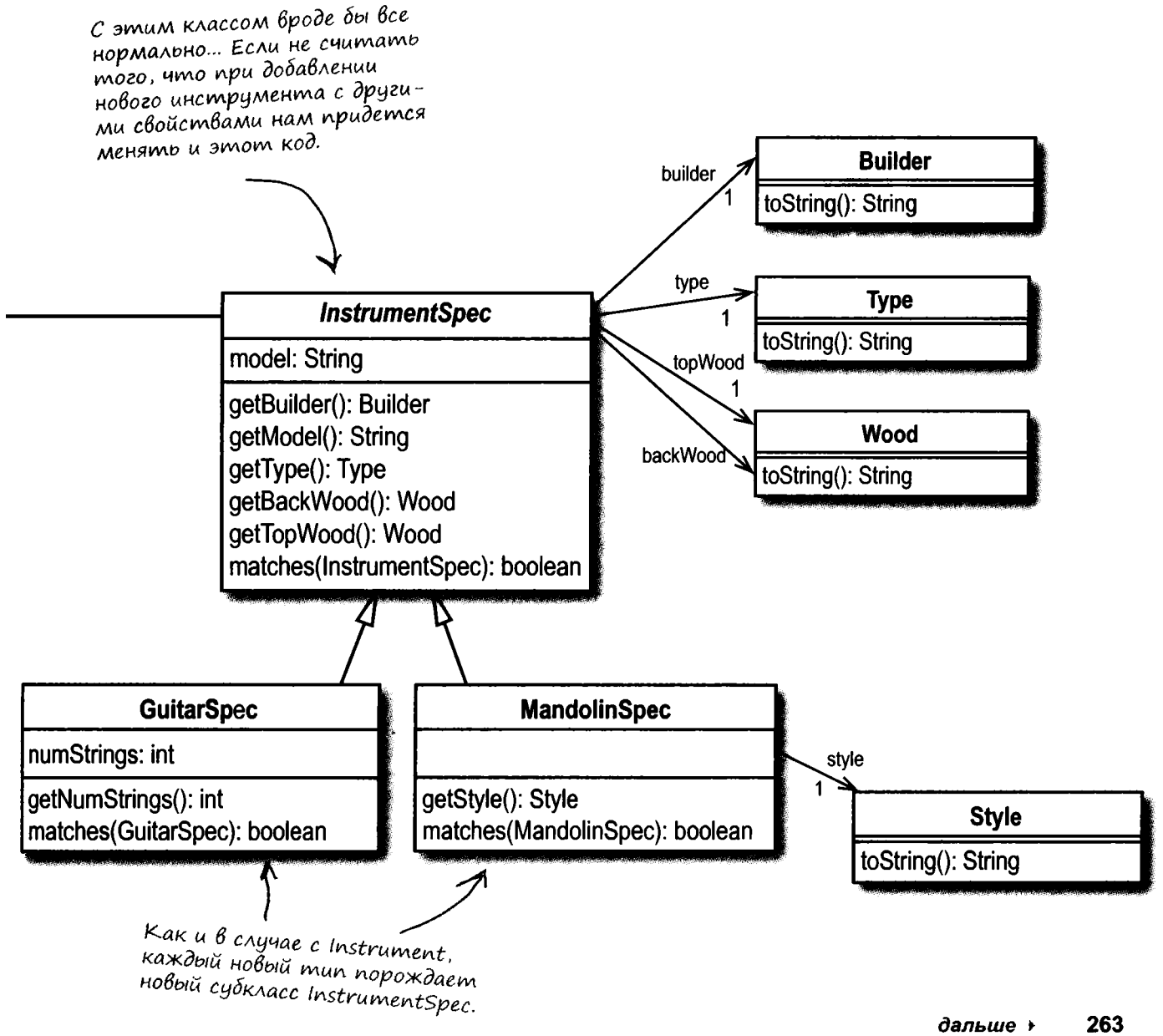


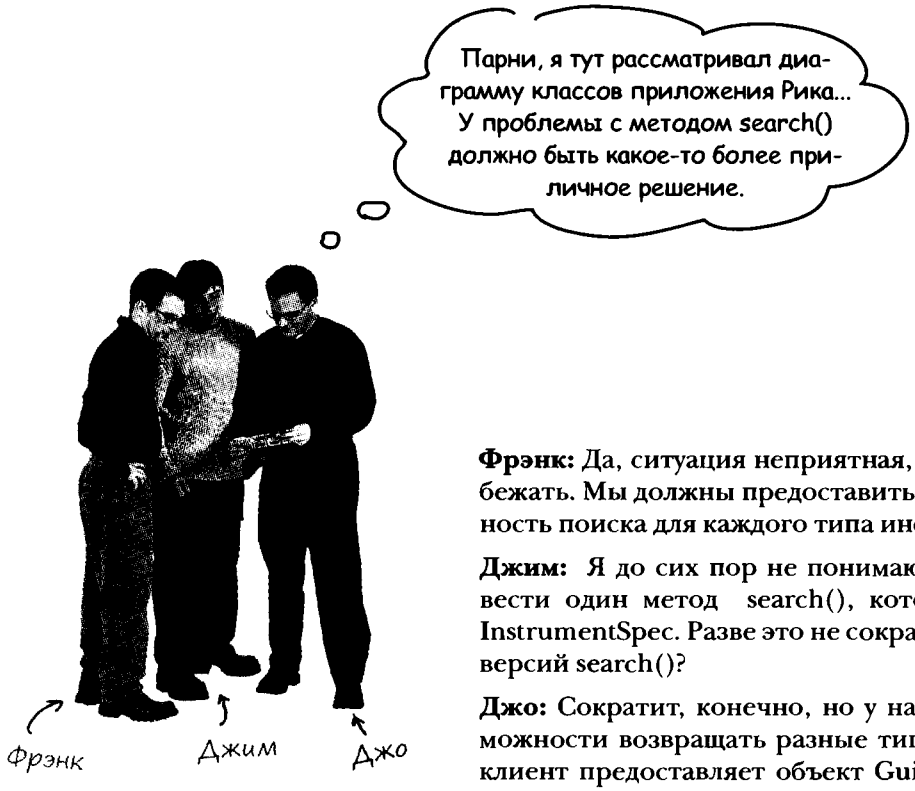
А вам когда-нибудь хотелось обрести большую гибкость? Если внесение изменений в приложение создает проблемы, скорее всего, это означает, что вашей программе не хватает гибкости и пластичности. Чтобы помочь приложению прийти в хорошую форму, необходимо провести анализ, основательно проработать структуру кода приложения и узнать, как ОО-принципы способствуют ослаблению связности. А в завершение вы увидите, как высокое сцепление способствует ослаблению связности. Звучит заманчиво? Переверните страницу, и мы продолжим наводить порядок в нашем негибком приложении.

Возвращаемся к приложению Рика

Вооружившись новыми принципами ООАП, мы готовы к тому, чтобы сделать приложение Рика гибким и хорошо спроектированным. Стоит напомнить, на чем мы остановились (и какие проблемы обнаружили):




С этим классом вроде бы все нормально... Если не считать того, что при добавлении нового инструмента с другими свойствами нам придется менять и этот код.
InstrumentSpec
model: String
getBuilder(): Builder
getModel(): String
getType(): Type
getBackWood(): Wood
getTopWood(): Wood
matches(InstrumentSpec): boolean
builder
1
Builder
toString(): String
type
1
Type
toString(): String
topWood
1
backWood
Wood
toString(): String
GuitarSpec
numStrings: int
getNumStrings(): int
matches(GuitarSpec): boolean
MandolinSpec
getStyle(): Style
matches(MandolinSpec): boolean
style
1
Style
toString(): String
Как и в случае с Instrument, каждый новый тип порождает новый субкласс InstrumentSpec.



Фрэнк: Да, ситуация неприятная, но я не знаю, как ее избежать. Мы должны предоставить клиентам Рика возможность поиска для каждого типа инструментов.

Джим: Я до сих пор не понимаю, почему бы нам не завести один метод search(), который получает объект InstrumentSpec. Разве это не сократит количество разных версий search()?

Джо: Сократит, конечно, но у нас по-прежнему нет возможности возвращать разные типы инструментов. Если клиент предоставляет объект GuitarSpec, он никогда не совпадет с BanjoSpec или MandolinSpec. Таким образом, список, возвращаемый search(), всегда будет содержать только один тип инструментов, к которому относится спецификация клиента.

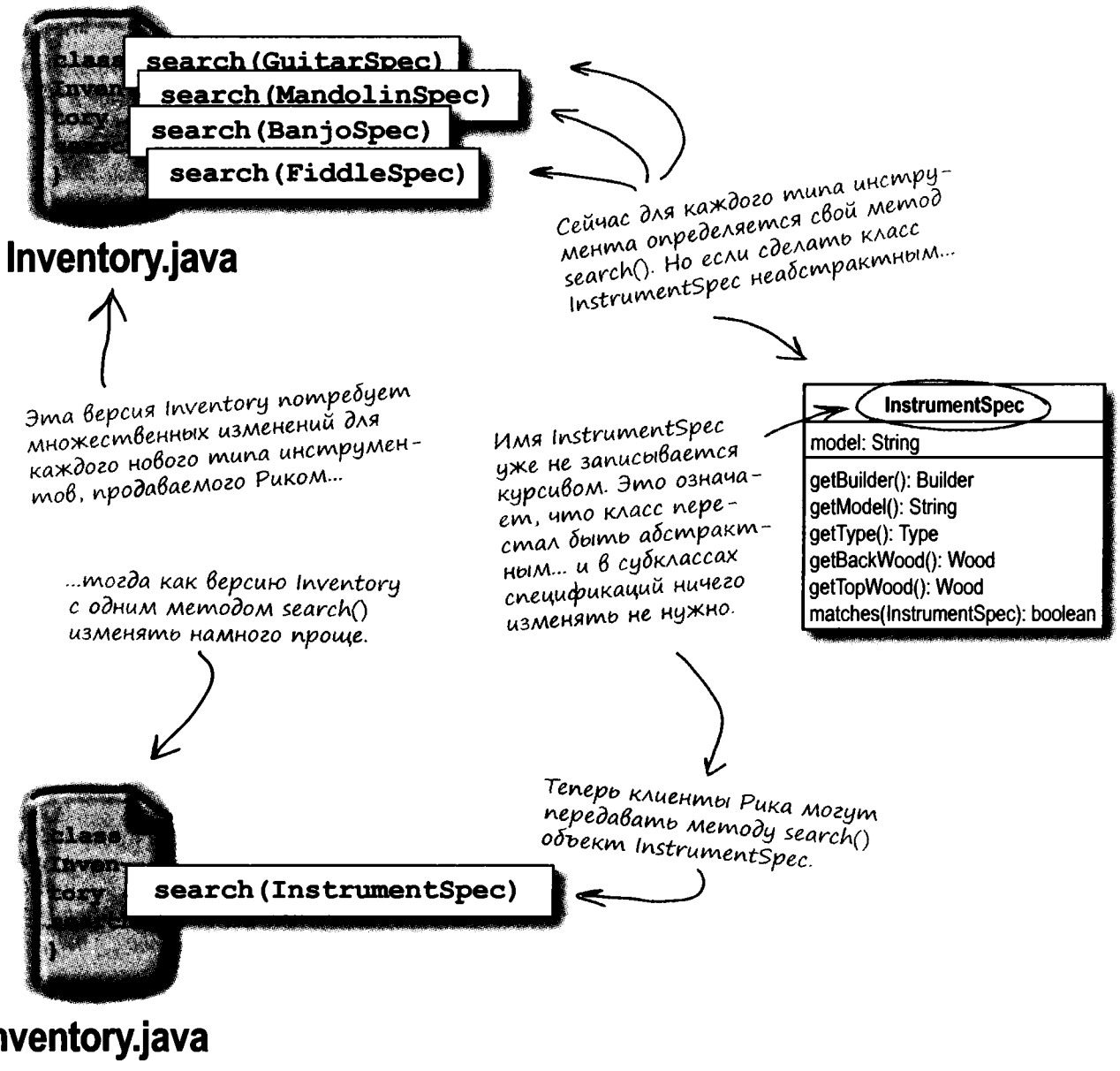
Джим: Потому что мы не можем создать экземпляр InstrumentSpec, верно? Это абстрактный класс, поэтому нам приходится создавать MandolinSpec, или BanjoSpec, или еще что-нибудь.

Фрэнк: Возможно, в этом и кроется проблема... Кстати, а разве мы не должны программировать для интерфейса InstrumentSpec вместо программирования для реализации вроде GuitarSpec или BanjoSpec?

Джо: Хмм... Я об этом еще не думал, но ты прав; мы должны сосредоточиться на интерфейсе, а не на разных классах реализации.

Присмотримся повнимательнее к методу search()

Достаточно очевидно, что наш способ организации поиска для клиентов Рика имеет недостатки. Класс **InstrumentSpec** можно сделать конкретным, но решит ли это все наши проблемы?



Результаты анализа

Давайте посмотрим, улучшит ли структуру кода **Inventory** преобразование **InstrumentSpec** в конкретный класс.

Inventory
inventory: Instrument [*]
addInstrument(String, double, InstrumentSpec) get(String): Instrument **search(InstrumentSpec): Instrument [*]**

Inventory.java

Принципиальное изменение, описанное на этой странице.

```
public class Inventory {

  private List inventory;

  public Inventory() {
    inventory = new LinkedList();
  }

  public void addInstrument(String serialNumber, double price,
                            InstrumentSpec spec) {
    Instrument instrument = null;
    if (spec instanceof GuitarSpec) {
      instrument = new Guitar(serialNumber, price, (GuitarSpec)spec);
    } else if (spec instanceof MandolinSpec) {
      instrument = new Mandolin(serialNumber, price, (MandolinSpec)spec);
    }
    inventory.add(instrument);
  }

  public Instrument get(String serialNumber) {
    for (Iterator i = inventory.iterator(); i.hasNext(); ) {
      Instrument instrument = (Instrument)i.next();
      if (instrument.getSerialNumber().equals(serialNumber)) {
        return instrument;
      }
    }
    return null;
  }

  public List search(InstrumentSpec searchSpec) {
    List matchingInstruments = new LinkedList();
    for (Iterator i = inventory.iterator(); i.hasNext(); ) {
      Instrument instrument = (Instrument)i.next();
      if (instrument.getSpec().matches(searchSpec))
        matchingInstruments.add(instrument);
    }
    return matchingInstruments;
  }
}
```

Здесь по-прежнему остаются проблемы... Метод увеличивается и усложняется каждый раз, когда в приложение добавляется новый тип инструмента...

...и мы программируем для классов реализации, а не для базового класса Instrument.

Метод search() выглядит намного лучше! Всего одна версия, которая теперь получает объект InstrumentSpec.

В новом варианте мы программируем для базового типа Instrument, а не для классов реализации (таких, как Guitar или Mandolin) — налицо значительное улучшение структуры.

Кроме улучшения структуры кода, метод search() теперь возвращает все подходящие инструменты, даже если в списке инструменты разных типов (скажем, две гитары и одна мандолина).

Возьми в руку карандаш

Инструменты не похожи друг на друга...

или все-таки похожи?

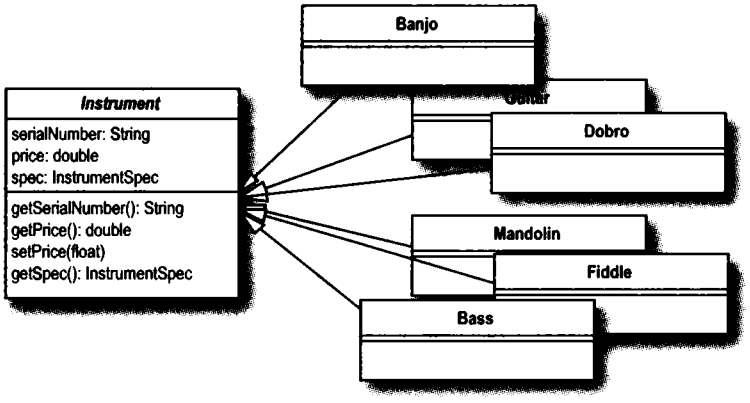
Метод search() — не единственное, что усложняет добавление новых инструментов в приложение Рика. Для каждого нового типа инструментов нам необходим новый субкласс Instrument. Но почему? Давайте проанализируем происходящее чуть подробнее.

Почему в приложении Рика необходим класс Instrument?

Что общего есть у всех инструментов?

Чем отличаются разные инструменты?

Если у вас есть мысли по поводу того, как изменить приложение Рика, чтобы избавиться от субклассов конкретных инструментов, пометьте эти изменения на диаграмме классов внизу. Добавляйте и удаляйте классы и свойства так, как считаете нужным.



Возьми в руку карандаш
Решение

Инструменты не похожи друг на друга...

или все-таки похожи?

Метод search() — не единственное, что усложняет добавление новых инструментов в приложение Рика. Для каждого нового типа инструментов нам необходим новый субкласс Instrument. Но почему? Давайте проанализируем происходящее чуть подробнее.

Почему в приложении Рика необходим класс Instrument?

Многие инструменты имеют по крайней мере несколько общих свойств (таких, как серийный номер или цена). Класс Instrument содержит общие свойства, а типы конкретных инструментов расширяют Instrument.

Вероятно, ваши ответы не совпадут с нашими, но по крайней мере вы должны мыслить в том же направлении.

Что общее есть у всех инструментов?

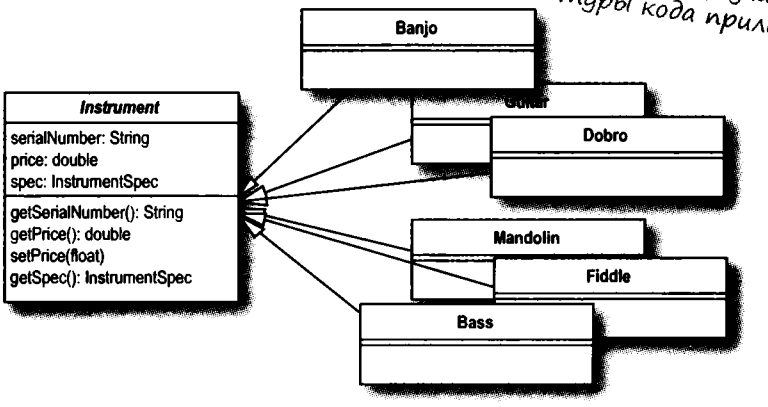
Серийный номер, цена и некоторые спецификации (хотя подробности этих спецификаций могут быть разными для разных типов инструментов).

Чем отличаются разные инструменты?

Спецификациями: разные типы инструментов могут содержать разные наборы свойств. А поскольку каждый инструмент использует собственный объект InstrumentSpec, разные инструменты имеют разные конструкторы.

Если у вас есть мысли по поводу того, как изменить приложение Рика, чтобы избавиться от субклассов конкретных инструментов, пометьте эти изменения на диаграмме классов внизу. Добавляйте и удаляйте классы и свойства так, как считаете нужным.

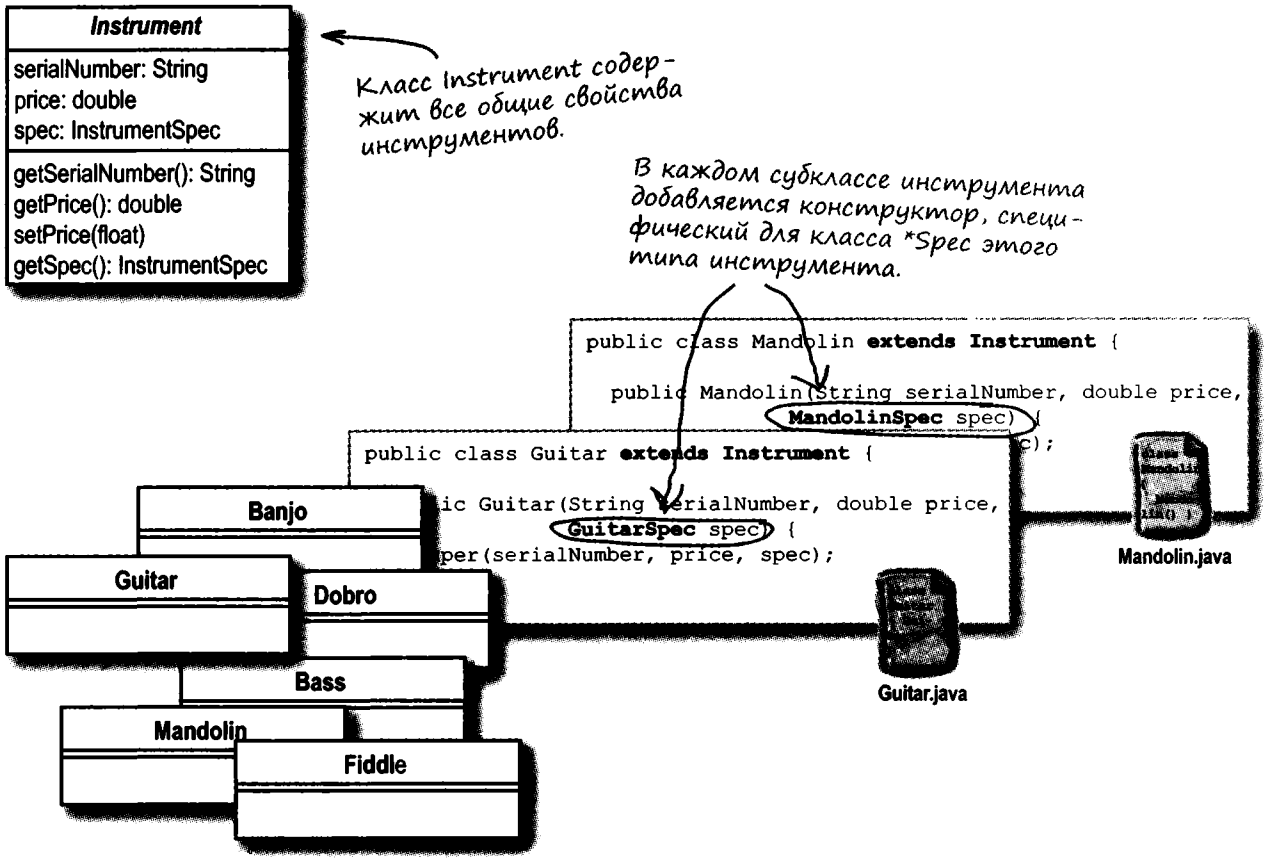
А вы предложили какие-нибудь варианты улучшения структуры кода приложения Рика?



Проблемы с классами инструментов

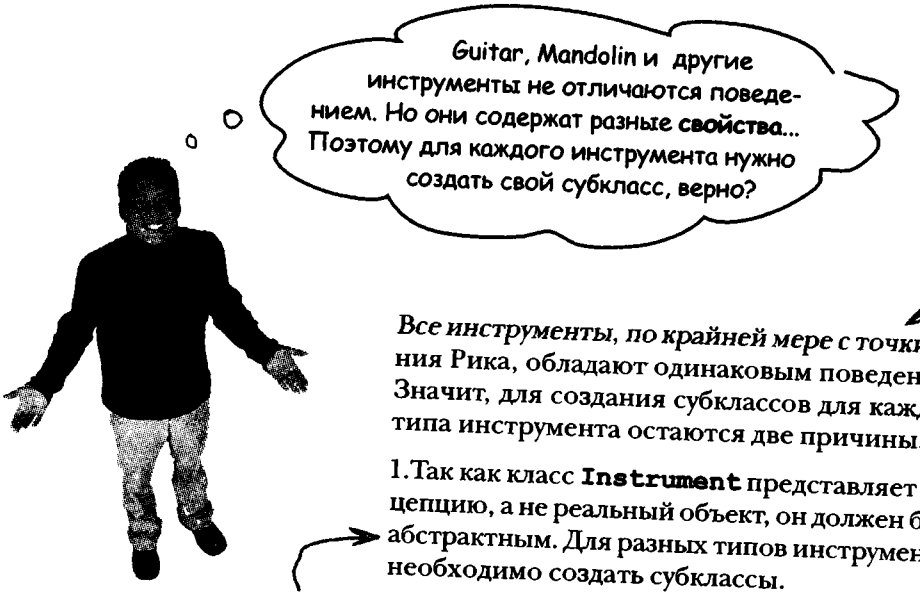
И хотя метод **`search()`** выглядит лучше, остались серьезные проблемы с субклассами инструментов и методом **`addInstrument()`** в **`Inventory`**.

Вспомните, что класс **`Instrument`** изначально был сделан абстрактным, потому что для каждого типа инструмента создается свой субкласс:



Но ведь классы создаются ради поведения!

Субклассы обычно создаются из-за того, что субкласс своим **поведением** отличается от суперкласса. В приложении Рика поведение **`Guitar`** отличается от поведения **`Instrument`**? Класс **`Guitar`** *функционирует* в приложении не так, как класс **`Mandolin`** или **`Banjo`**?



Если бы мы писали систему, в которой была задействована игра на этих инструментах, нам понадобились бы субклассы для разных приемов игры, таких как pluck(), strum() или frail().

Все инструменты, по крайней мере с точки зрения Рика, обладают одинаковым поведением. Значит, для создания субклассов для каждого типа инструмента остаются две причины:

1. Так как класс **`Instrument`** представляет концепцию, а не реальный объект, он должен быть абстрактным. Для разных типов инструментов необходимо создать субклассы.

Это полезный принцип, но со всеми этими субклассами он создает нам проблемы. Вскоре мы вернемся к нему.

2. Разные типы инструментов обладают разными свойствами и используют разные субклассы **`InstrumentSpec`**, поэтому для каждого типа инструмента нужен свой конструктор.

Еще один пример программирования для реализации вместо интерфейса. Следовательно, это не может быть веской причиной для сохранения абстрактности Instrument.

Причины выглядят достаточно убедительно (по крайней мере первая), но у нас появляется множество лишних классов, которые почти ничего не делают... А от этого программа становится негибкой и ее трудно модифицировать. Что же делать?

Вспомните второй шаг на пути создания хороших программ из главы 1:

Так как приложение Рика уже делает то, что должно (шаг 1), можно постараться улучшить гибкость его программы.

Применяйте базовые ОО-принципы для повышения гибкости.

Как применить этот шаг к проблемам, обнаруженным нами в приложении Рика?

Возьми в руку карандаш

На помощь приходят объектно-ориентированные принципы!

В приложении Рика определенно существует проблема, но мы пока не знаем, что с ней делать. А когда вы не знаете, что делать с проблемой проектирования, просто переберите известные вам ОО-принципы и посмотрите, не помогут ли они улучшить структуру кода программы.

Поставьте «галочку» рядом с каждым из перечисленных ниже принципов, если он, по вашему мнению, может помочь в нашей ситуации. Затем запишите, как использовать выбранные принципы для улучшения структуры поисковой программы Рика.

- ☐ Наследование

- ☐ Полиморфизм

- ☐ Абстракция

- ☐ Инкапсуляция

→ Сравните с нашими ответами на следующей странице.

Возьми в руку карандаш
Решение

На помощь приходят объектно-ориентированные принципы!

В приложении Рика определенно существует проблема, но мы пока не знаем, что с ней делать. А когда вы не знаете, что делать с проблемой проектирования, просто переберите известные вам ОО-принципы и посмотрите, не помогут ли они улучшить структуру кода программы.

☑ **Наследование**

Мы уже используем наследование с классами Instrument и InstrumentSpec и их субклассами. Но, похоже, субклассы конкретных инструментов не делают практически ничего, кроме наследования от Instrument... Если не считать слегка различающихся конструкторов.

☑ **Полиморфизм**

В методе search() благодаря полиморфизму все инструменты обрабатываются как экземпляры Instrument (вместо того, чтобы разбираться, являются ли они экземплярами Guitar, Mandolin и т.д.). Это значительно упрощает поиск... Но было бы неплохо также использовать полиморфизм и в addInstrument() для устранения части повторяющегося кода.

☑ **Абстракция**

Класс InstrumentSpec абстрагирует подробности спецификаций инструментов из класса Instrument, чтобы добавление новых свойств инструментов не влияло на базовый класс Instrument.

☑ **Инкапсуляция**

В нашей программе широко применяется инкапсуляция, но, может быть, ее следует применять еще шире... помните: то, что изменяется, следует инкапсулировать! Так как в нашем случае изменяются свойства каждого типа инструмента, нельзя ли каким-то образом полностью инкапсулировать их из Instrument и InstrumentSpec?

Парни, в структуре нашего приложения применяются наследование, полиморфизм и абстракция. Но я начинаю думать, что ключевое значение имеет инкапсуляция. Помните, что говорилось ранее об отделении переменных аспектов от постоянных?

Джо: Да, конечно. Ведь ты говоришь об инкапсуляции того, что изменяется?

Фрэнк: Точно! А мы знаем, что в нашем приложении изменяется набор свойств каждого инструмента.

Джим: Я думал, что мы уже прошли этот этап; именно поэтому у нас в приложении появились все эти субклассы Instrument, такие как Guitar и Mandolin. И они помогают нам представлять различия между инструментами.

Фрэнк: Но это нам не особенно помогло... К тому же поведение инструментов не изменяется, так действительно ли нам нужны субклассы для всех инструментов?

Джо: Итак, ты предлагаешь сделать Instrument конкретным классом вместо абстрактного? Тогда мы сможем избавиться от субклассов конкретных инструментов.

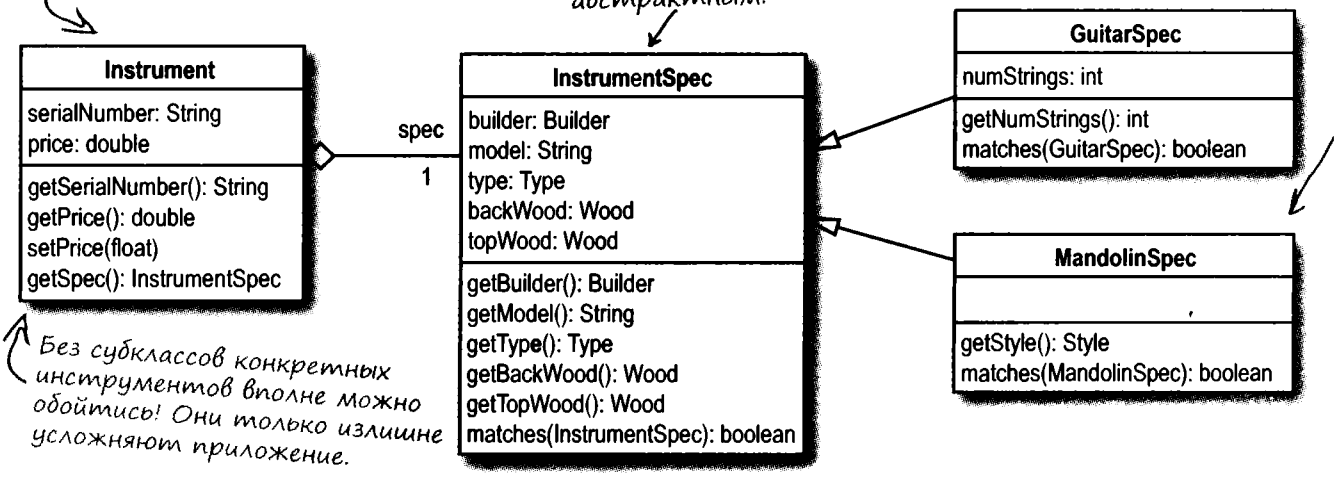
Джим: Но... Я совсем запутался. А как же свойства, набор которых зависит от инструмента?

Фрэнк: А что свойства? Класс Instrument содержит ссылку на InstrumentSpec, и все различия в свойствах могут обрабатываться этими классами. Посмотри:

Вообще говоря, мы уже инкапсулировали свойства из остальных частей приложения! Просто мы еще не пользовались результатами правильного решения.

Имя Instrument уже не курсивное; класс стал конкретным.

И класс InstrumentSpec тоже перестал быть абстрактным.

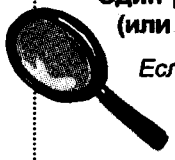


Без субклассов конкретных инструментов вполне можно обойтись! Они только излишне усложняют приложение.

Смерть проектировочных решений

Одна из самых сложных задач — расставание с ошибками *в ваших собственных решениях.* В поисковой программе Рика идея создания отдельного субкласса `Instrument` для каждого типа инструментов оказалась неудачной. Но чтобы понять это, нам понадобилось почти 30 страниц (и две части главы 5). Почему?

Потому что решение выглядело вполне разумно, а отказаться от чего-то, что вроде бы работает, всегда ТРУДНО!

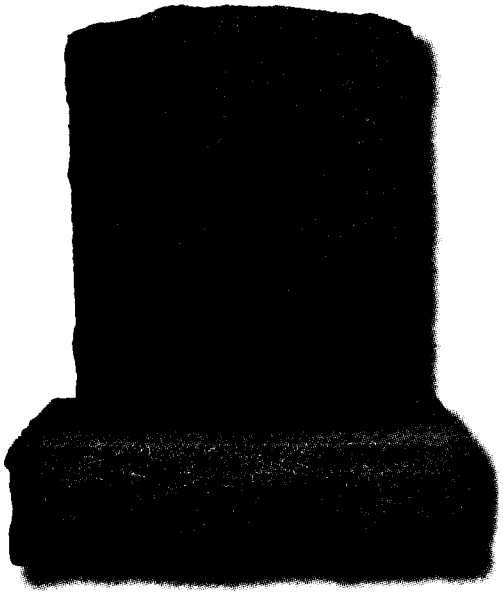


Один раз запрограммируй, два раза проверь (или больше!)

Если возникают проблемы, продолжайте критически анализировать свои программные структуры. Возможно, виновато принятое ранее решение.

Кромсать чужой код легко, но вы должны научиться анализировать свой код. В таких ситуациях нередко помогает критический анализ вашего кода коллегами-программистами. Не огорчайтесь, если придется вносить изменения, — хорошо спроектированная структура приложения сэкономит много времени в долгосрочной перспективе.

Проектирование — итеративный процесс... И вы должны быть готовы изменять собственные решения точно так же, как и решения, унаследованные от других программистов.



Давайте раз и навсегда похороним свое неудачное решение с созданием субклассов конкретных инструментов и продолжим создание хорошей программы.

Самомнение убивает хорошие решения

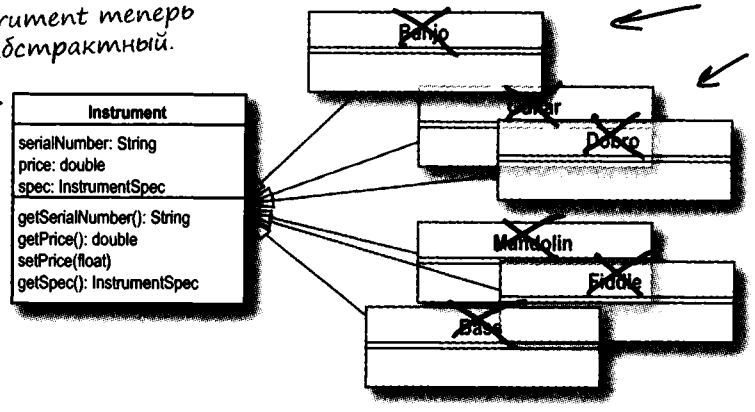
Никогда не бойтесь критически анализировать свои решения и совершенствовать их, даже если для этого вам придется отступить на шаг назад.

Преобразуем плохие структурные решения в хорошие

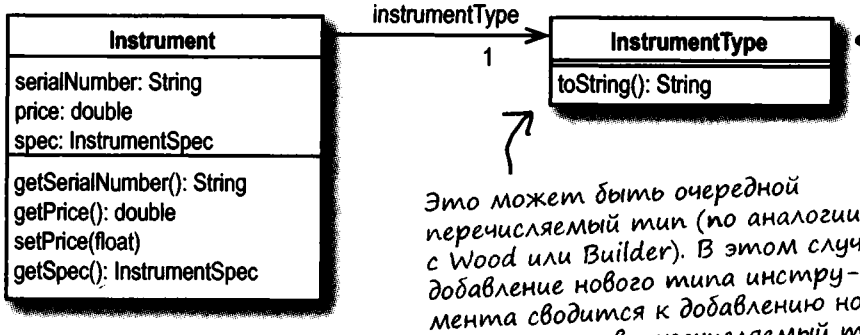
Начнем с истребления всех субклассов конкретных инструментов:

Instrument теперь не абстрактный.

Мы просто выкидываем из структуры приложения шесть классов. Теперь для новых музыкальных инструментов из магазина Рика новые классы уже не понадобятся!



Вероятно, в каждый инструмент придется включить новое свойство, которое определяет тип инструмента:



Здесь хранятся такие значения, как GUITAR, BANJO, MANDOLIN и т. п., — гораздо лучше набора субклассов.

Это может быть очередной перечисляемый тип (по аналогии с Wood или Builder). В этом случае добавление нового типа инструмента сводится к добавлению нового значения в перечисляемый тип.

Еще один разговор коллег (с участием Джилл)

Я не хочу вмешиваться, но я долго думала на эту тему, Джо: **Инкапсулируй эти переменные.**

Джилл прислушивалась к нашим объяснениям в этой главе, и у нее появились мысли о том, как улучшить приложение Рика.

Джо: Но ведь мы все сделали... Мы сделали класс Instrument конкретным и избавились от всех специализированных субклассов инструментов.

Джилл: Вообще-то я думаю, что это только первый шаг. Что на самом деле изменяется в приложении Рика?

Что изменяется в приложении Рика?

_____

_____

_____

Запишите здесь то, что изменяется в приложении Рика.

Фрэнк: Мы это уже проходили: изменяются свойства каждого инструмента.

Джилл: Можно ли их как-то инкапсулировать?

Джо: Уже инкапсулировали: мы использовали для этого класс InstrumentSpec.

Фрэнк: Погоди минуту, Джо. Мы использовали InstrumentSpec, потому что эти свойства использовались и клиентами, и инструментами. И это была борьба с дублированием кода...

Джилл: Да! Я как раз об этом... Свойства в InstrumentSpec тоже изменяются. Так что, возможно, нам понадобится еще один уровень инкапсуляции.

Джо: Раз свойства инструментов изменяются, мы должны извлечь их из InstrumentSpec? Получается двойная инкапсуляция или что-то в этом роде.

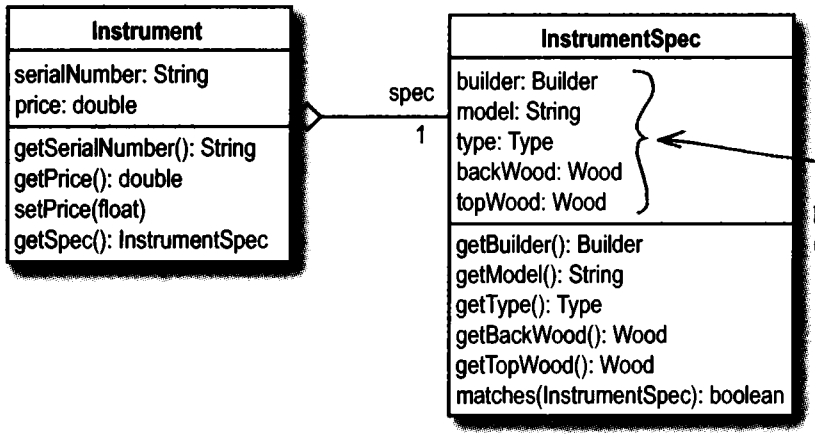
Джилл: Вроде того... Сначала мы инкапсулируем из класса Instrument спецификации, общие для клиентских запросов и инструментов, а потом инкапсулируем изменяющиеся свойства из класса InstrumentSpec.

«Двойная инкапсуляция» в программе Puka

Это не термин ООАП. Не удивляйтесь, когда ваш преподаватель странно посмотрит на вас, если вы используете эти слова на занятиях.

Давайте рассмотрим существующий уровень инкапсуляции. Нельзя ли добавить дополнительную инкапсуляцию для вывода изменяемых свойств из класса **InstrumentSpec**?

В главе 1 мы поняли, что эти свойства должны использоваться как покупателями, так и инструментами, поэтому мы создали класс InstrumentSpec, чтобы абстрагировать их из класса Instrument.



Проблема заключается в том, что набор свойств зависит от инструмента, поэтому теперь нам приходится добавлять субкласс для каждого типа инструментов.

Так как состав свойств изменяется, мы хотим вывести переменные свойства из класса **InstrumentSpec**. Нужен механизм для обращения к этим свойствам и их значениям, но без жесткого кодирования этих свойств в классе **InstrumentSpec**. Есть какие-нибудь мысли относительно того, как это сделать?

Как вы думаете, какие типы можно было бы использовать для представления свойств и обращения к их значениям, но так, чтобы для поддержки новых свойств не приходилось изменять класс InstrumentSpec?

_____

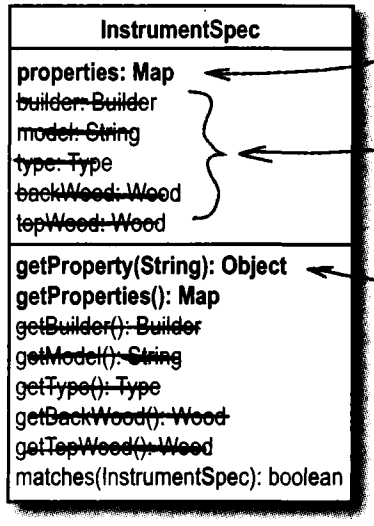
_____

_____

Инкапсуляция переменных аспектов делает ваше приложение более гибким и упрощает его модификацию.

Динамические свойства инструментов

И что вы предложили на предыдущей странице для хранения свойств? Мы решили, что контейнер **Map** отлично подойдет для работы с разными типами свойств и позволит в любой момент добавлять новые свойства:

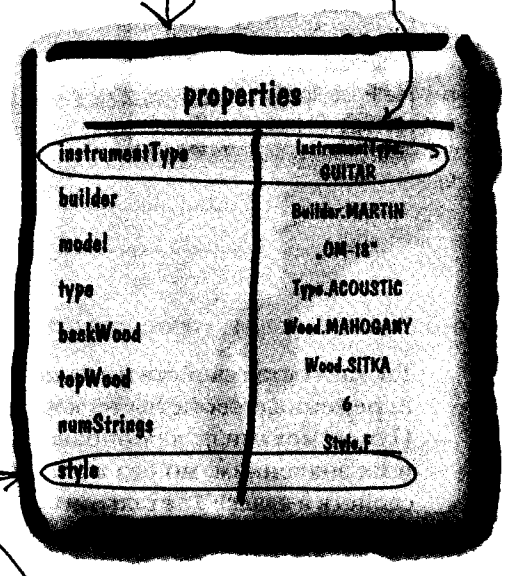


Теперь для хранения всех свойств используется единственная переменная Map.

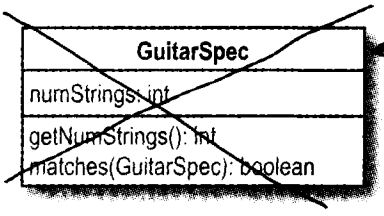
Мы избавляемся от этих свойств (и связанных с ними методов) и в дальнейшем используем для всех операций карту свойств.

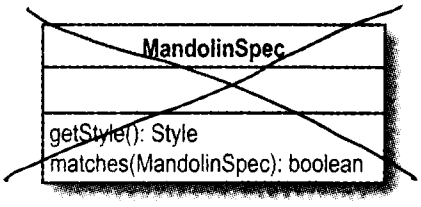
Метод getProperty() будет использоваться для возвращения значений свойств... Он запрашивает у карты значение, связанное с объектом String, переданным при вызове метода.

Это свойство сообщает, с каким типом инструмента мы имеем дело.



И что еще лучше, теперь мы можем отказаться от многочисленных субклассов InstrumentSpec!



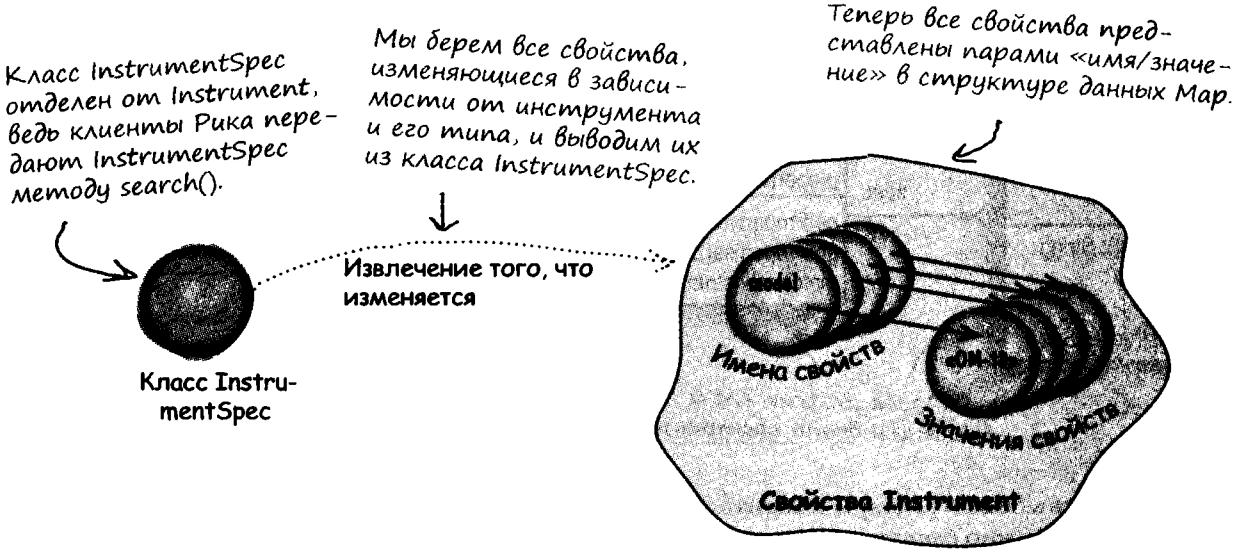


Субклассы InstrumentSpec создавались по единственной причине: для поддержки дополнительных свойств конкретных инструментов.

Свойства, которые находились в этих субклассах, просто перемещаются в карту из InstrumentSpec.

Что мы сделали: подробности

Каждый раз, когда в приложении обнаруживается нечто, что может изменяться, ищите возможности инкапсуляции. В примере с **InstrumentSpec** мы видим, что изменяется состав свойств инструментов.



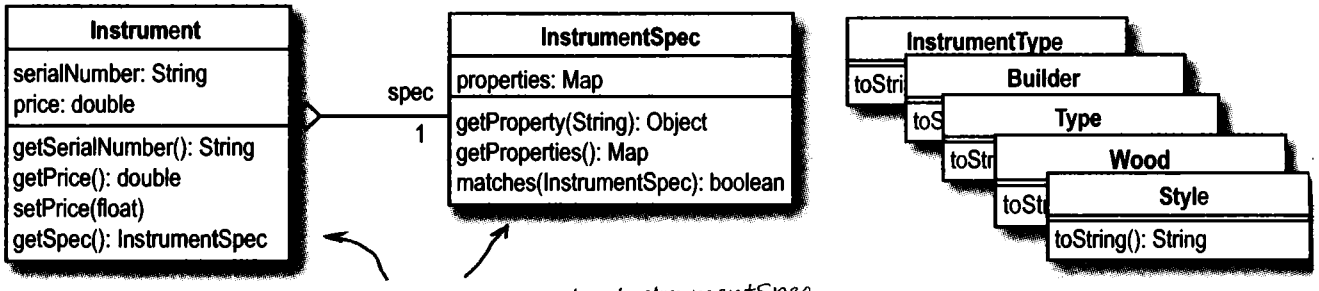
Если у вас имеется набор свойств, состав которого изменяется в зависимости от объекта, организуйте динамическое хранение этих свойств в коллекции (например, Map).

Из классов уйдет множество лишних методов, и вам не придется изменять код при добавлении новых свойств в приложение.

Использование новых классов Instrument и InstrumentSpec

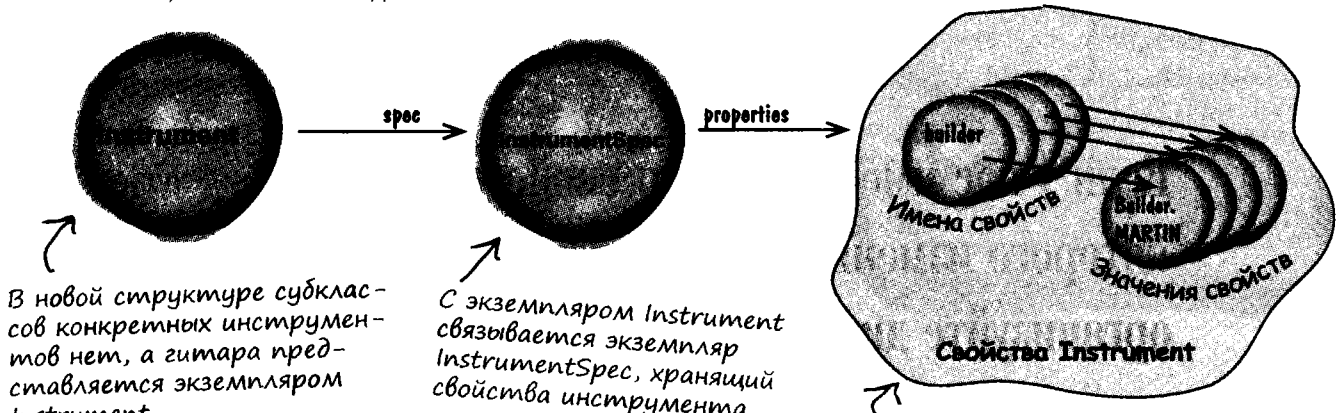
Еще раз посмотрим, как наши новые классы **Instrument** и **InstrumentSpec** работают на практике. Сейчас структура кода приложения выглядит так:

Перечисляемые типы, используемые контейнером Map класса InstrumentSpec:



Оба класса, Instrument и InstrumentSpec, перестали быть абстрактными.

Если вы обращаетесь к объекту гитары и хотите знать, кто ее изготовил, то это можно сделать так:



В новой структуре субклассов конкретных инструментов нет, а гитара представляется экземпляром Instrument.

С экземпляром Instrument связывается экземпляр InstrumentSpec, хранящий свойства инструмента.

А InstrumentSpec содержит контейнер Map с именами и значениями свойств.

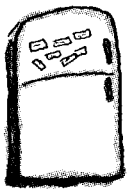
```
instrument
```

```
getSpec()
```

```
getProperty("builder")
```

```
instrument.getSpec().getProperty("builder");
```

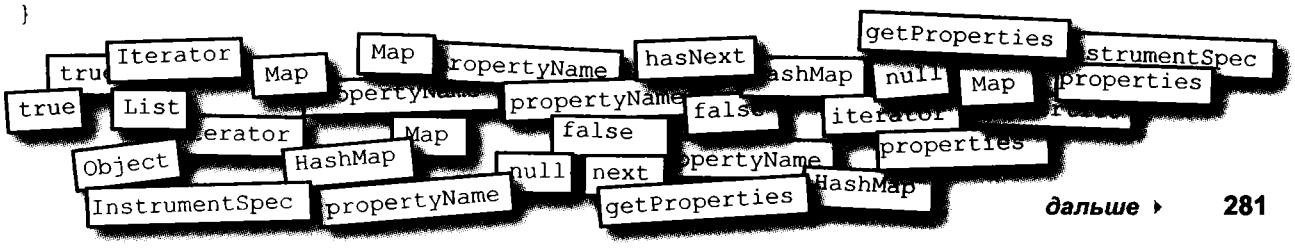
Возвращает изготовителя гитары.

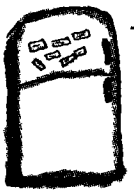


Развлечения с магнитами

Идея хранения свойств в Map выглядит неплохо, но давайте посмотрим, на что похож код с использованием новой версии InstrumentSpec. Расставьте по местам магниты, находящиеся в нижней части страницы, и завершите код.

```
import java.util._________;
import java.util._________;
import java.util._________;

public class InstrumentSpec {

  private ______ properties;

  public InstrumentSpec(______ ____________) {
    if (properties == ______) {
      this.properties = new _________();
    } else {
      this.properties = new _________(____________);
    }
  }

  public ________ getProperty(String ____________) {
    return properties.get(____________);
  }

  public ______ getProperties() {
    return ______________;
  }

  public boolean matches(______________ otherSpec) {
    for (__________ i = otherSpec.______________().keySet().____________();
         i._________(); ) {
      String ______________ = (String)i.______();
      if (!properties.get(______________).equals(
           otherSpec.getProperty(______________))) {
        return _________;
      }
    }
    return _________;
  }
}
```





Развлечения с магнитами. Решение

Идея хранения свойств в Map выглядит неплохо, но давайте посмотрим, на что похож код с использованием новой версии InstrumentSpec. Расставьте по местам магниты, находящиеся в нижней части страницы, и завершите код.

```
import java.util.Iterator;
import java.util.HashMap;
import java.util.Map;

public class InstrumentSpec {

  private Map properties;

  public InstrumentSpec(Map properties) {
    if (properties == null) {
      this.properties = new HashMap();
    } else {
      this.properties = new HashMap(properties);
    }
  }

  public Object getProperty(String propertyName) {
    return properties.get(propertyName);
  }

  public Map getProperties() {
    return properties;
  }

  public boolean matches(InstrumentSpec otherSpec) {
    for (Iterator i = otherSpec.getProperties().keySet().iterator();
         i.hasNext(); ) {
      String propertyName = (String)i.next();
      if (!properties.get(propertyName).equals(
          otherSpec.getProperty(propertyName)) {
        return false;
      }
    }
    return true;
  }
}
```

Здесь можно использовать любую реализацию интерфейса Map.

Не ошибитесь с порядком этих двух значений; в противном случае matches() всегда будет возвращать ошибочный результат.

getProperties
null
InstrumentSpec
List

Часто задаваемые вопросы

В: Теперь оба класса — Instrument и InstrumentSpec — стали конкретными?

О: Правильно. Класс `Instrument` представляет уже не концепцию, а инструмент из каталога Рика. Класс `InstrumentSpec` передается клиентами с набором данных искомого инструмента, а также используется классом `Instrument` для хранения свойств инструмента.

В: И я могу избавиться от субклассов Guitar и Mandolin?

О: Да. А также от `Banjo`, `Dobro` и любых других субклассов `Instrument`, которые вы создали в своей программе.

В: И это связано с тем, что класс Instrument используется напрямую?

О: Точно! Вспомните, что субклассы обычно создаются из-за изменения поведения. В субклассах `Instrument` поведение оставалось неизменным; для каждого субкласса просто создавался новый конструктор. Это привело к созданию многочисленных классов и снижению гибкости приложения, но не предоставило никакой полезной функциональности.

В: Я понимаю, зачем мы избавились от Guitar и Mandolin, но не до конца уловил, почему мы отказались от разных субклассов InstrumentSpec.

О: Ничего страшного, это один из самых нетривиальных аспектов структуры кода приложения Рика. Вспомните: один из ключевых принципов ОО-проектирования требует инкапсулировать то, что изменяется. В приложении Рика менялись свойства инструментов. Мы извлекли эти свойства из `InstrumentSpec` и поместили в `Map`. Теперь при добавлении инструмента с новым свойством оно добавляется в виде пары «имя/значение» в контейнер `Map`.

В: С уменьшением количества классов программа становится более гибкой?

О: В данном случае это так, но в некоторых ситуациях гибкость программы улучшается с добавлением новых классов. Добавление класса `InstrumentSpec` помогало разделить инструменты и свойства, и это было хорошо; но в этой главе мы удалили классы, и это упростило добавление новых инструментов в каталог Рика.

В: Ни за что бы не догадался, что в этом приложении можно обойтись без субклассов инструментов или их спецификаций. Как этому научиться?

О: Лучший способ освоить проектирование программ — писать программы! В приложении Рика нам пришлось опробовать и ошибочные решения (например, добавление классов `Guitar` и `Mandolin`), чтобы найти правильный путь.

К большинству хороших структур разработчик приходит через плохие структуры; редко когда все получается правильно с первого раза. Начните с проектирования разумной структуры, а затем постарайтесь усовершенствовать результат посредством применения ОО-принципов и паттернов.

Большинство хороших структур строится на основе анализа плохих структур.

Не бойтесь совершать ошибки и исправлять их.

Завершаем приложение Рика: перечисление InstrumentType

Мы почти доделали отличную программу. Осталось воплотить в жизнь новые идеи начиная с создания перечисляемого типа для разновидностей инструментов:

Типы инструментов, которые продает Рик (на данный момент).

```
public enum InstrumentType {

  GUITAR, BANJO, DOBRO, FIDDLE, BASS, MANDOLIN;

  public String toString() {
    switch(this) {
      case GUITAR:   return "Guitar";
      case BANJO:    return "Banjo";
      case DOBRO:    return "Dobro";
      case FIDDLE:   return "Fiddle";
      case BASS:     return "Bass";
      case MANDOLIN: return "Mandolin";
      default:       return "Unspecified";
    }
  }
}
```

InstrumentType.java

InstrumentType
toString(): String

toString() упрощает вывод результатов.

И Inventory тоже нужно обновить

С изменениями в **Instrument** и **InstrumentSpec** наш класс **Inventory** значительно упрощается:

Inventory
inventory: Instrument [*]
addInstrument(String, double, InstrumentSpec) get(String): Instrument search(InstrumentSpec): Instrument [*]

```
public class Inventory {

  public void addInstrument(String serialNumber, double price,
                            InstrumentSpec spec) {
    Instrument instrument = null;
    if (spec instanceof GuitarSpec) {
      instrument = new Guitar(serialNumber, price, (GuitarSpec)spec);
    } else if (spec instanceof MandolinSpec) {
      instrument = new Mandolin(serialNumber, price, (MandolinSpec)spec);
    }
    Instrument instrument = new Instrument(serialNumber, price, spec);
    inventory.add(instrument);
  }

  // И т.д.
}
```

Добавление инструментов сильно упростилось.

Теперь приложение может создавать экземпляры Instrument (вместо экземпляров субклассов), ведь класс перестал быть абстрактным.

Inventory.java

Возьми в руку карандаш

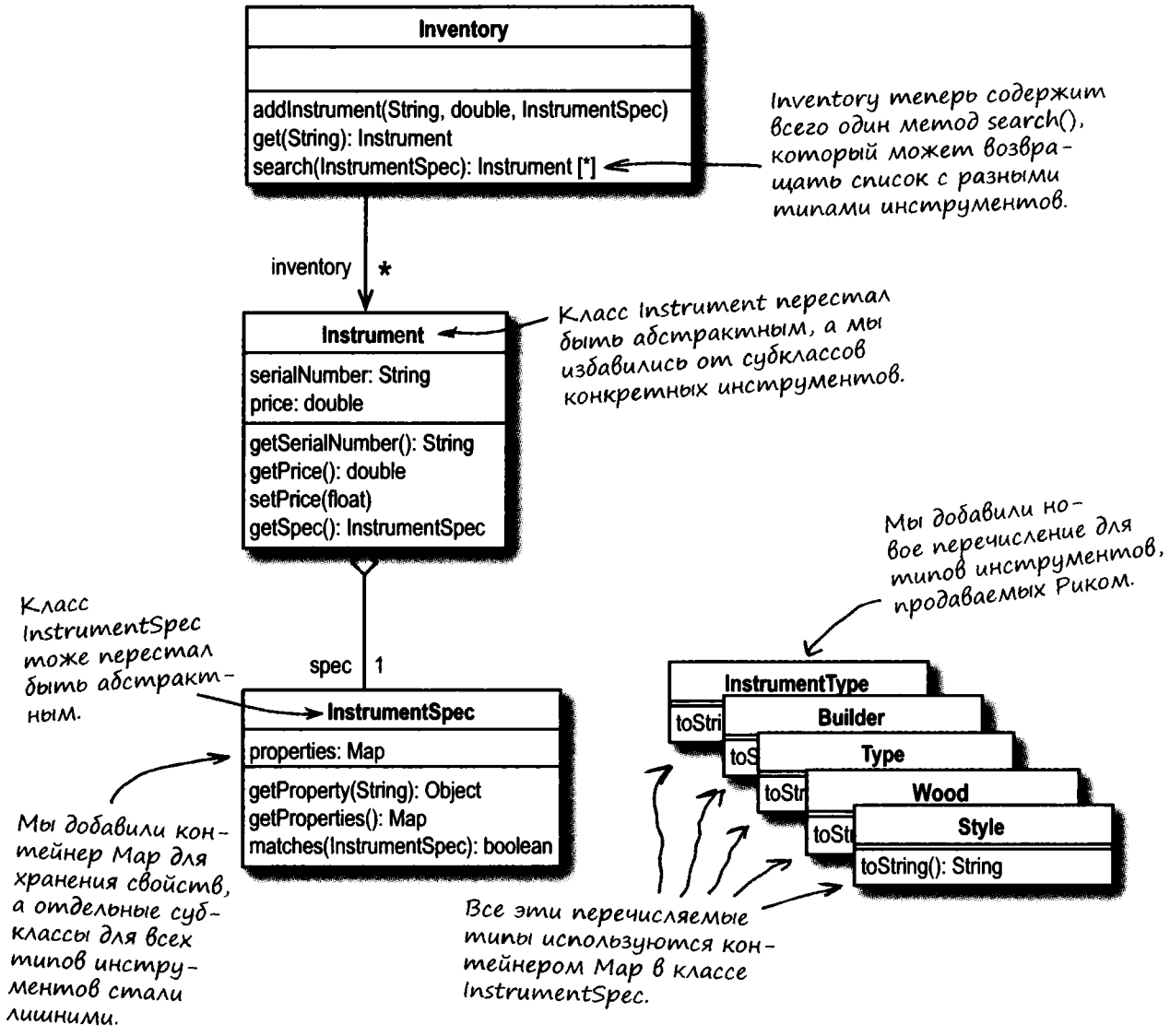
Давайте посмотрим, что же было реально сделано.

Мы внесли множество изменений в программу Рика и все ради «большей гибкости». Вернитесь к диаграмме классов приложения Рика на с. 262 и вспомните, как она выглядела на момент начала работы. Затем нарисуйте внизу, как эта диаграмма выглядит сейчас.

Ответы на следующей странице!

Смотрите: гибкое приложение Рика!

Мы внесли множество изменений в приложение Рика... Легко забыть, ради чего все это делалось. Но взгляните на приведенную ниже диаграмму классов: вы увидите, насколько упростилось приложение.



А приложение действительно работает?

Программа Рика выглядит гораздо лучше, чем в начале главы, и уж конечно, лучше, чем при добавлении всех этих субклассов для банджо и мандолин. Но мы еще должны убедиться в том, что изменения не нарушили работоспособность программы. Обновим наш тестовый класс и проверим, как работает поиск в новой версии:

FindInstrument.java

```
public class FindInstrument {

  public static void main(String[] args) {
    // Инициализация каталога гитар
    Inventory inventory = new Inventory();
    initializeInventory(inventory);

    Map properties = new HashMap();
    properties.put("builder", Builder.GIBSON);
    properties.put("backWood", Wood.MAPLE);
    InstrumentSpec clientSpec = new InstrumentSpec(properties);

    List matchingInstruments = inventory.search(clientSpec);
    if (!matchingInstruments.isEmpty()) {
      System.out.println("You might like these instruments:");
      for (Iterator i = matchingInstruments.iterator(); i.hasNext(); ) {
        Instrument instrument = (Instrument)i.next();
        InstrumentSpec spec = instrument.getSpec();
        System.out.println("We have a " + spec.getProperty("instrumentType") +
          " with the following properties:");
        for (Iterator j = spec.getProperties().keySet().iterator();
             j.hasNext(); ) {
          String propertyName = (String)j.next();
          if (propertyName.equals("instrumentType"))
            continue;
          System.out.println("    " + propertyName + ": " +
            spec.getProperty(propertyName));
        }
        System.out.println("  You can have this " +
          spec.getProperty("instrumentType") + " for $" +
          instrument.getPrice() + "\n---");
      }
    } else {
      System.out.println("Sorry, we have nothing for you.");
    }
  }

  // Здесь размещается метод initializeInventory()
}
```

Теперь клиент заполняет данными запроса объект InstrumentSpec. Так как тип инструмента при этом не указывается, запрос может вернуть гитары, мандолины и вообще любые инструменты, которые продает Рик.

Мы работаем с контейнером Map, используемым InstrumentSpec, но зато операция перебора свойств каждого инструмента и их вывода значительно упростилась.

Пропускаем свойство instrumentType, оно уже было обработано до начала цикла.

Нужно добавить в каталог Рика другие инструменты, чтобы поиск не ограничивался гитарами... Мы сделаем это на следующей странице.

Обзор инструментов

В ходе тестирования новой версии программы Рика поиск не должен ограничиваться гитарами. Напишите код метода initializeInventory() для FindInstrument.java и включите в каталог данные нескольких гитар, мандолин и банджо. Ниже перечислены инструменты, которые в данный момент находятся на складе Рика, и даже написан код добавления данных первой гитары.

Гитары

Collings CJ 6-string acoustic, Indian Rosewood back and sides, Spruce top, Serial #11277, for $3999.95

Martin D-18 6-string acoustic, Mahogany back and sides, Adirondack top, Serial #122784, for $5495.95

Fender stratocastor 6-string electric, Alder back and sides and top, Serial #V95693, for $1499.95

Fender stratocastor 6-string electric, Alder back and sides and top, Serial #V9512, for $1549.95

Gibson SG '61 Reissue 6-string electric, Mahogany back, sides, and top, Serial #82765501, for $1890.95

Gibson Les Paul 6-string electric, Maple back, sides, and top, Serial #70108276, for $2295.95

Мандолины

Gibson F5-G acoustic mandolin, Maple back, sides, and top, Serial #9019920, for $5495.99

Не забудьте: атрибут numStrings к мандолинам не относится.

Банджо

Gibson RB-3 5-string acoustic banjo, Maple back and sides, Serial #8900231, for $2945.95

У банджо нет свойства topWood.

Ответы на с. 290

Начало метода initializeInventory() с включением в каталог данных первой гитары.

```
private static void initializeInventory(Inventory inventory) {
  Map properties = new HashMap();
  properties.put("instrumentType", InstrumentType.GUITAR);
  properties.put("builder", Builder.COLLINGS);
  properties.put("model", "CJ");
  properties.put("type", Type.ACOUSTIC);
  properties.put("numStrings", 6);
  properties.put("topWood", Wood.INDIAN_ROSEWOOD);
  properties.put("backWood", Wood.SITKA);
  inventory.addInstrument("11277", 3999.95,
    new InstrumentSpec(properties));
  // далее следует ваш код
}
```

Напишите код для добавления других инструментов, приведенных выше.

FindInstrument.java

Пробный запуск улучшенной программы Рика

Добавьте данные всех инструментов с предыдущей страницы в метод **initializeInventory()** класса **FindInstrument.java**, после чего откомпилируйте все классы. Теперь программа Рика готова к тестированию...

...вернее, почти готова. Сначала необходимо разобраться, какие результаты должен вернуть поиск, использующий обновленную версию **FindInstrument**. Набор параметров, полученных от клиента Рика, выглядит так:

Клиент Рика не указал тип инструмента, но он хочет инструмент от Gibson с кленовой нижней декой.

```
Map properties = new HashMap();
properties.put("builder", Builder.GIBSON);
properties.put("backWood", Wood.MAPLE);
InstrumentSpec clientSpec =
  new InstrumentSpec(properties);
```

FindInstrument.java

Просмотрите список инструментов на предыдущей странице. Руководствуясь этой спецификацией, запишите, какие гитары, мандолины и банджо должна вернуть поисковая программа Рика:

File Edit Window Help TheSearchIsOn

```
%java FindInstrument
You might like these instruments:
```

Запишите, какие инструменты, по вашему мнению, вернет поиск FindInstrument для текущего каталога Рика.

Обзор инструментов. Решение

В ходе тестирования новой версии программы Рика поиск не должен ограничиваться гитарами. Напишите код метода initializeInventory() для FindInstrument.java и включите в каталог данные нескольких гитар, мандолин и банджо.

```
private static void initializeInventory(Inventory inventory) {
  Map properties = new HashMap();
  properties.put("instrumentType", InstrumentType.GUITAR);
  properties.put("builder", Builder.COLLINGS);
  properties.put("model", "CJ");
  properties.put("type", Type.ACOUSTIC);
  properties.put("numStrings", 6);
  properties.put("topWood", Wood.INDIAN_ROSEWOOD);
  properties.put("backWood", Wood.SITKA);
  inventory.addInstrument("11277", 3999.95,
    new InstrumentSpec(properties));

  properties.put("builder", Builder.MARTIN);
  properties.put("model", "D-18");
  properties.put("topWood", Wood.MAHOGANY);
  properties.put("backWood", Wood.ADIRONDACK);
  inventory.addInstrument("122784", 5495.95,
    new InstrumentSpec(properties));

  properties.put("builder", Builder.FENDER);
  properties.put("model", "Stratocastor");
  properties.put("type", Type.ELECTRIC);
  properties.put("topWood", Wood.ALDER);
  properties.put("backWood", Wood.ALDER);
  inventory.addInstrument("V95693", 1499.95,
    new InstrumentSpec(properties));
  inventory.addInstrument("V9512", 1549.95,
    new InstrumentSpec(properties));
```

Collings CJ 6-string acoustic, Indian Rosewood back and sides, Spruce top, Serial #11277, for $3999.95

Один и тот же контейнер Map используется снова и снова.

Martin D-18 6-string acoustic, Mahogany back and sides, Adirondack top, Serial #122784, for $5495.95

Fender stratocastor 6-string electric, Alder back and sides and top, Serial #V95693, for $1499.95

Fender stratocastor 6-string electric, Alder back and sides and top, Serial #V9512, for $1549.95

Спецификации этих двух гитар полностью совпадают; различаются только свойства в Instrument.

```
    properties.put("builder", Builder.GIBSON);
    properties.put("model", "Les Paul");
    properties.put("topWood", Wood.MAPLE);
    properties.put("backWood", Wood.MAPLE);
    inventory.addInstrument("70108276", 2295.95,
      new InstrumentSpec(properties));

    properties.put("model", "SG '61 Reissue");
    properties.put("topWood", Wood.MAHOGANY);
    properties.put("backWood", Wood.MAHOGANY);
    inventory.addInstrument("82765501", 1890.95,
      new InstrumentSpec(properties));

    properties.put("instrumentType", InstrumentType.MANDOLIN);
    properties.put("type", Type.ACOUSTIC);
    properties.put("model", "F-5G");
    properties.put("backWood", Wood.MAPLE);
    properties.put("topWood", Wood.MAPLE);
    properties.remove("numStrings");
    inventory.addInstrument("9019920", 5495.99,
      new InstrumentSpec(properties));

    properties.put("instrumentType", InstrumentType.BANJO);
    properties.put("model", "RB-3 Wreath");
    properties.remove("topWood");
    properties.put("numStrings", 5);
    inventory.addInstrument("8900231", 2945.95,
      new InstrumentSpec(properties));
    }
}
```

Gibson Les Paul 6-string electric, Maple back, sides, and top, Serial #70108276, for $2295.95

Gibson SG '61 Reissue 6-string electric, Mahogany back, sides, and top, Serial #82765501, for $1890.95

Не забудьте удалить numStrings для мандолины, если вы повторно используете объект properties.

Gibson F5-G acoustic mandolin, Maple back, sides, and top, Serial #9019920, for $5495.99

У банджо нет верхней деки, поэтому это свойство удаляется.

Gibson RB-3 5-string acoustic banjo, Maple back and sides, Serial #8900231, for $2945.95

FindInstrument.java

Рик получает готовую программу. У его клиентов есть три варианта:

Клиент Рика подбирает нужный вариант из трех видов инструментов: гитара, мандолина и банджо.

```
File Edit Window Help SatisfyTheCustomer
%java FindInstrument
You might like these instruments:
We have a Guitar with the following properties:
    topWood: Maple
    backWood: Maple
    builder: Gibson
    type: electric
    model: Les Paul
    numStrings: 6
  You can have this Guitar for $2295.95
---
We have a Mandolin with the following properties:
    topWood: Maple
    backWood: Maple
    builder: Gibson
    type: acoustic
    model: F-5G
  You can have this Mandolin for $5495.99
---
We have a Banjo with the following properties:
    backWood: Maple
    builder: Gibson
    type: acoustic
    model: RB-3 Wreath
    numStrings: 5
  You can have this Banjo for $2945.95
---
```

Гитара соответствует заданной спецификации: изготовитель Gibson, нижняя дека и обечайки изготовлены из клена.

Мандолина от Gibson с кленовой нижней декой... Тоже соответствует спецификации.

И еще один инструмент из клена от Gibson... На этот раз банджо. У банджо нет верхней деки, но это неважно.

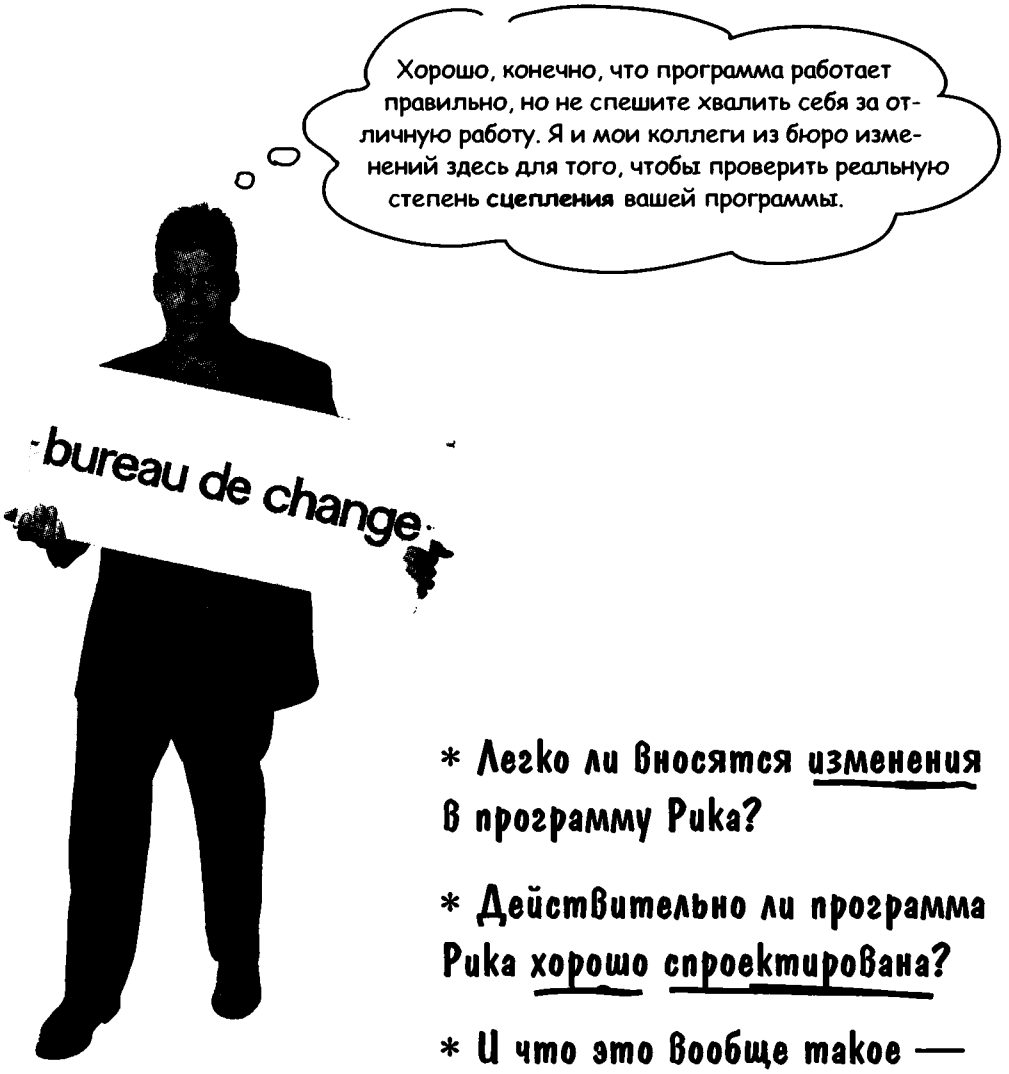
Часто задаваемые вопросы

В: Мои результаты не совпадают с вашими. Что я сделал не так?

О: Если ваша версия программы возвращает другие гитары (или те же гитары, но с другими свойствами), убедитесь в том, что ваш каталог содержит те же инструменты, что и наш. Проверьте упражнение на с. 288 и ответы на с. 261–262 и сравните содержимое своего каталога с нашим.

В: У нас в каталоге всего одно банджо и одна мандолина... Можно ли это считать надежным тестом?

О: Хороший вопрос. Да, вы правы, лучше было бы создать несколько мандолин и несколько банджо и убедиться в том, что программа отбирает только подходящие мандолины и банджо. Попробуйте добавить несколько неподходящих банджо и мандолин и протестируйте программу Рика с расширенным набором инструментов.

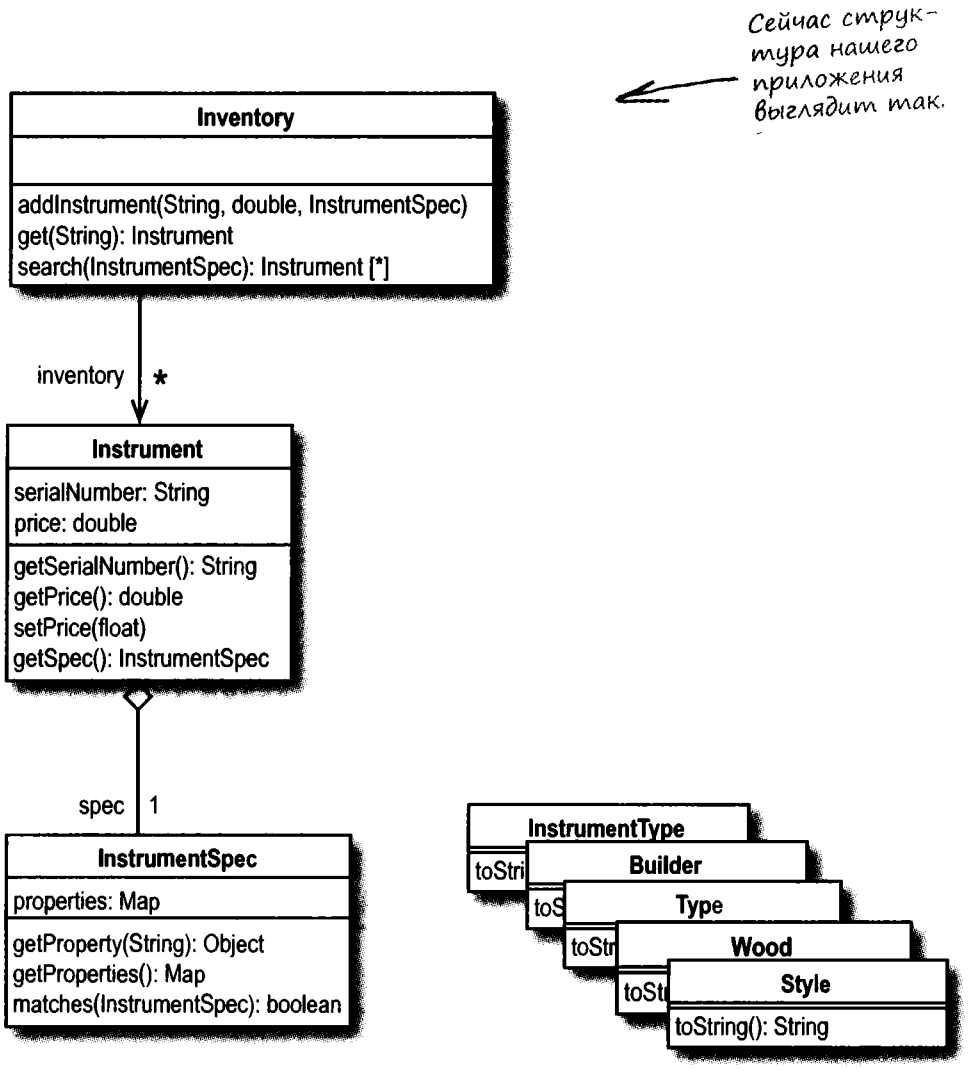


* Легко ли вносятся изменения в программу Рика?
* Действительно ли программа Рика хорошо спроектирована?
* И что это вообще такое — сцепление?

Большая проверка на простоту внесения изменений

Легко ли внести изменения в приложение?

Давайте включим в приложение Рика поддержку гитар «добро» и скрипок. Мы уже пытались сделать это ранее, в первой части главы 5, и кончилось все полным хаосом. Сейчас все должно быть намного проще, верно? Ниже приведена диаграмма классов для текущей версии программы.



bureau de change

Проверка простоты внесения изменений — один из лучших способов проверки качества структуры программного продукта.

Применим к нашей программе **проверку простоты изменения:**

1. Сколько классов придется ***добавить*** для поддержки новых типов инструментов?

2. Сколько классов придется ***изменить*** для поддержки новых типов инструментов?

3. Предположим, Рик захотел хранить в каталоге год изготовления инструментов. Сколько классов придется ***изменить***, чтобы обеспечить хранение новой информации?

4. Рик также хочет добавить новое свойство neckWood для хранения материала, из которого изготовлен гриф инструмента. Сколько классов придется ***изменить*** для поддержки этого свойства?

Ответы на с. 296

Большая проверка на простоту изменений

Легко ли внести изменения в приложение?

Давайте включим в приложение Рика поддержку гитар «добро» и скрипок. Мы уже пытались сделать это ранее, в первой части главы 5, и кончилось все полным хаосом. Сейчас все должно быть намного проще, верно?

Применим к нашей программе **проверку простоты изменения:**

1. Сколько классов придется ***добавить*** для поддержки новых типов инструментов?

   Ни одного! Мы избавились от субклассов Instrument и InstrumentSpec для конкретных инструментов.

2. Сколько классов придется ***изменить*** для поддержки новых типов инструментов?

   Один: новые типы инструментов добавляются в перечисляемый тип InstrumentType.

3. Предположим, Рик захотел хранить в каталоге год изготовления инструментов. Сколько классов придется ***изменить***, чтобы обеспечить хранение новой информации?

   Ни одного! Год изготовления инструмента просто сохраняется в карте properties из InstrumentSpec.

4. Рик также хочет добавить новое свойство neckWood для хранения материала, из которого изготовлен гриф инструмента. Сколько классов придется ***изменить*** для поддержки этого свойства?

   В худшем случае — один, а может, и ни одного! neckWood — еще одно свойство, которое может храниться в карте InstrumentSpec... Но, возможно, нам придется добавить для него новые значения в перечисление Wood.

Прекрасно! Наша программа легко изменяется...
...но что там насчет «сцепления»?

Класс с высоким сцеплением хорошо делает что-то одно и не пытается делать что-то другое.

Чем выше сцепление классов, тем выше сцепление программного продукта.

Классы с высоким сцеплением сконцентрированы на выполнении конкретных задач. Наш класс Inventory предназначен для работы с каталогом; он не имеет отношения к выбору дерева, из которого изготовлена гитара, или сравнению двух спецификаций инструментов.

Просмотрите методы классов: все ли они имеют прямое отношение к имени класса? Если какой-то метод кажется посторонним, возможно, его лучше переместить в другой класс.

Класс Instrument не пытается выполнять поиск или отслеживать доступные типы дерева. Он сконцентрирован на описании инструмента, и ничего более.

За партой

Сцепление — степень логического сопряжения элементов одного модуля, класса или объекта. Чем выше сцепление программного продукта, тем четче определены обязанности каждого класса приложения. Каждый класс обеспечивает выполнение четко определенного набора взаимосвязанных действий.

Высокое сцепление и одна причина для изменения

Возможно, вы не осознали этого, но в этой книге сцепление уже упоминалось. Помните?

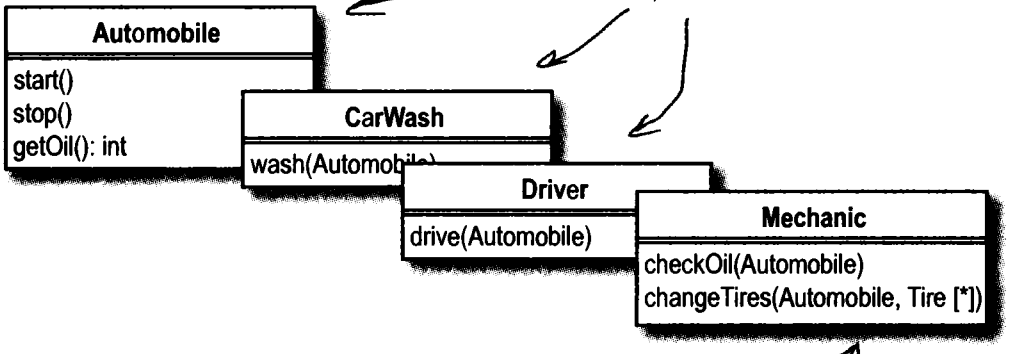
Каждый класс должен, насколько возможно, иметь только одну причину для этого. Неизменение [illegible]

Один из ответов из телевикторины «ОО-Катастрофа»! А вы помните вопрос?

Сцепление является мерой логического сопряжения функциональности классов приложения. Если класс содержит только взаимосвязанную функциональность, то у него есть только одна причина для изменения... О чем мы и говорили в **ОО-КАТАСТРОФЕ!**

Вот те классы, которые упоминались ранее, когда мы обсуждали необходимость одной причины для изменений:

Функция каждого класса четко определена. Каждый класс обладает высоким сцеплением, и это упрощает его возможное изменение без изменения других классов.



Можете ли вы предложить способ повысить сцепление класса Mechanic?

Часто задаваемые вопросы

В: Итак, сцепление – это просто модное словечко, позволяющие охарактеризовать простоту изменения приложения?

О: Не совсем. Сцепление является характеристикой построения отдельных классов, объектов и пакетов приложения. Если каждый класс выполняет четко сгруппированные функции, то, вероятно, продукт обладает высоким сцеплением. Но если класс выполняет всевозможные функции, не имеющие прямого отношения друг к другу, то он характеризуется низким сцеплением.

В: Выходит, программный продукт с высоким сцеплением обладает слабой связностью?

О: Точно! Почти всегда чем выше сцепление вашего продукта, тем слабее связность между классами. В приложении Рика класс `Inventory` занимается исключительно управлением данными каталога, а не сравнением инструментов или хранением свойств в спецификации. Следовательно, класс `Inventory` обладает высоким сцеплением. Это также означает, что он слабо связан с остальными компонентами приложения, — изменения в `Instrument`, например, не повлияют на класс `Inventory`.

В: Но ведь из этого следует, что такой программный продукт будет проще изменять?

О: В большинстве случаев — да. Но вы помните версию приложения Рика, с которой начиналась эта глава? Она поддерживала только гитары и в ней даже не было классов `Instrument` или `InstrumentSpec`. Программа обладала довольно высоким сцеплением — класс `Guitar` был слабо связан с `Inventory`. Тем не менее для поддержки мандолин нам пришлось основательно потрудиться.

При фундаментальном изменении функциональности приложения, скажем при переходе от одного типа продаваемых инструментов к нескольким, приходится вносить многочисленные изменения в структуру, которая уже обладает высоким сцеплением и слабой связностью. Таким образом, сцепление не всегда является мерой простоты изменения программы; но если работа программного продукта не подвергается радикальным изменениям, программы с высоким сцеплением обычно проще изменяются.

В: А высокое сцепление лучше низкого?

О: Лучше. В хорошо спроектированной ОО-структуре каждый класс и модуль продукта выполняет одну базовую функцию, и выполняет ее хорошо. Как только один класс начинает выполнять две или три функции, скорее всего, это свидетельствует об отходе от сцепления и хорошего стиля ОО-проектирования.

В: И наверное, высокое сцепление упрощает не только изменение, но и повторное использование кода?

О: Верно. Высокое сцепление и слабая связность способствуют созданию продукта, который легко расширяется и даже разбивается на компоненты для повторного использования из-за отсутствия взаимозависимостей между объектами.

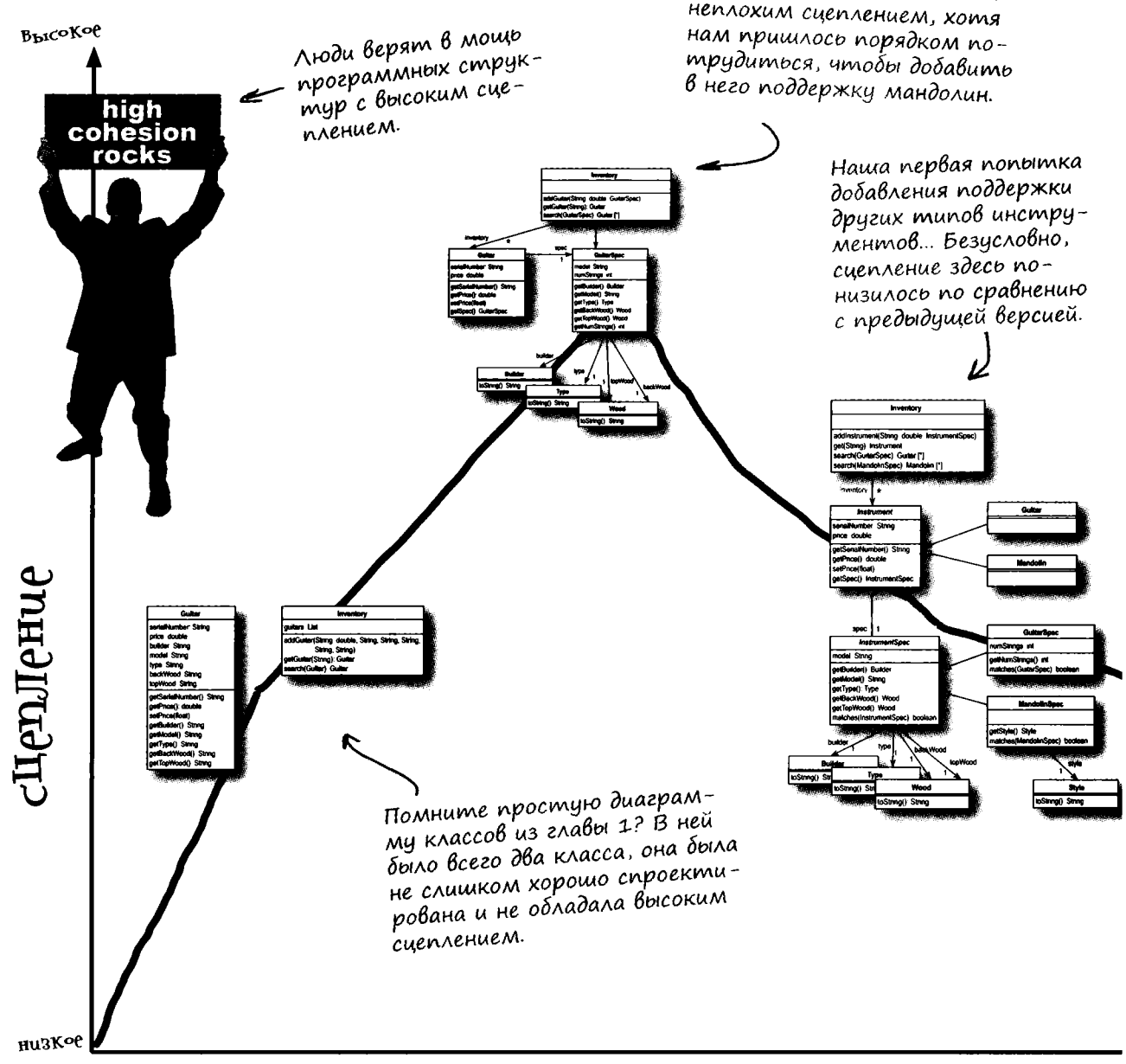
На это можно взглянуть так: чем выше сцепление в приложении, тем лучше определены функции каждого объекта. А чем лучше определен объект (и его задача), тем проще извлечь этот объект из одного контекста и заставить его выполнять ту же задачу в другом контексте. Объект справляется со своей четко определенной задачей независимо от того, где он используется.

В: И в этой главе мы повысили сцепление в приложении Рика?

О: В основном — да. Но давайте рассмотрим этот вопрос чуть подробнее...

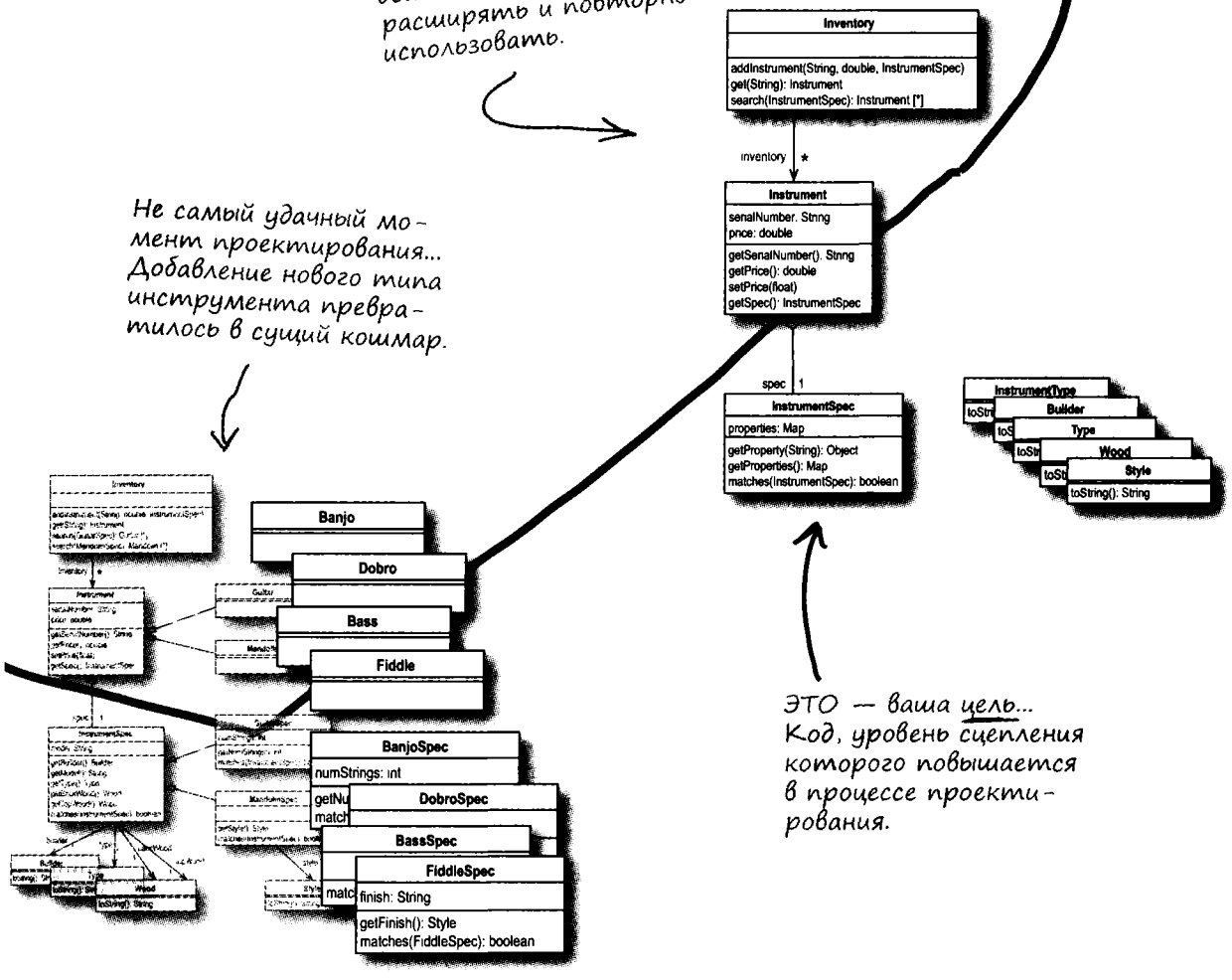
Анализ программы

Итак, привели ли наши изменения в программе Рика к повышению сцепления? Можно ли сказать, что объекты нашей программы слабо связаны? И легко ли внести изменения в программу? Давайте посмотрим:



Когда в программу вносятся изменения, постарайтесь, чтобы они приводили к повышению сцепления.

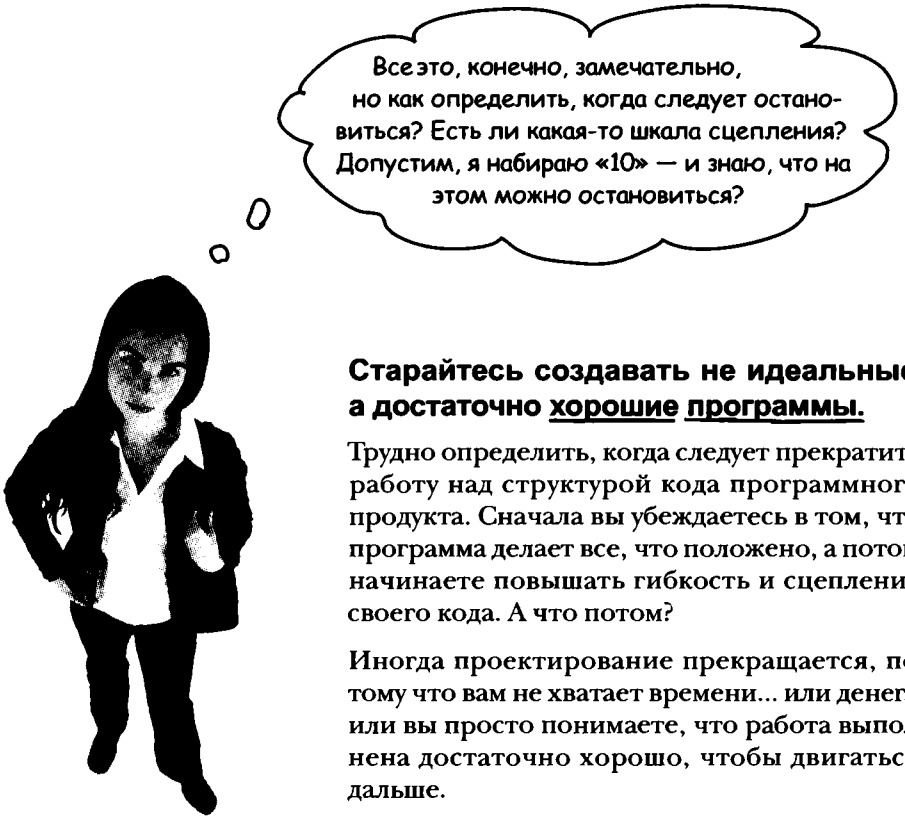
А вот как выглядит структура кода сейчас. Высокое сцепление, слабая связность — код действительно просто расширять и повторно использовать.



Не самый удачный момент проектирования... Добавление нового типа инструмента превратилось в сущий кошмар.

ЭТО — ваша цель... Код, уровень сцепления которого повышается в процессе проектирования.

Жизненный цикл структуры кода



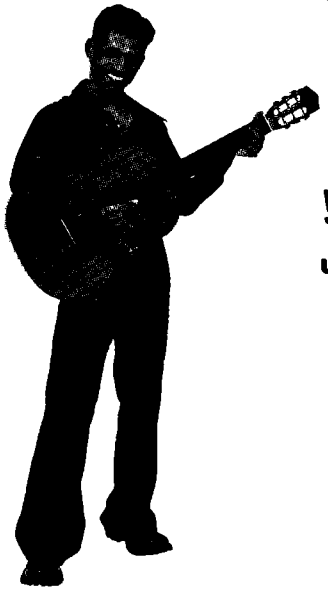
Старайтесь создавать не идеальные, а достаточно хорошие программы.

Трудно определить, когда следует прекратить работу над структурой кода программного продукта. Сначала вы убеждаетесь в том, что программа делает все, что положено, а потом начинаете повышать гибкость и сцепление своего кода. А что потом?

Иногда проектирование прекращается, потому что вам не хватает времени... или денег... или вы просто понимаете, что работа выполнена достаточно хорошо, чтобы двигаться дальше.

Если программа работает, а заказчик доволен, если вы приложили все усилия к тому, чтобы программа была хорошо спроектирована, — возможно, пришло время браться за следующий проект. Попытки создать «идеальную программу» оборачиваются напрасной тратой времени; с другой стороны, время, потраченное на написание хорошей программы, с закономерным переходом к следующей программе, наверняка обернется новыми заказами, повышениями по работе и премиями.

Когда следует сказать: «Достаточно!»



Мне нравится! Я наконец-то могу продавать любые инструменты, какие захочу, а программа помогает моим клиентам найти то, что им нужно.

Прежде чем выходить из проекта, всегда убедитесь в том, что ваша программа делает то, что ей положено делать.

Убедитесь в том, что заказчик доволен

Хорошая структура кода. Высокое сцепление, слабо связанные классы... Когда в следующий раз Рик захочет что-нибудь изменить, мы сможем легко это сделать.

Разобравшись с функциональностью, переходите к хорошим структурным решениям, основанным на проверенных ОО-принципах, для улучшения гибкости.

Убедитесь в том, что структура кода получилась гибкой

Если оба условия выполнены, возможно, пора переходить... К следующему проекту, следующему приложению и даже к следующей главе!

Инструментарий ООАП

С того момента, когда мы начали работать на Рика в главе 1, мы прошли долгий путь. Вы узнали много нового о проектировании, поэтому будет нелишне привести краткую сводку новых дополнений в вашем инструментарии.

В этой главе мы основательно поработали над структурой кода приложения. Ниже приведен краткий обзор основных моментов.

Требования

Хорошие требован
том, что ваша си
хочет заказчик.

Проследите за тел
пространялись на
пользования вашей

Из вариантов испо
то, о чем заказчик

Анализ вариантов
явить неполные ил
вания.

Требования всегда и
временем.

Анализ и проектирование

Хорошо спроектированный код легко изменяется и расширяется.

Используйте базовые ОО-принципы (инкапсуляцию и наследование) для повышения гибкости программы.

Если структура кода приложения недостаточно гибка, ИЗМЕНИТЕ ЕЕ! Не используйте плохую структуру, даже если она создана вами.

Классы должны обладать высоким сцеплением; каждый класс должен хорошо делать ЧТО-ТО ОДНО.

На протяжении всего жизненного цикла структуры кода стремитесь к повышению сцепления.

Цель качественного проектирования — код с высоким сцеплением и слабой связностью.

ОО-принципы

Инкапсулируйте то, что изменяется.

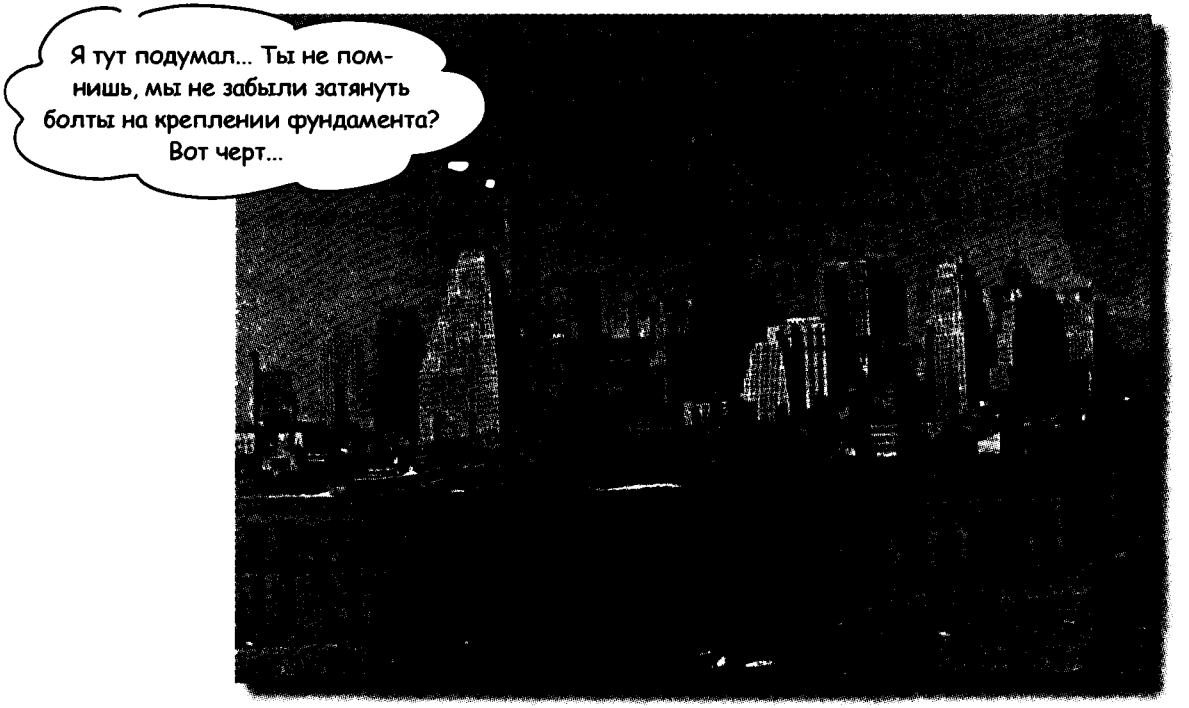
Программируйте для интерфейса, а не для реализации.

Каждый класс приложения должен иметь только одну причину для изменения.

Классы создаются ради поведения и функциональности.

В этой главе наш инструментарий пополнился несколькими ОО-принципами.

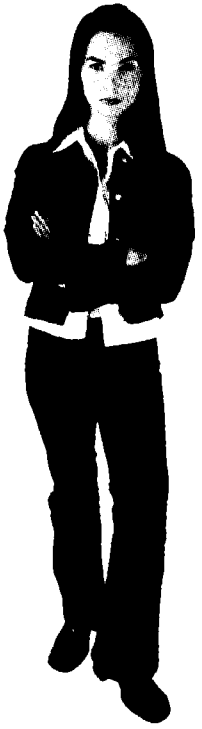
«Меня зовут Арт... И я архитектор»



Пора построить нечто ДЕЙСТВИТЕЛЬНО БОЛЬШОЕ. Готовы?

В вашем инструментарии ООАП скопилось множество инструментов, но как использовать эти инструменты при построении **большого проекта**? Возможно, вы этого и не осознали, но у вас имеется все необходимое для решения больших задач. В этой главе мы рассмотрим новые инструменты — **анализ предметной области** и **диаграммы вариантов использования**. Но даже эти инструменты основаны на том, что вам уже известно, — на необходимости прислушиваться к мнению заказчика и необходимости понимания того, что вы строите, до начала работы над кодом. Приготовьтесь... Пора поиграть в архитектора.

Разговоры о хороших программах — это очень здорово, но настоящие приложения содержат куда больше 5 или 10 классов. Как превратить **большое приложение** в хорошую программу?



Большие задачи решаются точно так же, как и маленькие.

До сих пор мы работали с относительно простыми приложениями. Приложение для магазина гитар в своем худшем состоянии содержало не более 15 классов, а в системе собачьих дверей их количество не превышало 5. Но все, о чем говорилось ранее в книге, в равной степени относится и к большим приложениям.

1. Убедитесь в том, что поведение программы соответствует требованиям заказчика.

Помните эти шаги на пути к написанию хороших программ? К огромным приложениям, содержащим 1000+ классов, они применимы в той же мере, что и к приложениям с двумя-тремя классами.

2. Применяйте базовые ОО-принципы для повышения гибкости.

3. Постарайтесь создать структуру кода, упрощающую его сопровождение и повторное использование.

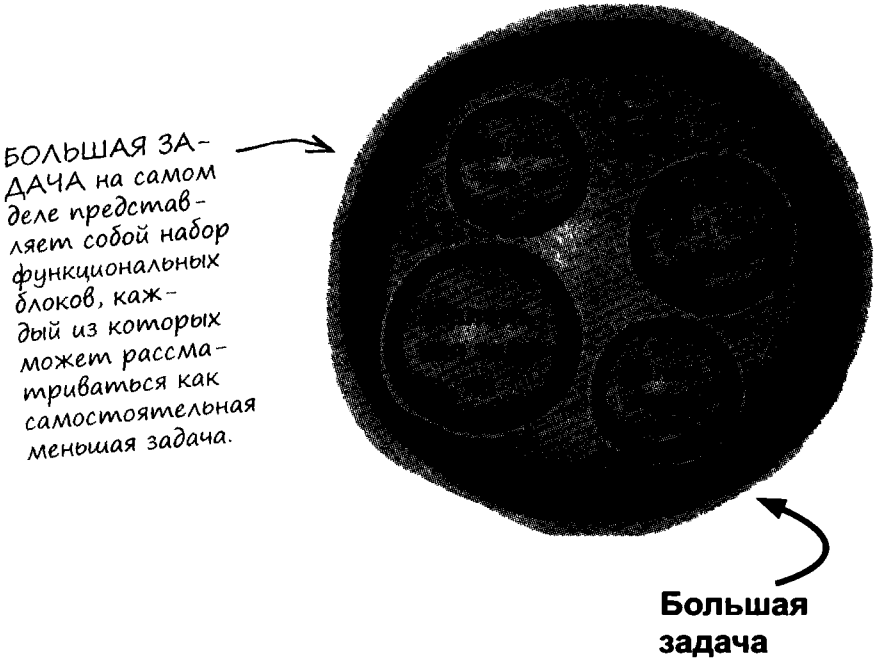
Все дело в подходе к решению большой задачи

Как мы обычно работаем над большими задачами в крупных программных проектах? Мы рассматриваем общую картину, а потом начинаем работать над отдельной частью функциональности приложения.

Большую задачу лучше всего рассматривать как совокупность отдельных функциональных блоков.

Каждый блок рассматривается как отдельная задача, к которой применяется все, что вам уже известно. Когда одна часть приложения заработает так, как нужно, можно переходить к другому функциональному блоку. При этом на каждом шаге применяются те же принципы, о которых мы говорили последние 250 страниц или около того.

Большая задача может решаться посредством разбиения на функциональные блоки, после чего вы работаете над каждым блоком по отдельности.



То, что вы уже знаете...

Вам уже известно многое из того, что поможет вам в решении больших задач... Просто вы этого еще не осознали. Давайте кратко пройдемся по некоторым инструментам, которые помогают нам в написании хороших (и больших) программ:

Инкапсуляция того, что изменяется, делает приложение более гибким и упрощает его изменение.

Инкапсуляция также помогает и при решении больших задач. Чем выше уровень инкапсуляции, тем проще будет разбить большое приложение на функциональные блоки.

Программирование для интерфейса, а не для реализации, упрощает расширение программы.

Чтобы сформулировать хорошие требования, следует понять, как должна работать система.

Если вы знаете, что должен делать каждый маленький блок функциональности приложения, вы сможете легко объединить эти части в большое приложение, которое работает именно так, как было задумано.

В больших приложениях этот фактор еще важнее. Программирование для интерфейса снижает зависимости между разными частями приложения... а «слабая связность» — это всегда хорошо, помните?

Анализ поможет обеспечить работу вашей системы в реальном контексте.

Конечно, это положение остается истинным и для больших задач. Чем выше сцепление вашего приложения, тем менее функциональные блоки зависят друг от друга и тем проще работать над ними по отдельности.

Для больших программ анализ играет еще более важную роль... И в большинстве случаев следует начинать с анализа отдельных функциональных блоков, с последующим переходом к анализу взаимодействия между ними.

Хорошие программы легко расширяются и изменяются и работают так, как хочет заказчик.

Столкнулись с большой задачей? Вот вам несколько маленьких принципов. Уверяю, все проблемы как рукой снимет.

Так давайте решим БОЛЬШУЮ задачу!

Довольно говорить о том, что вы уже знаете; давайте посмотрим, как применить эти знания к новой большой программе. На следующей странице вы познакомитесь с Гэри, узнаете о его компании-разработчике игр и о крупном программном проекте.

Большая задача, над которой мы будем работать в нескольких ближайших главах.

Gary's Games

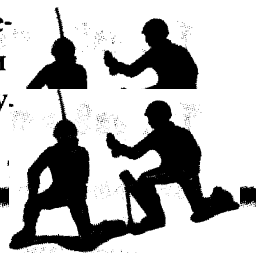
Общая концепция

Фирма Gary's Games занимается разработкой библиотек, используемых разработчиками для создания пошаговых стратегических игр. В отличие от аркадных «стрелялок» и игр, основанных на аудио- и видеоэффектах, наши игры ориентируются на стратегию и тактику. Наша библиотека берет на себя всю рутинную работу по созданию игры, избавляя разработчика от бремени программирования стандартных задач.

Библиотека GSF (Ggame System Framework) предоставляет основу для всех игр Gary's Games. Она представляет собой библиотеку классов с четко определенным API, которые будут использоваться всеми группами, занимающимися разработкой настольных игр. Библиотека предоставляет стандартные средства для выполнения следующих операций:

- Определение и представление конфигурации доски
- Определение родов войск и настройка армий или других боевых единиц
- Перемещение боевых единиц по доске
- Определение допустимых ходов
- Проведение сражений
- Предоставление информации о боевой единице

GSF упрощает разработку пошаговых стратегических настольных игр, чтобы пользователи GSF могли уделить больше времени игровому содержанию.



Меня не интересуют все эти однообразные поделки на темы «Звездных войн»... Хочу стратегии, чтобы нужно было думать! Пошаговая военная игра — вот наш козырь.

Это Гэри. Выглядит вполне серьезно, но при этом помешан на компьютерных «стратегиях».

Возьми в руку карандаш

С чего начать?

Ниже перечислены действия, с которых можно начать работу над проектом для Gary's Games. Установите флажки рядом с теми пунктами, с которых, по вашему мнению, следует начать.

- ☐ Поговорить с Гэри.
- ☐ Собрать требования.
- ☐ Создать диаграмму классов.
- ☐ Поговорить с людьми, которые будут использовать библиотеку.
- ☐ Написать вариант использования.
- ☐ Создать диаграмму пакетов.

Здесь все просто. Начинаем с записи требований и вариантов, как мы делали в случае с собачьими дверями.

Один из программистов вашей группы.

Требования и варианты использования — хорошая отправная точка...

Начать работу над системой с построения списка требований и написания вариантов использования – отличная мысль. Вы разберетесь в том, что должна делать система, и будете двигаться по списку, добавляя новую функциональность блок за блоком... Решая множество мелких задач для решения одной большой задачи.

Но я не уверен, что у нас хватает информации для формулировки требований или вариантов использования... Все, что у нас есть, — это общая концепция. А из нее непонятно, что должна делать система, которую нам предстоит построить.

...но что мы знаем о системе?

Документ довольно подробно описывает пожелания Гэри, но оставляет многовато места для интерпретаций.

Какие доски собирается использовать Гэри? И кто, собственно, будет пользователями – игроки или разработчики? Будут игры историческими или нам придется поддерживать лазеры, космические корабли и т. д.? Похоже, для написания хорошего набора требований нам еще нужно многое узнать.

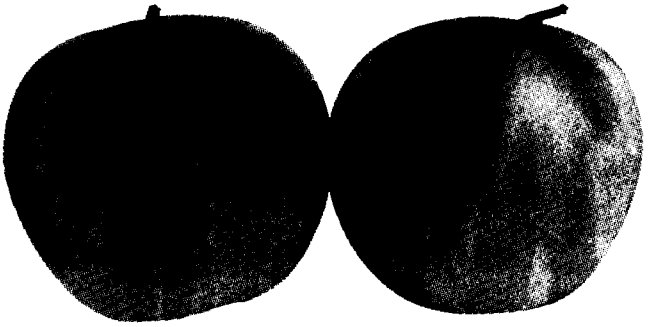
Необходимо больше информации

Пока что для работы над системой Гэри у нас нет ничего, кроме общей концепции... А она толком ничего не говорит о системе. Итак, теперь мы должны определить, как должна работать система. Как это сделать?

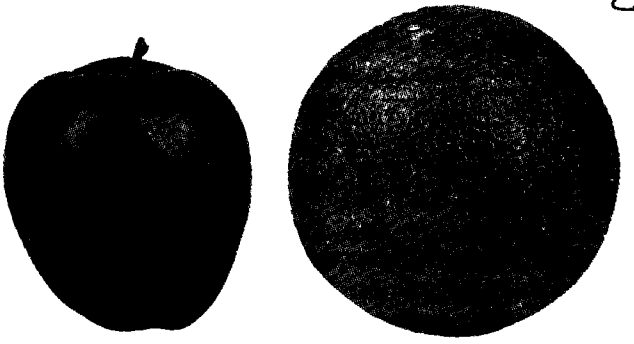
Это называется общностью... Какие аспекты системы похожи на другие?

На что похожа система?

Один из способов получения дополнительной информации о системе основан на определении того, на что похожи те или иные аспекты системы. Другими словами, известно ли вам что-нибудь о том, как должна функционировать система?



Это называется расхождением... Какие аспекты системы отличны от других?



На что не похожа система?

Другой способ определения того, как должна работать система, основан на идентификации ее отличий от остальных систем. Это поможет вам понять, о каких проблемах в системе можно не беспокоиться.

Давайте зайдем на одну из рабочих встреч в фирме и посмотрим, что удастся выяснить...

Разговоры заказчиков

Давайте немного послушаем, чего хотят Гэри и его группа, прежде чем браться за разработку игровой библиотеки, которую нам заказали.

Том: Да, Гэри любит текстовые игры. А народ уже устал от визуальных эффектов в играх типа «Звездные войны, эпизод 206» (или сколько их там).

Расхождение — система не будет обладать богатой графикой.

Бетани: И нам нужно охватить разные исторические периоды. В одной игре речь пойдет о Гражданской войне в Америке. В другой события будут происходить в Европе времен I Мировой войны... Могу поспорить, что игрокам понравится широкий исторический охват.

Гибкость — ключ к поддержке всех расхождений.

Сьюзен: Замечательная идея! И дать возможность разработчикам создавать дополнения, чтобы пользователь мог купить игру про II Мировую войну, а потом докупить войска, которых нет в основном комплекте.

Боб: Сильный маркетинговый ход! Если система будет поддерживать разные временные периоды, типы войск и все такое, мы сможем продать ее любому разработчику, занимающемуся играми.

Бетани: Как ты думаешь, нам стоит заниматься сражениями, не имеющими исторической основы? Я хочу сказать, что мы могли бы продать систему людям, которые занимаются научной фантастикой, чтобы они могли моделировать космические сражения?

Том: Хммм... Гэри ухватится за эту идею, если такие игры еще пользуются спросом. Почему бы не войти еще и на этот рынок, кроме рынка любителей исторических игр?

Боб: Думаешь, мы сможем представить свою библиотеку как систему для создания любых игр, от сетевой версии «Риска» до современного «Стратего»? Это были настоящие хиты в мире настольных игр... Я был бы рад продать нашу систему людям, разрабатывающим такие игры.

И еще немного общности... Итак, мы ориентируемся на походовые военные стратегии.

Бетани: Обсудим подробности. Мы знаем, что библиотека будет продаваться многим разработчикам, поэтому она должна быть по-настоящему гибкой. Думаю, мы начнем с квадратной доски, разбитой на клетки.

Том: И разработчик сможет изменять количество полей на доске, верно? Ширину-высоту, или что-нибудь в этом роде?

Бетани: Да, и еще мы должны поддерживать разные типы рельефа: горы, реки, равнины, луга...

Кажется, реальная функциональность игровой системы постепенно начинает проясняться.

Сьюзен: ...А также космос, кратеры, астероиды и все такое для космических игр...

Боб: И даже подводные локации — водоросли, ил...

Бетани: Отличная мысль! Следовательно, нам понадобятся классы для базового поля, которое может расширяться и настраиваться, и для доски, которая может заполняться разными полями.

Сьюзен: А как насчет перемещения и других правил, которые обычно присутствуют в играх?

Снова стратегические игры... Определенно здесь имеется сходство с этим типом игр, на которое стоит обратить внимание.

Том: Они тоже нужны, конечно. Ведь в стратегических играх сложные правила: юнит может продвинуться на столько-то полей, потому что несет слишком большой груз... и все такое.

Бетани: Однако я думаю, что большинство правил зависит от конкретной игры. Их нужно оставить на усмотрение разработчиков, которые будут использовать нашу библиотеку. А библиотека должна всего лишь следить за тем, чей ход, и обрабатывать простейшие перемещения юнитов.

Сьюзен: Мы построим библиотеку для интересных, увлекательных стратегических игр, а заодно неплохо заработаем.

Боб: Давайте поговорим с Гэри и программистами, которых он нанял, чтобы они могли взяться за дело.

Определение функциональных возможностей

Мы получили немало полезной информации о том, чего Гэри и его команда хотят от игровой библиотеки. Давайте воспользуемся этой информацией для определения ***функциональных возможностей*** системы.

Бетани сказала, что система должна поддерживать разные исторические периоды, — это относится к функциональным возможностям.

Бетани: И нам нужно охватить разные исторические периоды. В одной игре речь пойдет о Гражданской войне в Америке. В другой события будут происходить в Европе времен I Мировой войны... Могу поспорить, что игрокам понравится широкий исторический охват.

Другая функциональная возможность: разные типы рельефа. Вероятно, она преобразуется в несколько раздельных требований.

Бетани: Да, и еще мы должны поддерживать разные типы рельефа: горы, реки, равнины, луга...

Сьюзен: ...А также космос, кратеры, астероиды и все такое для космических игр...

Боб: И даже подводные локации — водоросли, ил...

Но что такое «функциональная возможность»?

Так называется *высокоуровневое описание* чего-то, что должна делать система. Как правило, функциональные возможности проясняются в ходе общения с заказчиками (или в ходе участия в их разговорах, как в нашем случае).

Довольно часто на базе одной функциональной возможности создается несколько требований, предназначенных для ее реализации. Таким образом, определение функциональных возможностей системы закладывает отличное начало для работы над требованиями.

Начинать с определения функциональных возможностей системы особенно полезно в больших проектах (таких, как игровая библиотека Гэри), когда вы не располагаете полной информацией о системе.

Функциональная возможность (от заказчика)

Поддержка разных типов рельефа

Отдельная функциональная возможность, сформулированная заказчиком.

Требование (для разработчика)

Поле связывается с определенным типом рельефа.

Разработчики игр могут создавать собственные типы рельефов.

Каждый рельеф обладает характеристиками, влияющими на перемещение юнитов.

Одна функциональная возможность преобразуется в набор требований.

Получите информацию о функциональных возможностях у заказчика, а затем определите требования, необходимые для их реализации.

Возьми в руку карандаш

Нам нужен список функциональных возможностей для игровой системы Гэри.

Мы получили достаточно информации от Гэри и его группы, и теперь вы знаете, как преобразовать эту информацию в набор функциональных требований. Запишите функциональные возможности, которые должна реализовать игровая библиотека.

Возьми в руку карандаш

Решение

Нам нужен список функциональных возможностей для игровой системы Гэри.

Мы получили достаточно информации от Гэри и его группы, и теперь вы знаете, как преобразовать эту информацию в набор функциональных требований. Запишите функциональные возможности, которые должна реализовать игровая библиотека.

Поддержка разных видов рельефа.

Поддержка разных типов войск или юнитов в зависимости от конкретной игры.

В каждой игре имеется доска из клеток. Каждому полю соответствует определенный тип рельефа.

Поддержка разных исторических периодов, а также научно-фантастического и фэнтезийного антуража.

Поддержка модулей расширения для дополнительных кампаний или вариантов.

Библиотека следит за ходами и координирует базовые перемещения.

Допустимо, если это не точные формулировки, предоставленные заказчиком, или в списке имеются более подробные описания.

Пока все выглядит весьма условно... Некоторые функциональные возможности очень похожи на требования. Не все ли равно, как их назвать — функциональной возможностью или требованием?

Не зацикливайтесь на «различиях» между функциональной возможностью и требованием.

«Функциональной возможностью» часто называют совершенно разные вещи, поэтому не стоит слишком полагаться на него. Для некоторых людей функциональная возможность и требование — *одно и то же.* Иногда даже встречается выражение «функциональное требование», что может создать изрядную путаницу.

Другие рассматривают функциональные возможности как концепции более высокого уровня, чем требования, — так, как это делаем и мы. Таким образом, для реализации одной функциональной возможности может понадобиться несколько требований.

Важно другое: если вы не знаете, с чего начать работу (особенно над большим проектом), займитесь сбором описаний функциональных возможностей (или требований!), чтобы получить представление о высокоуровневых аспектах, которые должны быть реализованы в создаваемой вами системе.

Часто задаваемые вопросы

В: Выходит, четких различий между функциональной возможностью и требованием нет?

О: Вообще-то это зависит от того, кого вы спрашиваете. Некоторые считают, что функциональная возможность — это нечто «большое», что делает система (например, «поддержка разных типов рельефа»). Но для ее реализации система должна делать много мелких подзадач, скажем, «определить базовый тип рельефа», «позволить разработчику расширять базовый тип рельефа» и «обеспечить возможность представления разных типов рельефа полями». Все эти подзадачи считаются требованиями. Таким образом, одна функциональная возможность обслуживается несколькими требованиями.

Функциональные возможности — высокоуровневые аспекты, реализация которых обеспечивается совокупностью нескольких требований.

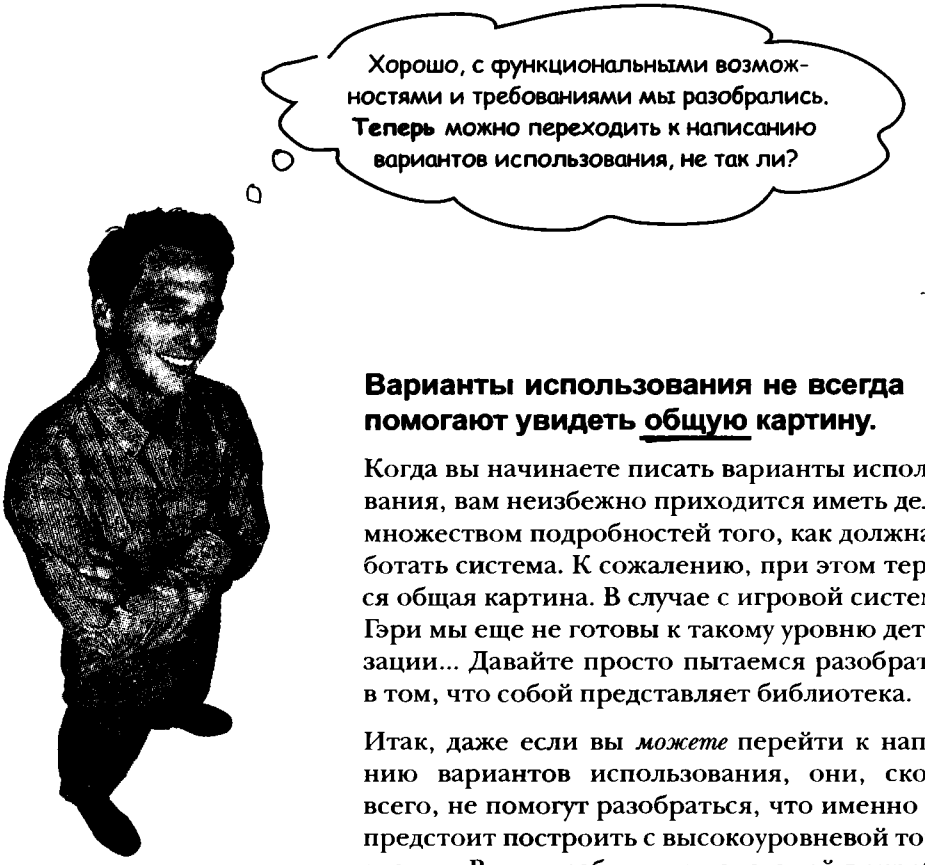
В: Вы сказали «некоторые считают». Значит, существуют разные подходы к определению функциональных возможностей и требований?

О: Правильно. Многие специалисты не различают требования и функциональные возможности: например, в систему включаются функциональные возможности «поддержка разных исторических периодов» (масштабная задача) и «поддержка воды как возможного типа рельефа» (относительно небольшая и конкретная задача). При таком подходе принципиальных различий между требованиями и функциональными возможностями нет, а отношения между ними выглядят так:

При таком подходе понятия функциональной возможности и требования отчасти перекрываются. Эти два термина в значительной степени становятся синонимами.

В: И какой подход является правильным?

О: Оба! Или ни одного, если хотите. Не существует «единственно правильного» подхода, особенно если вы не хотите потратить много времени на обсуждение определений со своими коллегами-программистами. Лучше рассматривать функциональные возможности и требования как некие аспекты, которые должны быть реализованы системой. Если вы предпочитаете рассматривать функциональные возможности как «элементы общей картины», а требования — как «низкоуровневые подробности» — это нормально... Только не ссорьтесь с коллегами по этому поводу, хорошо?



Варианты использования не всегда помогают увидеть общую картину.

Когда вы начинаете писать варианты использования, вам неизбежно приходится иметь дело с множеством подробностей того, как должна работать система. К сожалению, при этом теряется общая картина. В случае с игровой системой Гэри мы еще не готовы к такому уровню детализации... Давайте просто пытаемся разобраться в том, что собой представляет библиотека.

Итак, даже если вы *можете* перейти к написанию вариантов использования, они, скорее всего, не помогут разобраться, что именно вам предстоит построить с высокоуровневой точки зрения. В ходе работы над системой подробности всегда следует откладывать настолько, насколько это возможно... Нельзя копаться в *мелочах*, когда нужно работать *на высоком уровне.*

Всегда откладывайте определение подробностей настолько, насколько возможно.

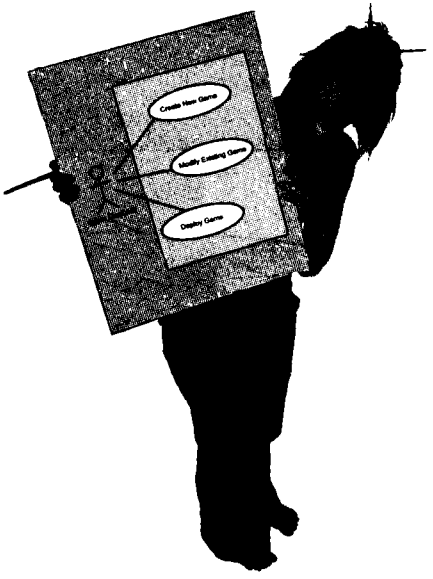
МОЗГОВОЙ ШТУРМ

Если мы напишем варианты использования для игровой системы Гэри, кем будут субъекты этих сценариев?

И что сейчас **нужно** делать? Вы уже страниц 200 говорите, что нужно узнать, как должна работать система, и вдруг выясняется, что варианты использования писать не нужно? Что происходит?

Необходимо знать, как должна работать система... но нужна ОБЩАЯ картина.

Хотя на данной стадии проектирования системы варианты использования слишком сосредоточены на подробностях, хорошее понимание работы системы все равно необходимо. Следовательно, мы должны каким-то образом представить себе общую картину и понять, как должна работать система, не углубляясь в излишние подробности.



Знаете поговорку: «Лучше один раз увидеть, чем сто раз услышать»?

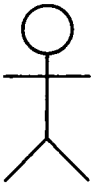
Давайте попробуем представить работу системы в графическом виде.

Используем диаграммы вариантов

Иногда нам бывает нужно понять, как должна работать система, не углубляясь в подробности, необходимые для вариантов использования. В таких ситуациях часто выручают диаграммы вариантов использования.

Большой прямоугольник представляет систему. Внутри прямоугольника находится сама система, снаружи — пользователи. Таким образом, прямоугольник обозначает границы системы.

Этот человечек — субъект. Он выполняет операции с системой, которой в данном случае является игровая библиотека.



Разработчик игры

Помните: субъектом в этой системе является разработчик, а не игрок.

Создание новой игры

Изменение существующей игры

Установка игры

Каждый овал представляет отдельный вариант использования системы.

Возможно, такая диаграмма вариантов использования не похожа на подробный набор спецификаций, но она показывает все, что должна делать система, в простом и удобочитаемом формате. Варианты использования более детализированы и не помогают представить общую картину так же хорошо, как диаграмма вариантов использования.

По-моему, это просто глупо. Какой прок от этой диаграммы? Нам действительно нужно рисовать картинку, чтобы понять, что разработчики игр будут создавать и изменять игры?

Диаграмма вариантов использования — план системы.

Не забудьте, что сейчас нас интересует ***общая картина***. Диаграмма вариантов использования может показаться слишком общей, но она помогает сосредоточиться на фундаментальных вещах, которые *должна* делать ваша система. Без нее можно легко увязнуть в подробностях того, как разработчик будет создавать новую игру, и полностью забыть о том, что игру нужно будет *установить*. С диаграммой вариантов использования вы никогда не забудете общую картину.

А как же функциональные возможности, над которыми мы так усердно трудились? Они даже не представлены на диаграмме вариантов использования!

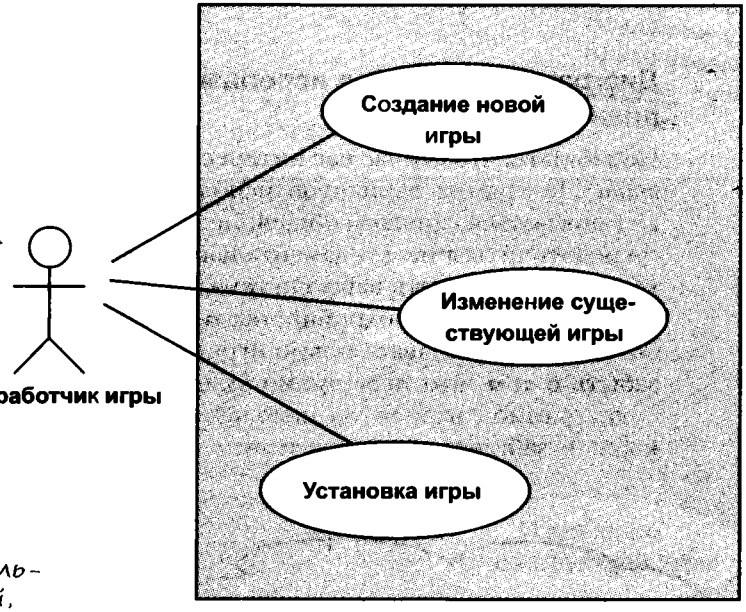
Используйте список функциональных возможностей для проверки полноты диаграммы вариантов использования.

Располагая списком функциональных возможностей и диаграммой вариантов использования, можно проследить за тем, чтобы создаваемая система делала все необходимое. Возьмите диаграмму вариантов использования и убедитесь в том, что перечисленные варианты охватывают все функциональные возможности, о которых вам сообщил заказчик. Когда вы будете уверены в том, что диаграмма (план системы) полна, приступайте к построению системы.

Развлечения с магнитами

Пришло время связать функциональные возможности игровой библиотеки с вариантами использования на нашей диаграмме. Поместите магнит каждой функциональной возможности на вариант использования, который будет обеспечивать ее реализацию. Если диаграмма вариантов использования полна, вы сможете связать каждый магнит с вариантом на диаграмме. Удачи!

Диаграмма вариантов использования, наш план системы.



Каждая функциональная возможность должна быть закреплена за одним из вариантов использования в системе.

Список функциональных возможностей, построенный нами на с. 318.

Игровая библиотека

Список функциональных возможно

1. Поддержка разных типов рельефа.
2. Поддержка разных исторических периодов, а также фантастического и фэнтезийного антуража.
3. Поддержка разных типов войск в зависимости от конкретной игры.
4. Поддержка модулей расширения для дополнительных кампаний и вариантов.
5. Доска состоит из клетчатого поля, каждой клетке присваивается тип рельефа.
6. Библиотека следит за очередностью хода и координирует базовые перемещения.

Поддержка разных типов рельефа.

Поддержка разных исторических периодов.

Поддержка разных типов войск.

Поддержка модулей расширения.

Доска состоит из полей, каждое из которых имеет определенный тип рельефа.

Библиотека следит за очередностью хода и координирует базовые перемещения.

Магниты функциональных возможностей.

Часто задаваемые вопросы

В: Субъект — это человек, использующий систему?

О: Субъект представляет собой любую внешнюю сущность, взаимодействующую с системой (не обязательно человека). Скажем, для банкомата человек, использующий систему, очевидно является субъектом, но наряду с этим субъектом также может быть банк, который кладет деньги в банкомат. Банк не является частью системы, но работает с ней, следовательно, он является субъектом.

В: Для чего нужно рисовать прямоугольник? И почему субъекты находятся вне его?

О: Прямоугольник обозначает границы системы. Вы должны запрограммировать все, что находится внутри прямоугольника, но о том, что находится снаружи, беспокоиться не нужно. Субъекты — разработчики игр, использующие вашу библиотеку, — находятся снаружи, они используют систему, но не являются ее частью.

В: И каждый кружок — это вариант использования?

О: Да. Вот почему диаграммы вариантов использования так хорошо подходят для представления общей картины. На них можно представить несколько вариантов использования и их взамодействие при выполнении высокоуровневых операций. Диаграмма также помогает избежать слишком ранней привязки к подробностям конкретных требований (на той стадии, когда нужно думать об общей структуре кода системы).

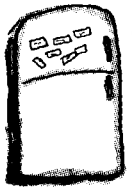
В: Я видел диаграммы вариантов использования, на которых линии были снабжены пометками <<include>> и <<extend>>. Что это?

О: UML и диаграммы вариантов использования определяют различные способы представления связей между вариантами использования. Можно сказать, что один вариант использования включает другой или расширяет его. Для этих целей используются ключевые слова **<<include>>** и **<<extend>>**.

Однако за спорами о том, включает ли один вариант использования другой, расширяет ли он его и т. д., можно потратить много времени. И внезапно вы начинаете обсуждать, как организовать поддержку горного рельефа или рюкзаков для дополнительного груза, вместо того чтобы сосредоточиться на общей картине. Вы можете использовать обозначения <<include>> и <<extend>>, но в действительности они не так уж важны. Во всяком случае, они не должны отвлекать вас от общего процесса проектирования.

В: Итак, диаграммы вариантов использования должны представлять общую картину системы вместо большого количества мелких подробностей?

О: В точку! Если вы начинаете беспокоиться о том, как следует назвать вариант использования и не надо ли использовать определенный тип связей между вариантами, вы теряете общую картину. Вариант использования должен давать наглядное высокоуровневое представление системы, ничего более.

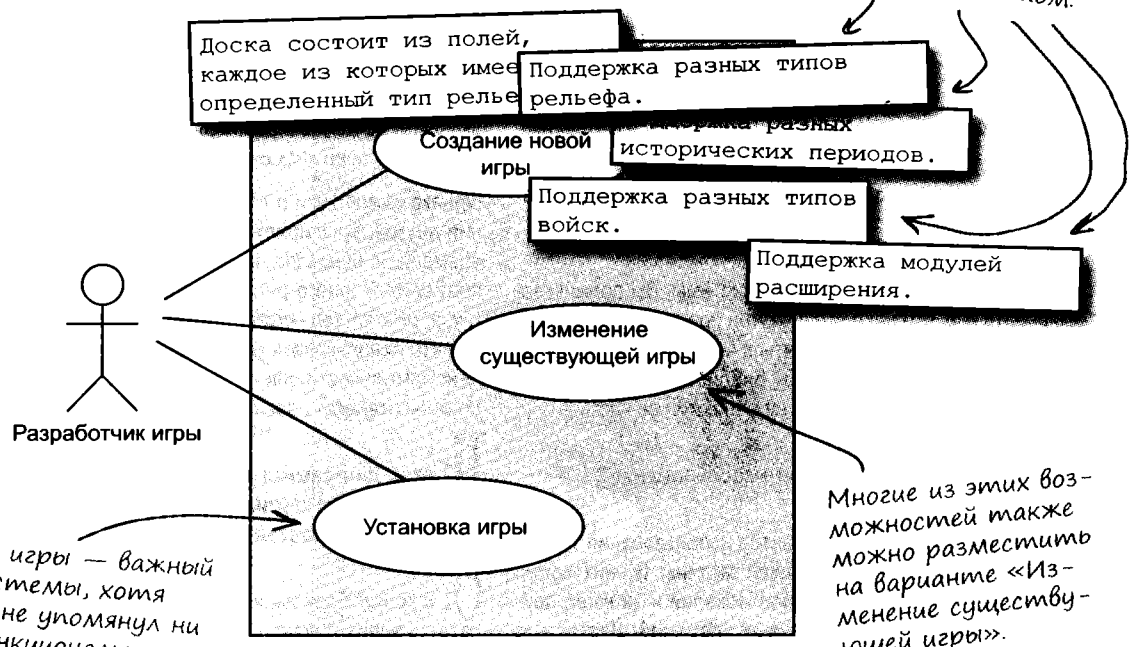


Развлечения с магнитами. Решение

На этот раз решение получилось немного неполным...

Пришло время связать функциональные возможности игровой библиотеки с вариантами использования на нашей диаграмме. Удалось ли вам найти вариант использования для каждой функциональной возможности в списке?

Почти все функциональные возможности относятся к созданию новой игры разработчиком.



Многие из этих возможностей также можно разместить на варианте «Изменение существующей игры».

Установка игры — важный аспект системы, хотя заказчик и не упомянул ни о каких функциональных возможностях, относящихся к ней.

Одна функциональная возможность осталась нераспределенной... Что с ней делать?

Библиотека следит за очередностью хода и координирует базовые перемещения.

Одну из функциональных возможностей трудно разместить на диаграмме вариантов использования. Это не то, с чем напрямую взаимодействует разработчик игры, поскольку данная функциональность уже реализована.

Как же эта функциональная возможность связана с системой? Какие субъекты задействованы? Не упустили ли мы чего-то на диаграмме?

Как вы думаете?

Почему-то этой функциональной возможности не нашлось места в нашем плане системы?

Маленький субъект

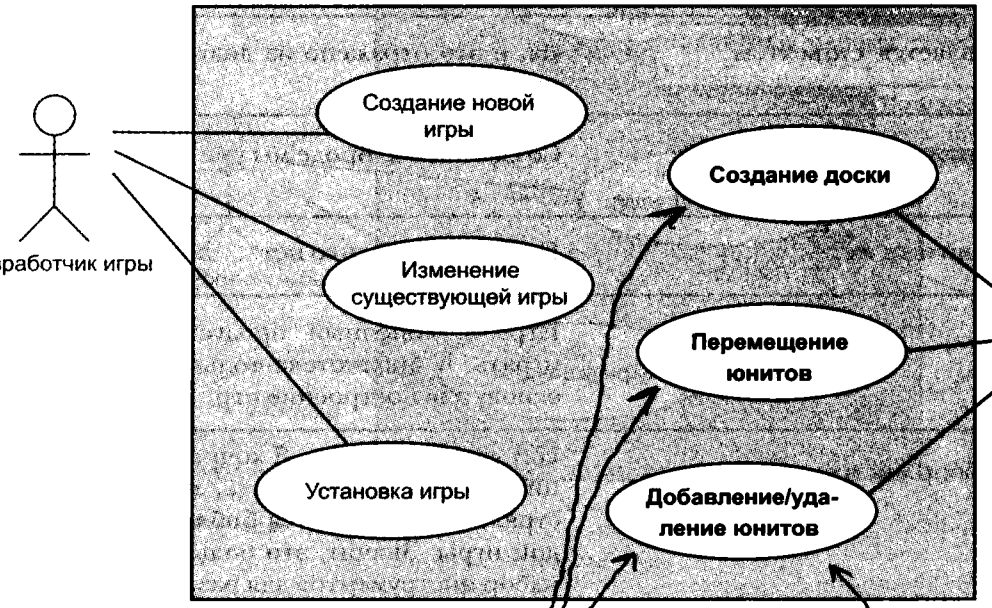
Небольшой сократовский диалог

Какую системы ты проектируешь?	Игровую библиотеку, конечно!
Для чего нужна эта библиотека?	Чтобы помочь разработчикам создавать игры.
Значит, разработчик является субъектом в системе?	Да, и это отражено на диаграмме вариантов использования.
И что разработчик игры делает с библиотекой?	Создает игры. Вроде мы уже это выяснили.
Игра и библиотека – одно и то же?	Нет... Полагаю, что нет.
Почему?	Игра – конечный продукт, в нее можно играть. А библиотека только предоставляет основу для построения игр.
Значит, библиотека – набор инструментов для разработчиков игр?	Нет, нечто большее. Я хочу сказать, что функциональная возможность, на которой я застрял, обеспечивается библиотекой для каждой игры. Значит, это больше, чем простой набор инструментов для разработчика.
Интересно. Тогда получается, что библиотека – часть игры?	Вероятно. Но она работает на низком уровне, предоставляя игре некий базовый сервис. Сама игра в каком-то смысле находится над библиотекой.
Значит, игра использует библиотеку?	Да, безусловно.
А следовательно, игра использует систему, которую вы строите?	Верно, я же только что сказал. Погодите... Но тогда получается, что...
...Если игра использует систему, то чем она является?	Субъектом! Игра в нашем примере – это субъект!

Субъекты тоже люди (но не всегда)

Оказывается, наряду с разработчиком сама игра также является субъектом для создаваемой нами библиотеки. Давайте посмотрим, как включить нового субъекта в диаграмму вариантов использования:

Мы добавили нового субъекта для игры (которую создает разработчик с использованием нашей библиотеки).



Некоторые задачи, для выполнения которых игра использует библиотеку.

Дополнительные варианты использования, которые должны быть реализованы нашей системой для полноты.

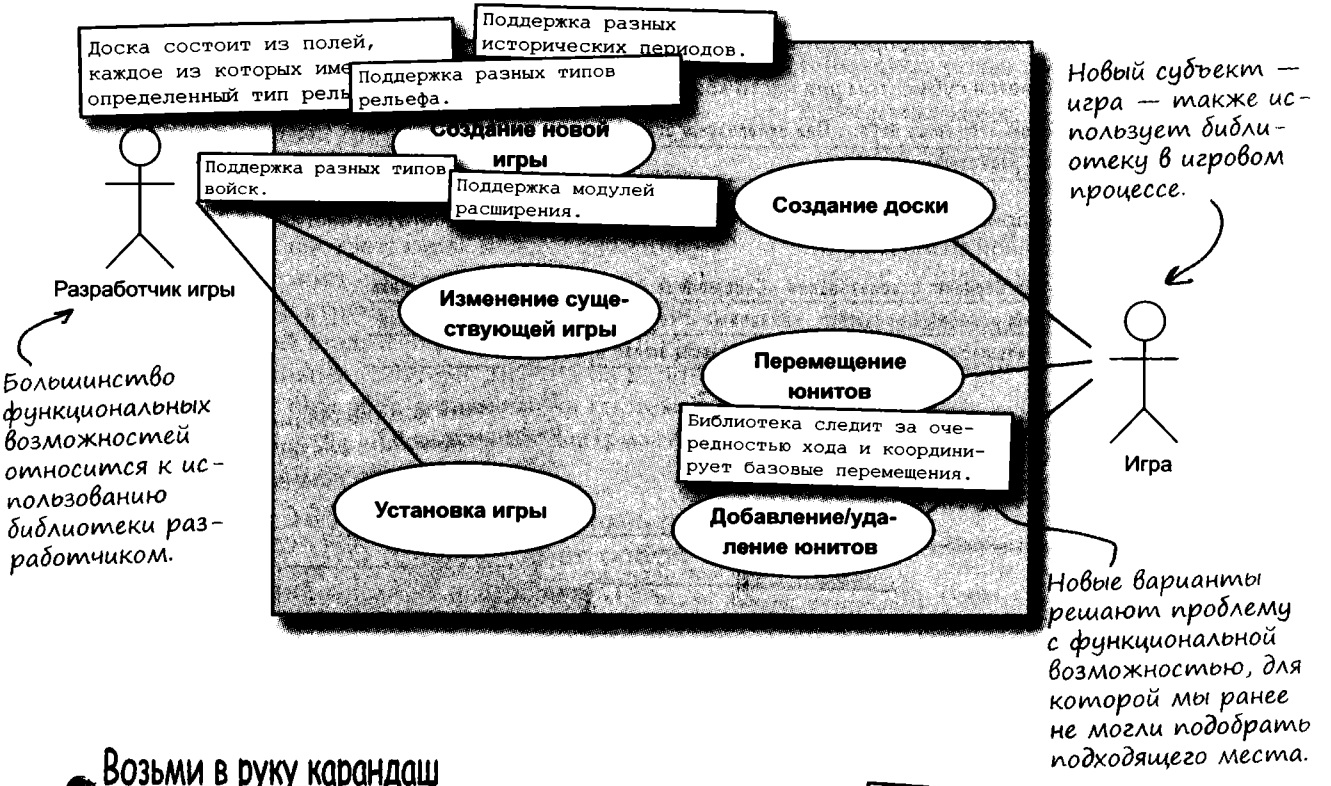
Помните: субъекты не обязаны быть людьми... На этот раз с нашей системой взаимодействует сама игра.

Решают ли новые варианты использования проблему с функциональной возможностью, для которой мы не могли найти подходящего места?

Библиотека следит за очередностью хода и координирует базовые перемещения.

Диаграмма вариантов использования... есть! Функциональные возможности... распределены!

С новым субъектом мы наконец-то сможем окончательно связать функциональные возможности с диаграммами вариантов использования.



Возьми в руку карандаш

И все-таки с последней функциональной возможностью что-то не так...

Библиотека следит за очередностью хода и координирует базовые перемещения.

Вторая часть — о перемещении — подходит к варианту использования «Перемещение юнитов»... Но как насчет очередности хода? Похоже, на нашей диаграмме снова чего-то не хватает. Попробуйте ответить на два вопроса:

1. Кто является субъектом для части «Библиотека следит за очередностью хода»?

_____

2. Какой вариант использования вы бы добавили для поддержки этой частичной функциональной возможности?

_____

ДОПОЛНИТЕЛЬНО: внесите эти изменения в приведенную выше диаграмму вариантов использования.

Возьми в руку карандаш
Решение

Библиотека следит за очередностью хода и координирует базовые перемещения.

И все-таки с последней функциональной возможностью что-то не так...

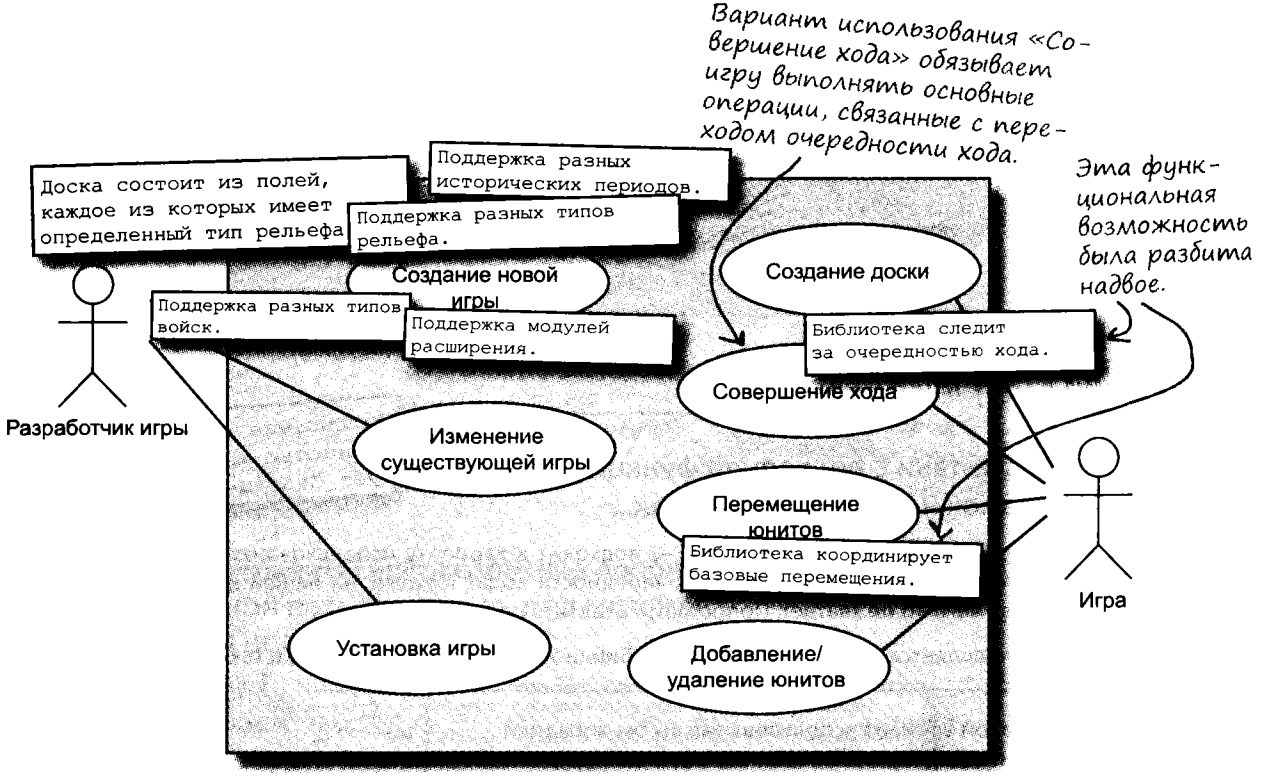
Вторая часть — о перемещении — подходит к варианту использования «Перемещение юнитов»... Но как насчет очередности хода? Похоже, на нашей диаграмме снова чего-то не хватает. Попробуйте ответить на два вопроса:

1. Кто является субъектом для части «Библиотека следит за очередностью хода»?

   Субъектом остается игра... Она использует библиотеку для управления очередностью хода.

2. Какой вариант использования вы бы добавили для поддержки этой частичной функциональной возможности?

   Нам нужен вариант использования «Совершение хода», в котором библиотека выполняет основные действия, связанные с ходом, и позволяет расширению игры обработать специфические особенности этого процесса.



Итак, что же именно мы сделали?

Мы написали список функциональных возможностей, которые должны поддерживаться игровой библиотекой Гэри. Он определяет все высокоуровневые блоки системы, которую нам предстоит создать. Процедура во многом напоминает список требований, построенный в главе 2 для собачьей двери Тодда и Джины... не считая того, что она направлена на формирование общей картины.

Список функциональных возможностей... То, что должна делать наша система.

Игровая библиотека

Список функциональных возможностей

1. Поддержка разных типов рельефа.
2. Поддержка разных исторических периодов, а также фантастического и фэнтезийного антуража.
3. Поддержка разных типов войск в зависимости от конкретной игры.
4. Поддержка модулей расширения для дополнительных кампаний и вариантов.
5. Доска состоит из клетчатого поля, каждой клетке присваивается тип рельефа.
6. Библиотека следит за очередностью хода.
7. Библиотека координирует базовые перемещения.

Используйте список функциональных возможностей или требований для определения ВЫСОКОУРОВНЕВЫХ ЗАДАЧ, которые должна решать ваша система.

После установления соответствия между функциональными возможностями и требованиями необходимо представить себе структуру системы. Варианты использования на этой стадии часто оказываются излишне детализированными, поэтому диаграмма вариантов использования поможет взглянуть на систему «с высоты птичьего полета»... Ее можно рассматривать как своего рода общий план приложения.

Нарисуйте диаграмму вариантов использования. Она покажет, ЧТО представляет собой ваша система, но без лишних подробностей.

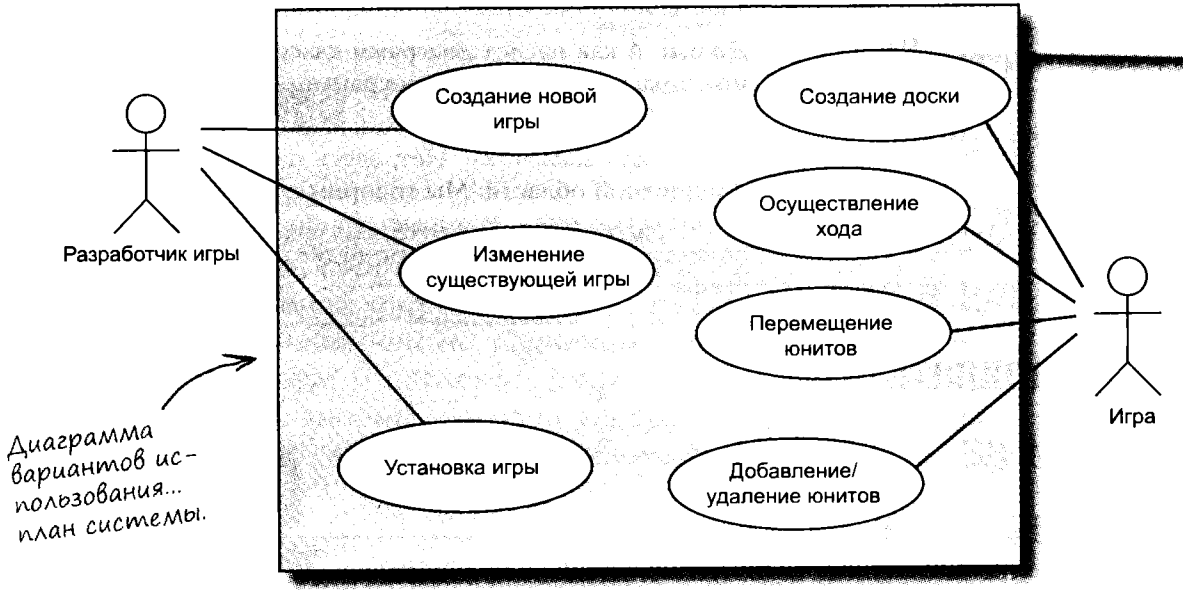


Диаграмма вариантов использования... план системы.

Разговор в офисе

Фрэнк: Не знаю, Джим. Мне кажется, что мы все это время говорили о коде.

Джим: Как это понимать? Я хочу сказать, в какую строку кода преобразуется «поддержка разных типов рельефа»?

Фрэнк: Мы говорим о тех функциональных возможностях, которые мы определили, верно? Это будет не одна строка, а очень большой блок кода.

Джим: Конечно... Но когда мы начнем обсуждать классы, которые нам предстоит написать, и пакеты, в которых эти классы будут объединены?

Фрэнк: Безусловно, дойдет и до этого. Но заказчик не понимает, что это такое... Мы никогда не будем уверены в том, что строим правильную систему, если начнем обсуждать классы и переменные.

Джим: А как насчет диаграмм классов? Мы ведь можем на них показать, что будем программировать?

Фрэнк: Можем... Но ты полагаешь, они будут намного понятнее для заказчика? Нет, здесь нам понадобится анализ предметной области. Мы говорим с заказчиком о его системе в выражениях, которые ему понятны. Вместо классов, объектов и методов для Гэри будут поля, юниты и виды рельефа.

Анализ предметной области позволяет проверить результаты проектирования и при этом разговаривать с заказчиком на понятном ему языке.

Займемся анализом предметной области!

Давайте соберем воедино все, что мы узнали об игровой системе, притом так, чтобы информация была понятна Гэри, нашему заказчику. Этот процесс называется ***анализом предметной области***. Этим термином обозначается описание задачи в выражениях, понятных заказчику.

Предметную область в данном случае составляют игровые системы.

Функциональные возможности определяются в выражениях, понятных заказчику.

Игровая библиотека

Список функциональных возможностей

1. Поддержка разных типов рельефа.
2. Поддержка разных исторических периодов, а также фантастического и фэнтезийного антуража.
3. Поддержка разных типов войск в зависимости от конкретной игры.
4. Поддержка модулей расширения для дополнительных кампаний и вариантов.
5. Доска состоит из клетчатого поля, каждой клетке присваивается тип рельефа.
6. Библиотека следит за очередностью хода.
7. Библиотека координирует базовые перемещения.

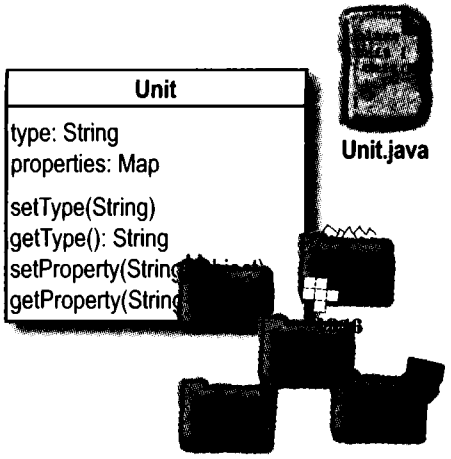
1946

Весь список возможностей также является разновидностью анализа (наподобие того, что мы делали в предшествующих главах).

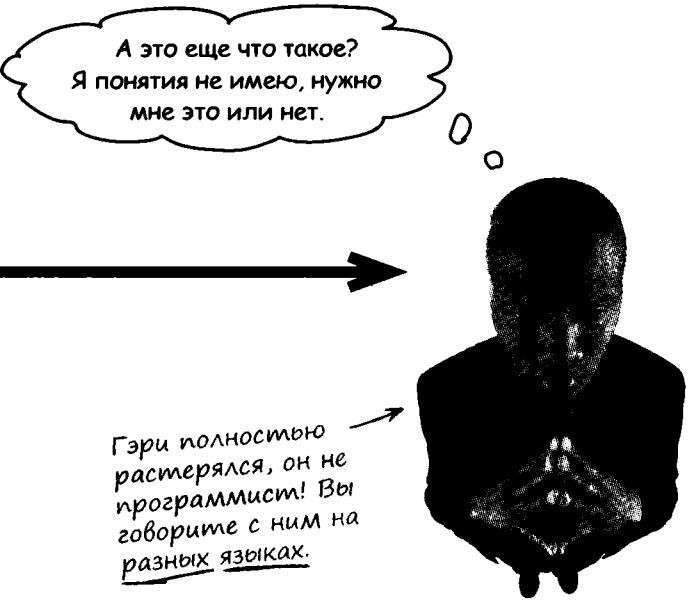
За партой

Анализ предметной области. Процесс идентификации, сбора, упорядочения и представления актуальной информации о предметной области, основанной на изучении существующих систем и истории их развития, экспертных знаний, базовой теории и уровня технологического развития.

Что обычно дают заказчику...

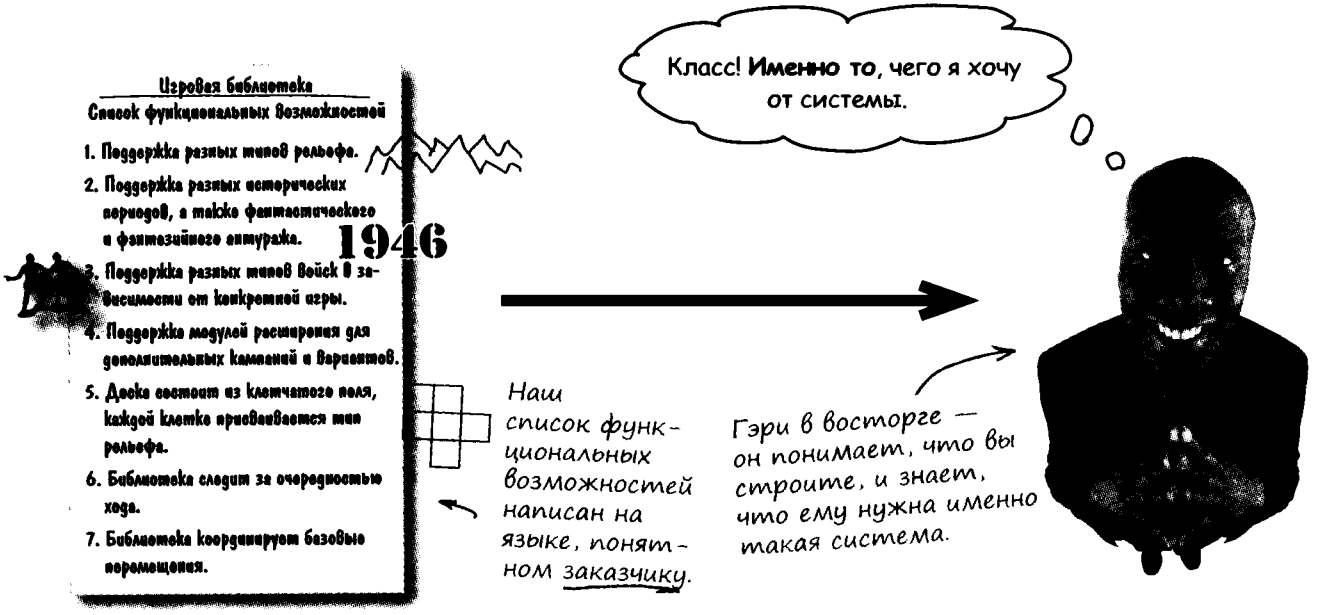


Диаграммы классов и пакетов, информация о построении игровой системы Гэри на уровне программного кода.



Гэри полностью растерялся, он не программист! Вы говорите с ним на разных языках.

Что мы даем заказчику...



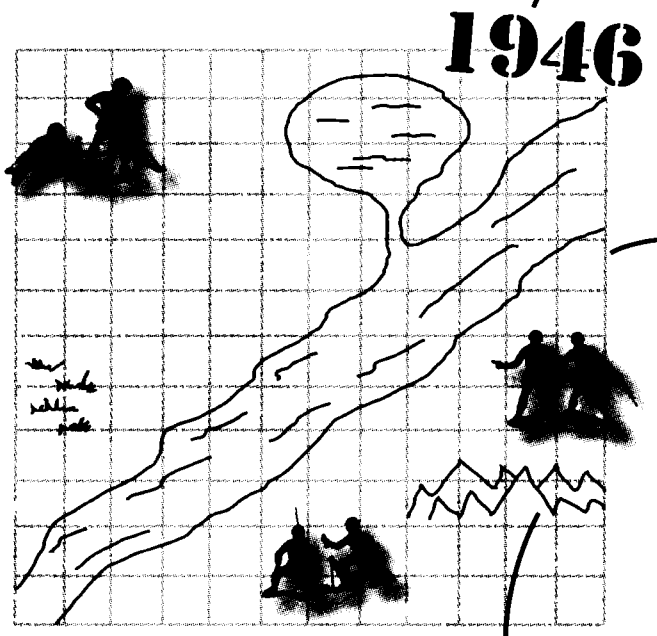
Наш список функциональных возможностей написан на языке, понятном заказчику.

Гэри в восторге — он понимает, что вы строите, и знает, что ему нужна именно такая система.

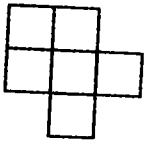
Разделяй и Властвуй

Когда вы наладите взаимодействие с заказчиком и будете располагать хорошим, полным набором планов системы, можно переходить к разбиению большой задачи на меньшие функциональные блоки и потом использовать то, что вы узнали, для последовательного решения меньших задач.

Примерная схема основных частей игровой системы.



Исторические периоды
Здесь сложной работы не ожидается. Если обеспечить поддержку разных видов рельефа, типов войск и вооружения, эта функциональная возможность реализуется естественным образом.



Поля
В библиотеке должен быть определен базовый тип поля, и каждое поле должно поддерживать тип рельефа и располагающиеся на нем юниты и, вероятно, также обеспечивать проведение сражений.

Юниты
Нам понадобится механизм представления базового юнита (боевой единицы). Разработчики игр должны иметь возможность его расширения для создания юнитов под конкретную игру.

Большая библиотека разбивается на несколько меньших, более удобных для управления блоков.

Типы рельефа
Каждое поле должно поддерживать не менее одного типа рельефа. Разработчики игр должны создавать и использовать собственные рельефы — от зеленых лугов до метеоритных кратеров.

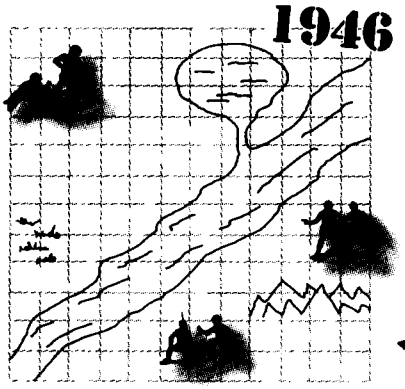
Время разбивать

Пора разбить нашу большую задачу — игровую библиотеку — на несколько меньших функциональных блоков. Вы уже видели, как игра и ее функциональные возможности делятся на функциональные группы, так что часть пути уже пройдена.

Ниже представлены функциональные возможности и диаграммы, которые использовались нами в этой главе для представления того, что должна делать система Гэри. Посмотрите на них и подумайте, какие модели понадобятся для реализации всей этой функциональности и как организовать разбиение функциональных возможностей и требований. Проверьте, чтобы в модулях было учтено все, что, по вашему мнению, должна делать игровая библиотека!

Игровая библиотека
Список функциональных возможностей

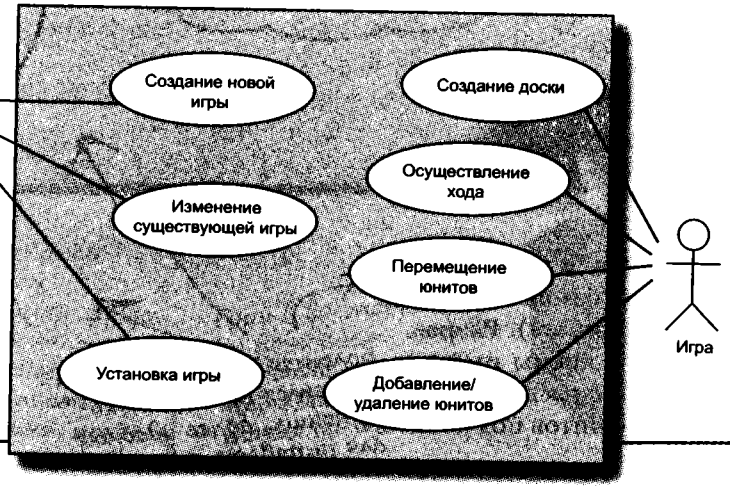
1. Поддержка разных типов рельефа.
2. Поддержка разных исторических периодов, а также фантастического и фэнтезийного антуража.
3. Поддержка разных типов войск в зависимости от конкретной игры.
4. Поддержка модулей расширения для дополнительных кампаний и вариантов.
5. Доска состоит из клетчатого поля, каждой клетке присваивается тип рельефа..
6. Библиотека следит за очередностью хода.
7. Библиотека координирует базовые перемещения.



Игровая доска напоминает о важных областях, на которых стоит сосредоточиться... Но помните: это еще не все!

Нужно реализовать все функциональные возможности системы...

...а также варианты использования, представленные на диаграмме.



Мы добавили модуль Units, чтобы вам было проще начать. В этом модуле объединяются классы, представляющие войска, армии и связанную с ними функциональность.

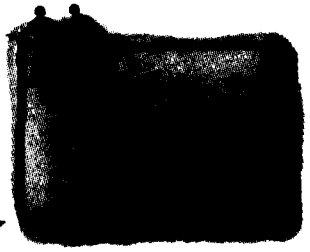
Для каждого пакета/модуля напишите, на чем, по вашему мнению, должен специализироваться этот модуль.

При необходимости вы можете добавить дополнительные модули или, наоборот, использовать меньше модулей, чем мы изобразили. Решайте сами!

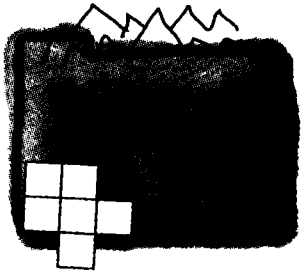
Я работаю с **БОЛЬШИМИ** задачами, и мне они не под силу. Пора разделить их на задачи поменьше.

Время разбивать

Ниже представлен наш вариант разбиения всех функциональных возможностей системы Гэри на несколько меньших, более удобных в управлении функциональных блоков.

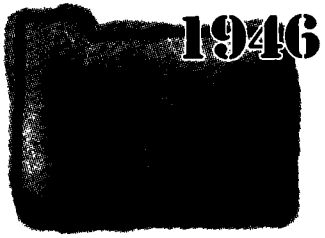


Войска, армии и другие юниты, используемые в игре.

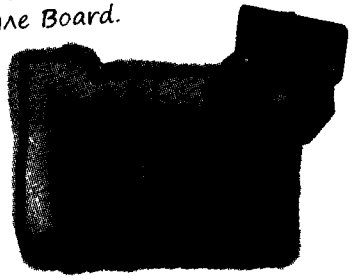


Модуль Board работает с доской, полями, юнитами, рельефом и другими классами, относящимися к созданию доски в каждой игре.

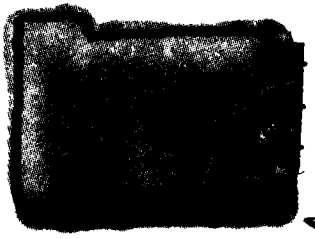
Мы НЕ СТАЛИ создавать модуль только для рельефа или полей, потому что эти модули будут содержать всего один-два класса. Вместо этого мы объединили их в модуле Board.



Модуль Game используется для хранения базовых классов, которые могут расширяться разработчиками игр. Они относятся к историческому периоду игры, основным свойствам и вообще всему, что определяет базовую структуру игры.



Необходимо предусмотреть служебный модуль Utilities для хранения вспомогательных классов, совместно используемых другими модулями.



Модуль обрабатывает ходы каждого игрока, базовые перемещения и все остальное, что относится к фактическому ходу игры. Это своего рода координатор для игр, создаваемых разработчиками.

РАССЛАБЬТЕСЬ

На это упражнение нет ЕДИНСТВЕННО ПРАВИЛЬНОГО ответа!

Не огорчайтесь, если ваши ответы совпали с нашими не полностью. Существует много способов структурирования систем, мы выбрали лишь один из возможных вариантов. Важно, чтобы ваше решение охватывало все функциональные возможности и варианты использования и было разумным... Модули не должны содержать всего один класс или стодвести классов.

Да эти игры будут полным **БАРАХЛОМ**! У них даже нет графического пакета... Без визуальных эффектов можно обойтись, но я должен хотя бы видеть доску с юнитами.

Тони много знает о видеоиграх, но он — не ваш клиент!

Не забывайте, кто ваш заказчик

Может показаться, что Тони говорит логичные вещи... Пока вы не вспомните, кто будет пользоваться игровой библиотекой Гэри. Вы пишете библиотеку для разработчиков, а не создаете игры. В этих играх будет разный пользовательский интерфейс, и графикой должен заниматься разработчик игры, а не вы.

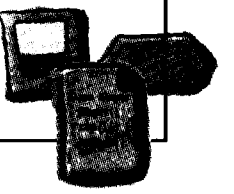
Анализ предметной области помогает избежать построения частей системы, которыми вам заниматься не положено.

Графикой будут заниматься разработчики игр... Это не ваша область ответственности.

Подсказка для читателей книги «Паттерны проектирования».

ГОЛОВОЛОМКА

Присмотритесь повнимательнее к модулям и группам в игровой библиотеке. Вы не узнаете в них распространенный паттерн проектирования?



После того как разработчик игры добавит модуль Graphics, это будет очень похоже на паттерн «Модель-Представление-Контроллер».

Да это же паттерн «Модель-Представление-Контроллер»!

Обычно программисты используют сокращенное обозначение MVC (от Model-View-Controller).

Контроллер игры, который мы собираемся написать. Он обеспечивает выполнение базовых ходов и определяет, что происходит с доской, юнитами и т. д.

Модули Board и Units являются частью модели... Они моделируют фактические события, происходящие в игре.

Представление

Разработчик игры обеспечивает работу представления и добавляет графику для отображения состояния модели, чтобы игроки видели что-то на экране.

Контроллер

модель оповещает представление об изменении состояния

Модель

контроллер выполняет операции с моделью

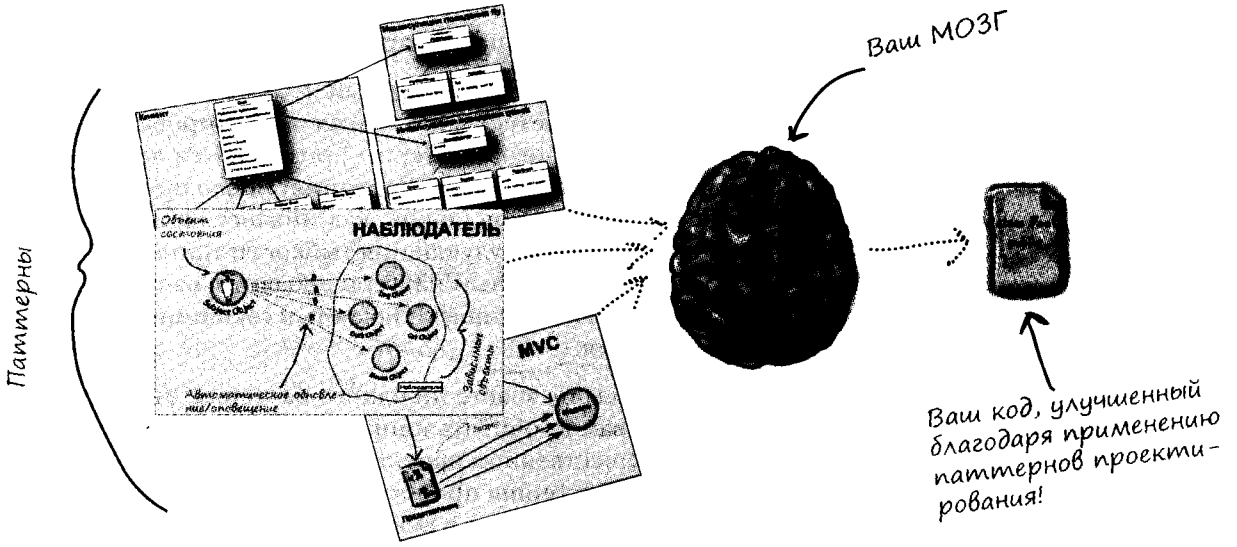
Разработчики игр могут расширять эти модули собственными контроллерами, предназначенными для конкретных игр, но этот модуль остается основным контроллером игры.

Эти модули не входят в паттерн MVC, но они все равно являются частью системы.

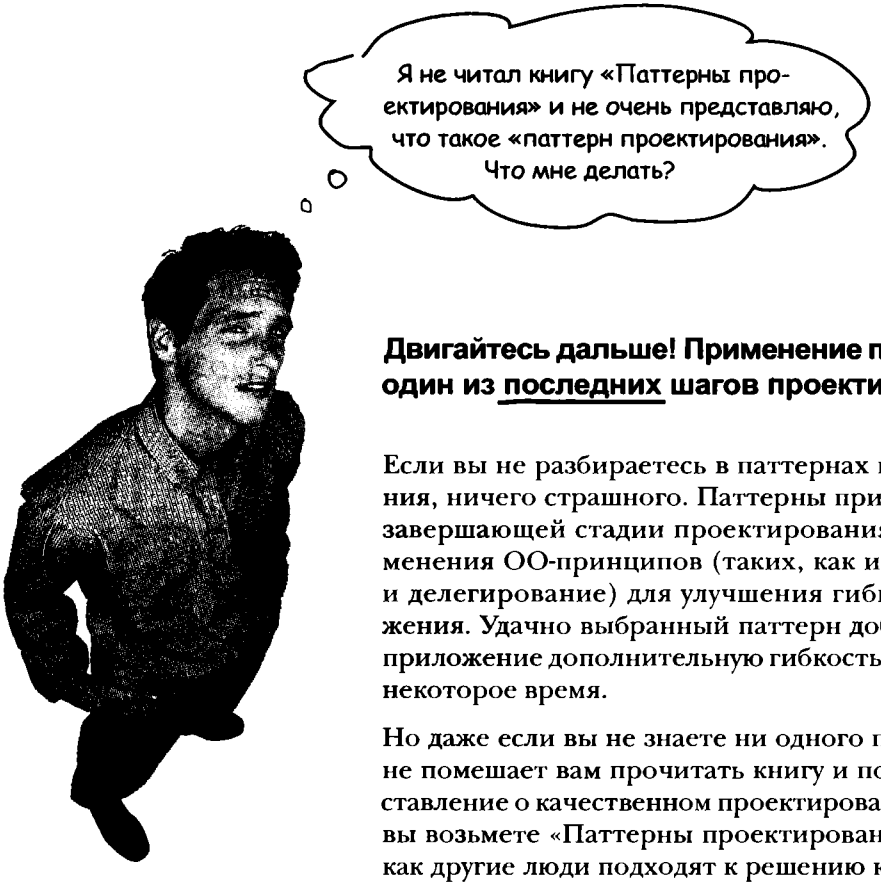
Что такое «паттерны проектирования»? И как их использовать?

Все мы в своей работе использовали готовые библиотеки и программные среды. Мы пишем код для их API, компилируем в свои программы и экономим время за счет использования кода, написанного кем-то другим. Только подумайте, сколько функциональности вам предоставляет Java API: сеть, графический интерфейс, ввод/вывод... Библиотеки и программные среды становятся значительным шагом на пути к модели разработки, при которой вы просто выбираете нужный компонент и подключаете его к своей программе. Однако компоненты не способствуют формированию структуры приложений, с которой они становятся более понятными, гибкими и удобными в сопровождении. И здесь на помощь приходят паттерны проектирования.

Паттерны проектирования не переходят сразу в код, сначала они запечатлеваются у вас в МОЗГЕ. Паттерн представляет собой стандартный способ проектирования для решения определенного вида задач. Когда у вас в мозге сформируется хорошее практическое знание паттернов, вы постепенно начнете применять их для решения новых задач и переработки старого кода, который со временем превращается в запутанный клубок «спагетти-кода».



Дополнительную информацию о паттернах проектирования и о том, как их использовать, вы найдете в книге «Паттерны проектирования».



Двигайтесь дальше! Применение паттернов — один из последних шагов проектирования.

Если вы не разбираетесь в паттернах проектирования, ничего страшного. Паттерны применяются на завершающей стадии проектирования, после применения ОО-принципов (таких, как инкапсуляция и делегирование) для улучшения гибкости приложения. Удачно выбранный паттерн добавит в ваше приложение дополнительную гибкость и сэкономит некоторое время.

Но даже если вы не знаете ни одного паттерна, это не помешает вам прочитать книгу и получить представление о качественном проектировании. А потом вы возьмете «Паттерны проектирования», узнаете, как другие люди подходят к решению классических задач проектирования, и извлечете полезные уроки из их опыта.

Немного запутались?

В этой главе мы занимались разными темами. Может показаться, что некоторые из них никак не связаны друг с другом...

→ Сбор функциональных требований

→ Анализ предметной области

→ Разбиение системы Гэри на модули

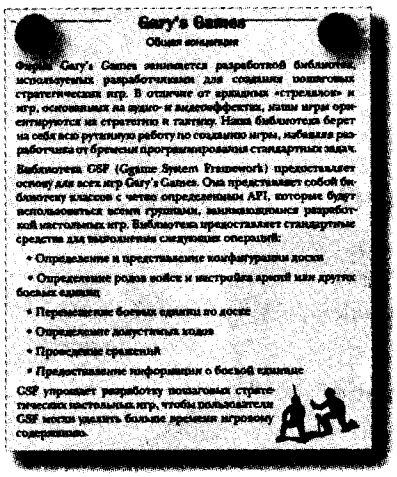
→ Определение факта применения паттерна MVC в системе Гэри.

Но как все это помогает в решении действительно БОЛЬШИХ задач?

Не забывайте: все это делалось для того, чтобы дать вам представление о том, как следует подходить к решению больших задач (таких, как игровая библиотека Гэри), работа над которыми отнюдь не сводится к простейшему проектированию и программированию.

А теперь мы раскроем секрет: вы уже сделали все необходимое для решения БОЛЬШОЙ задачи Гэри.

Сила ООАП
(и немного здравого смысла)



Gary's Games

Общая концепция

Фирма Gary's Games занимается разработкой библиотек, используемых разработчиками для создания пошаговых стратегических игр. В отличие от аркадных «стрелялок» и игр, основанных на аудио- и видеоэффектах, наши игры ориентируются на стратегию и тактику. Наша библиотека берет на себя всю рутинную работу по созданию игры, избавляя разработчика от бремени программирования стандартных задач.

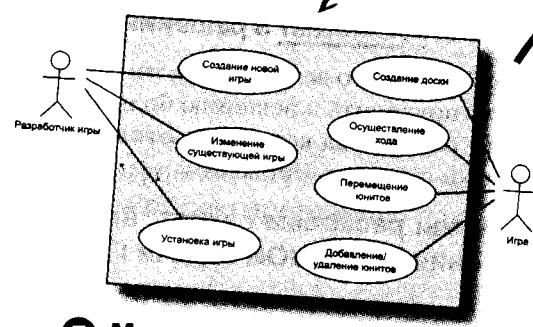
Библиотека GSF (Game System Framework) предоставляет основу для всех игр Gary's Games. Она представляет собой библиотеку классов с четко определенным API, которые будут использоваться всеми группами, занимающимися разработкой настольных игр. Библиотека предоставляет стандартные средства для выполнения следующих операций:

* Определение и представление конфигурации доски
* Определение родов войск и настройка армий или других боевых единиц
* Перемещение боевых единиц по доске
* Определение допустимых ходов
* Проведение сражений
* Предоставление информации о боевой единице

GSF упрощает разработку пошаговых стратегических настольных игр, чтобы пользователи GSF могли уделить больше времени игровому содержанию.

Мы начали с этой, довольно-таки туманной общей концепции. Перед нами БОЛЬШАЯ задача.

Теперь мы знаем, что нам нужно сделать, и диаграмма помогает понять общую задачу.



1 Мы прислушались к заказчику.

3 Мы нарисовали план создаваемой системы.

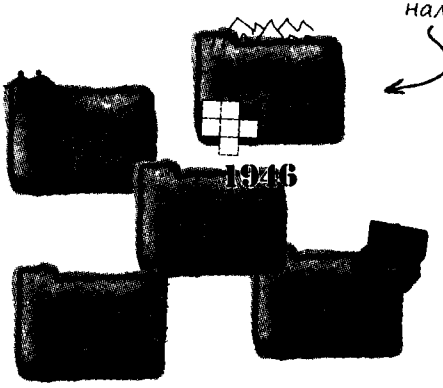
Игровая библиотека
Список Возможностей

1. Поддержка разных типов рельефа.
2. Поддержка разных исторических периодов, а также фантастического и фантазийного антуража.
3. Поддержка разных типов войск в зависимости от конкретной игры.
4. Поддержка модулей расширения для дополнительных кампаний и вариантов.
5. Доска состоит из клетчатого поля, каждой клетке присваивается тип рельефа.
6. Библиотека следит за очередностью хода.
7. Библиотека координирует базовые перемещения.

Анализ предметной области помог нам разобраться в том, чего хочет от своей системы Гэри.

2 Мы убедились в том, что правильно понимаем систему.

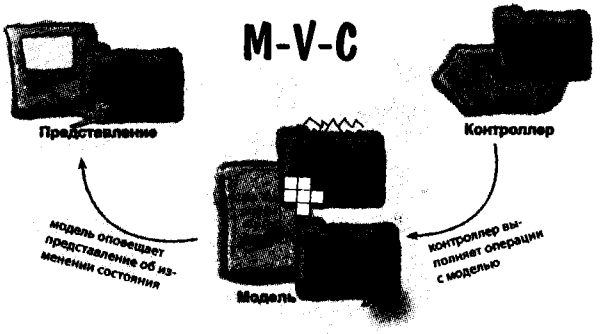
Имея готовый план системы и список функциональных возможностей, мы смогли разбить большое приложение Гэри на функциональные блоки меньшего размера.



4 Большая задача разбивается на меньшие функциональные блоки.

Мы даже взяли паттерны проектирования, уже известные нам, и применили их к своей системе.

Вы уже умеете решать меньшие задачи с использованием всего, что знаете об анализе и проектировании...



...и даже можете понять, как в этой ситуации применить паттерн MVC из книги «Паттерны проектирования».

5 Паттерны проектирования помогают в решении меньших задач.

Поздравляем!

Вы успешно превратили БОЛЬШУЮ ЗАДАЧУ в набор МЕНЬШИХ ЗАДАЧ, которые Вы уже умеете решать.

Инструментарий ООАП

В этой главе вы столкнулись с большой задачей, но она вас не напугала! Вспомните некоторые приемы решения больших задач, с которыми вы познакомились в этой главе.

Требования

Хорошие
гают уб
ваша сис
как хоче

Требован
страняп
антов и

Из вариа
можно у
казчик з

Анализ в
ния мож
или отс
вания.

Требован
(и расте

Анализ

Хорошо сн
легко изм
ется.

Используй
принципы
суляция и
улучшени
программ

Если стр
ложения н
ИЗМЕНИ
вольствуй
турой, да
вами.

Следите
обладали

Решение больших задач

Выслушайте заказчика, чтобы понять, какая система ему нужна.

Постройте список функциональных возможностей на языке, понятном заказчику.

Убедитесь, что функциональны
возможности соответствуют
нуждам заказчика.

Постройте общие планы систе
с помощью диаграмм вариант
использования (и вариантов ис
пользования).

Разбейте большую систему
множество небольших частей.

Применяйте паттерны проек
рования к фрагментам систе

Используйте ООАП при рабо
фрагментами.

В этой главе появилась новая категория инструментов.

ОО-принципы

Инкапсулируйте то, что изменяется.

Программируйте для интерфейса, а не для реализации.

Каждый класс приложения должен иметь только одну причину для изменения.

Классы создаются ради поведения и функциональности.

КЛЮЧЕВЫЕ МОМЕНТЫ

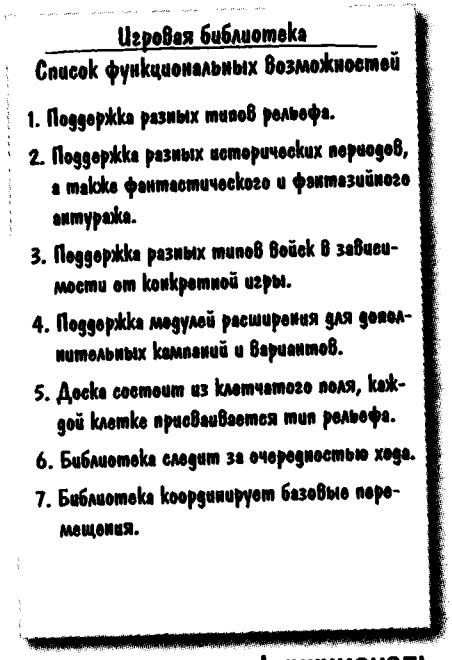
- Большую задачу лучше всего рассматривать как совокупность меньших задач.
- Как и в меньших проектах, работа над большой задачей должна начинаться со сбора функциональных возможностей и требований.
- «Функциональные возможности» — это «высокоуровневые» задачи, решаемые системой, но иногда этот термин используется как синоним «требований».
- Общность и изменчивость применяются для сравнения новой системы с другими, уже известными вам.
- Варианты использования подробны, диаграммы вариантов использования направлены на получение общей картины.
- На диаграмме вариантов использования должны быть отражены все функциональные возможности системы.
- В ходе анализа предметной области система представляется на языке, понятном заказчику.
- Субъектом называется любая сущность, которая взаимодействует с системой, но не является ее частью.

Навести порядок в хаосе

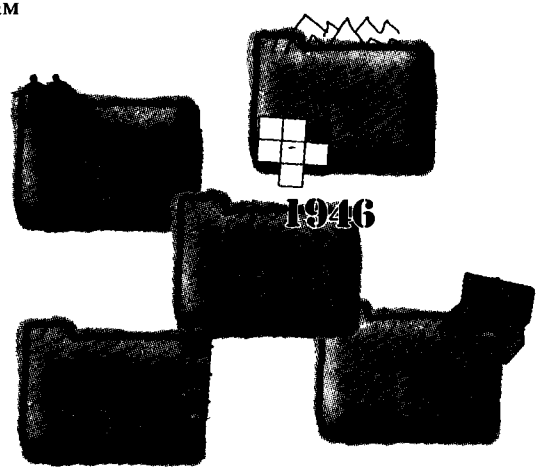
Любая работа с чего-то начинается, но вам стоит выбрать *правильное* что-то! Вы уже знаете, как разбить приложение на множество мелких задач, но это означает, что мелких задач будет действительно МНОГО. В этой главе мы поможем вам разобраться, с чего начать и как избежать потерь времени на движение в ошибочном направлении. Пришло время взять все эти мелкие детали, разложенные на вашем рабочем месте, и понять, как собрать из них четко структурированное, хорошо спроектированное приложение. Попутно вы узнаете о трех главных вопросах архитектуры и поймете, что «Риск» — это не только интересная настольная игра из 1980-х годов.

Слегка растерялись?

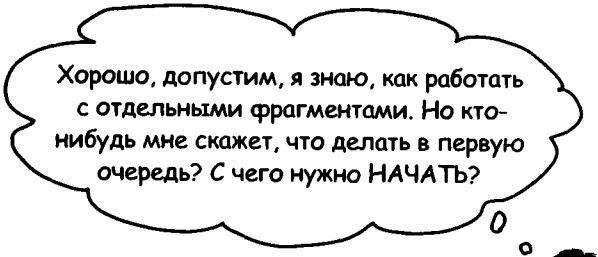
Итак, теперь у вас имеется множество мелких функциональных блоков, с которыми вы уже умеете работать... А также диаграммы вариантов использования, списки функциональных возможностей и множество других вещей, о которых вам придется позаботиться.



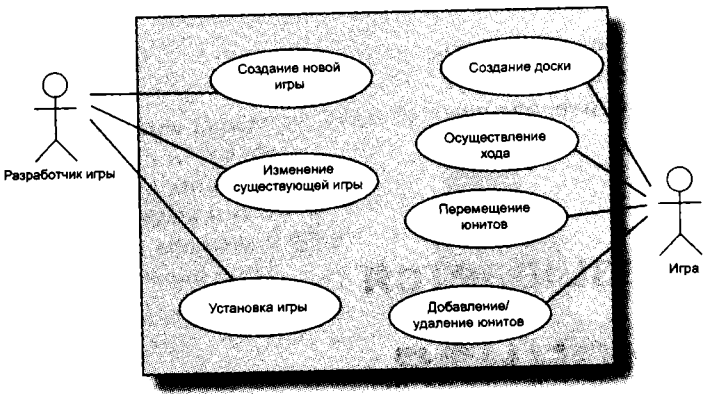
У нас есть списки функциональных возможностей...



...модули, которые нужно запрограммировать...



Помните нашего друга-программиста? Вот на чем мы расстались с ним в главе 6.



...высокоуровневая схема того, что нужно построить...

Gary's Games

Общая концепция

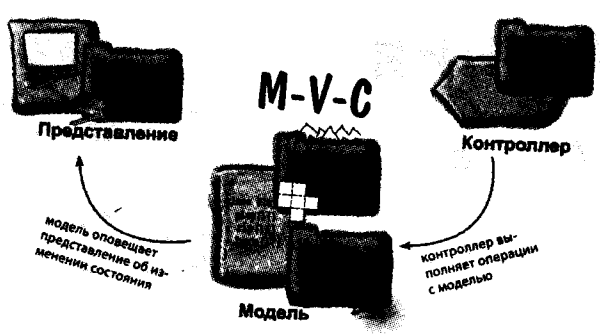
Фирма Gary's Games занимается разработкой библиотек, используемых разработчиками для создания пошаговых стратегических игр. В отличие от аркадных «стрелялок» и игр, основанных на аудио- и видеоэффектах, наши игры ориентируются на стратегию и тактику. Наша библиотека берет на себя всю рутинную работу по созданию игры, избавляя разработчика от бремени программирования стандартных задач.

Библиотека GSF (Ggame System Framework) предоставляет основу для всех игр Gary's Games. Она представляет собой библиотеку классов с четко определенным API, которые будут использоваться всеми группами, занимающимися разработкой настольных игр. Библиотека предоставляет стандартные средства для выполнения следующих операций:

- Определение и представление конфигурации доски
- Определение родов войск и настройка армий или других боевых единиц
- Перемещение боевых единиц по доске
- Определение допустимых ходов
- Проведение сражений
- Предоставление информации о боевой единице

GSF упрощает разработку пошаговых стратегических настольных игр, чтобы пользователи GSF могли уделить больше времени игровому содержанию.

...представления заказчика...



...и даже паттерны проектирования.

МОЗГОВОЙ ШТУРМ

Как вы думаете, важно ли, с чего вы начнете работу? И если важно, то почему? И с чего бы начали вы?

Нам нужна архитектура

Идентифицировать отдельные части большой задачи недостаточно. Также необходимо знать, как эти части работают в сочетании друг с другом и какие из них важнее других, — так вы определите, над чем следует работать *в первую очередь*.

Архитектура помогает в проектировании больших систем.

Архитектурой называется

организационная структура системы,

которая выделяет

важнейшие

части приложения и

отношения

между ними.

Диаграмма варианта использования дает некоторое представление об этом, но в целом относительно неясно, как организовано взаимодействие между модулями.

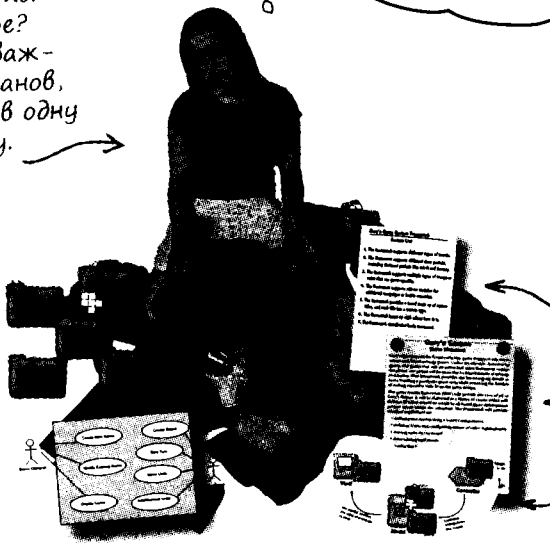
То, что нужно... Как же определить, что является самым важным, чтобы начать работу над приложением с этих частей?

Все это особенно важно при работе с другими программистами... Все должны понимать архитектуру системы.

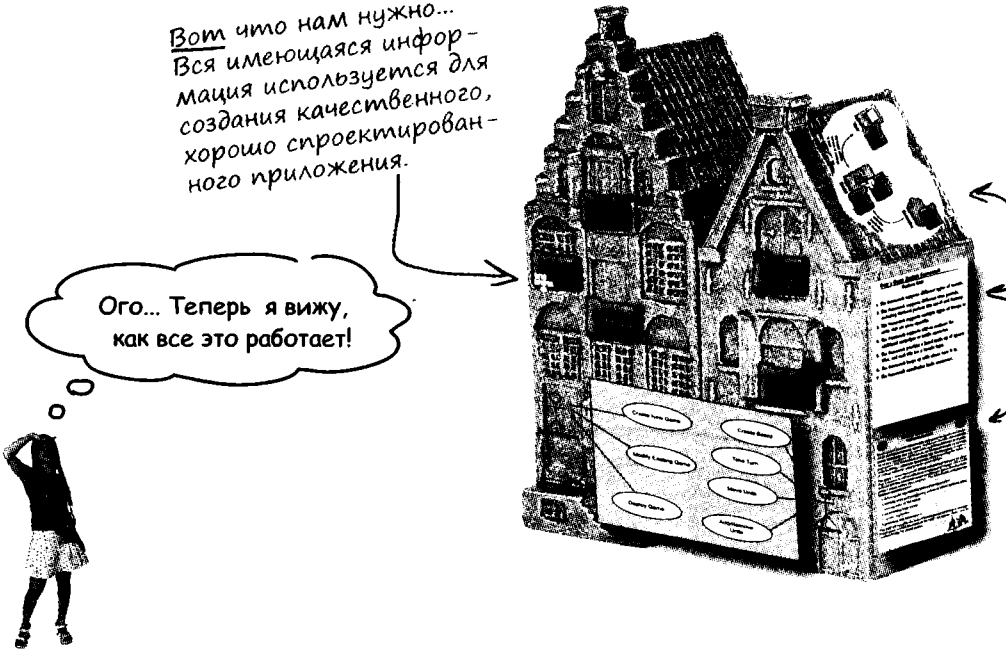
За партой

Архитектура. Организационная структура системы, включающая разбиение на части, связи между частями, механизмы взаимодействия, а также руководящие принципы и решения, используемые в ходе проектирования системы.

Архитектура превращает беспорядочную мешанину...



...в упорядоченное приложение



Хорошие программы пишутся одинаково независимо от того, работаете вы над мелким или большим проектом. Примените 3 шага, которые были представлены еще в главе 1.

Помните эту страницу из главы 1? 3 шага подходят и для построения хороших БОЛЬШИХ программ.

Хорошая программа за 3 простых шага

Может, сейчас они не кажутся простыми, но, как вы вскоре убедитесь, применение ООАП и некоторых базовых принципов навсегда изменит качество вашего кода

1. Убедитесь в том, что поведение программы соответствует требованиям заказчика.

Этот шаг ориентирован на заказчика. ПРЕЖДЕ ВСЕГО убедитесь что приложение делает то, что положено. В этом вам помогут четко сформулированные требования и анализ

2. Применяйте базовые ОО-принципы для повышения гибкости.

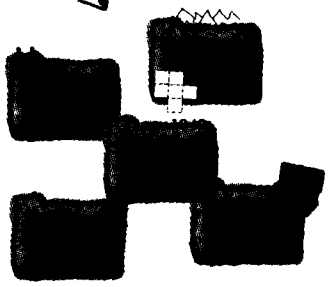
Когда программа заработает, поищите дубликаты кода, которые могли в ней остаться, и убедитесь в том, что в коде используются проверенные приемы ОО-программирования

3. Постарайтесь создать структуру кода, упрощающую его сопровождение и повторное использование.

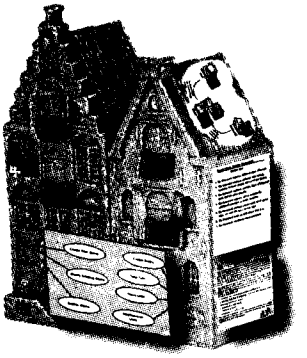
Получилось хорошее объектно-ориентированное приложение, которое делает то, что нужно? Примените паттерны и принципы, чтобы ваше приложение оставалось жизнеспособным в будущем.

дальше ›

Мы знаем, что эти 3 шага помогут нам справиться с каждым из отдельных компонентов игровой библиотеки.



Эти 3 шага применимы и при работе с большими приложениями. Итак, прежде чем браться за проектирование, следует сначала выяснить, чего же хочет от приложения заказчик.



По-настоящему БОЛЬШОЕ приложение

Начнем с функциональности

Прежде всего следует убедиться в том, что приложение делает то, что положено. В мелких проектах для этой цели использовался список требований; в больших проектах для определения соответствия приложения ожиданиям заказчика используется список функциональных возможностей.

Игровая библиотека

Список функциональных возможностей

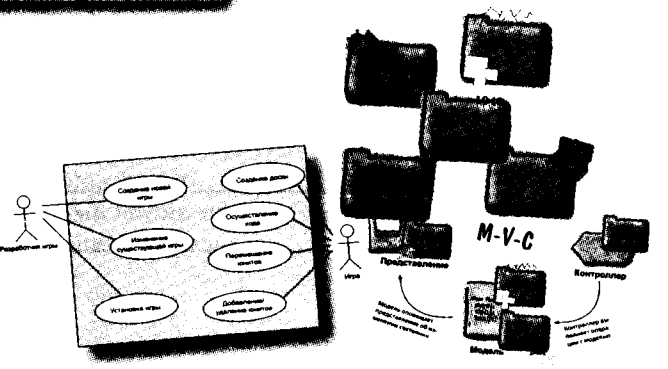
1. **Поддержка разных типов рельефа.**
2. **Поддержка разных исторических периодов, а также фантастического и фэнтезийного антуража.**
3. **Поддержка разных типов войск в зависимости от конкретной игры.**
4. **Поддержка модулей расширения для дополнительных кампаний и вариантов.**
5. **Доска состоит из клеточного поля, каждой клетке присваивается тип рельефа.**
6. **Библиотека следит за очередностью хода.**
7. **Библиотека координирует базовые перемещения.**

Все это относится к функциональности... Функциональные возможности определяют, что должна сделать система, а не то, какие принципы или паттерны будут использоваться для ее построения.

Позднее мы еще вернемся к другим диаграммам и паттернам... А пока сосредоточимся исключительно на функциональности системы.

Но что из этого считать самым важным?

Даже если мы знаем, что начинать следует с функциональности, все равно необходимо разобраться, какие блоки наиболее важны. Именно на них следует сосредоточиться в начале работы.



Возьми в руку карандаш

А как вы думаете, какие функциональные возможности из списка самые важные?

Хотя наш список состоит только из семи пунктов, каждый из них требует достаточно большого объема работы. Вы должны определить, какие функциональные возможности являются самыми важными и в каком порядке следует работать над ними.

Игровая система

Список функциональных возможностей

1. Поддержка разных типов рельефа.
2. Поддержка разных исторических периодов, а также фантастического и фэнтезийного антуража.
3. Поддержка разных типов войск в зависимости от конкретной игры.
4. Поддержка модулей расширения для дополнительных кампаний и вариантов.
5. Доска состоит из клеточного поля, каждой клетке присваивается тип рельефа.
6. Библиотека следит за очередностью хода.
7. Библиотека координирует базовые перемещения.

Нужно реализовать все перечисленные возможности, но в каком порядке за них браться?

Запишите 4 пункта, с которых вы бы начали работу над системой (в порядке реализации).

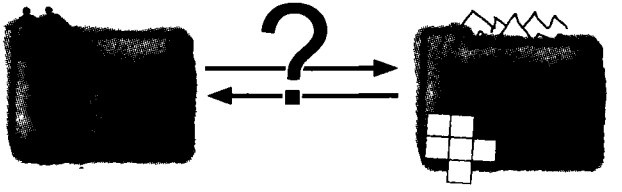
1. _____
2. _____
3. _____
4. _____

Один момент... Если архитектура описывает отношения между частями приложения, то почему мы говорим об отдельных частях? Разве не правильнее говорить о том, как эти части должны работать в сочетании друг с другом?

Действительно важные аспекты вашего приложения значимы с точки зрения архитектуры, поэтому НАЧИНАТЬ следует именно с них.

С чего-то же нужно начать!

Трудно говорить о взаимодействиях частей системы, если самих частей еще нет. Допустим, вы хотите поговорить о том, как модуль Board взаимодействует с модулем Units:



Чтобы понять, как взаимодействуют два модуля, необходимо сначала хотя бы в общих чертах представлять себе, как работает каждый из этих двух модулей.

Таким образом, архитектурное мастерство не сводится к выяснению отношений между частями приложения: проектировщик также должен определить, какие из частей являются **самыми** важными, чтобы начать с них построение системы.

Три основных вопроса архитектуры

Когда вы пытаетесь определить, имеет ли некоторый аспект архитектурную значимость, задайте себе три вопроса:

1. Относится ли эта часть к сути системы?

Является ли данная функциональная возможность неотъемлемой частью системы? Иначе говоря, можно ли представить систему без нее? Если нельзя, то, скорее всего, ваша функциональная возможность относится к **сути** системы.

И что это означает?

Если вы не уверены в том, какой смысл заложен в описание некоторой функциональной возможности, то, скорее всего, она достаточно важна, чтобы уделить ей особое внимание. Если вы в чем-то не уверены, работа над этим «чем-то» может занять много времени или создать проблемы с остальными составляющими системы. Лучше заняться этими функциональными возможностями на ранней, чем на поздней, стадии.

3. И как это сделать?

Также следует как можно раньше сосредоточиться на функциональных возможностях, которые **действительно трудны** в реализации или сопряжены с решением **совершенно новых** (для вас) задач из области программирования. Если вы понятия не имеете, как взяться за ту или иную задачу, лучше заняться ею пораньше, чтобы избежать серьезных проблем в процессе работы.

Стань архитектором

Справа приведен список функциональных возможностей, которые мы построили в предыдущей главе.

Определите, какие из пунктов списка являются значимыми с точки зрения архитектуры. В этом вам помогут три основных вопроса, о которых мы только что говорили.

Игровая библиотека

Список возможностей

1. Поддержка разных типов рельефа.
2. Поддержка разных исторических периодов, а также фантастического и фэнтезийного антуража.
3. Поддержка разных типов войск в зависимости от конкретной игры.
4. Поддержка модулей расширения для дополнительных кампаний и вариантов.
5. Доска состоит из клеточного поля, каждой клетке присваивается тип рельефа.
6. Библиотека следит за очередностью хода.
7. Библиотека координирует базовые перемещения.

Что важно?

_____

_____

_____

_____

_____

Проверьте, совпадают ли ваши ответы с нашими, приведенными на с. 358.

Почему?

_____

_____

_____

_____

_____

Запишите, какие из трех основных вопросов применимы по каждому пункту (вопросов может быть несколько).

Игровая система

Список функциональных возможностей

1. Поддержка разных типов рельефа.
2. Поддержка разных исторических периодов, а также фантастического и фэнтезийного антуража.
3. Поддержка разных типов войск в зависимости от конкретной игры.
4. Поддержка модулей расширения для дополнительных кампаний и вариантов.
5. Доска состоит из клеточного поля, каждой клетке присваивается тип рельефа.
6. Библиотека следит за очередностью хода.
7. Библиотека координирует базовые перемещения.

Стань архитектором. Решение

Справа приведен список функциональных возможностей, которые мы построили в предыдущей главе. Определите, какие из пунктов списка являются значимыми с точки зрения архитектуры. В этом вам помогут три основных вопроса, о которых мы только что говорили.

Что важно?	Почему?
Доска для игры	B1
Юниты, предназначенные для конкретной игры	B1, B2
Координация базовых перемещений	B3 (и возможно, B2)

Мы решили, что доска относится к сути игры... Без доски никакой игры вообще не будет!

Мы посчитали, что виды войск относятся к сути игры... а также не совсем ясно, что означает «предназначенные для конкретной игры». Поэтому к этому пункту применимы сразу два вопроса.

Выбор этих пунктов не столь очевиден, но у нас нет полной уверенности относительно их реализации. Определенно стоит заранее разобраться в том, что это означает и что нам нужно сделать.

Часто Задаваемые Вопросы

В: Я не понимаю, что имеется в виду под «сутью» системы. Можно чуть подробнее?

О: Суть системы — то, что она делает на самом общем уровне. Другими словами, если удалить все украшения и примочки, все «клевые» штучки, добавленные по соображениям маркетинга, и все творческие идеи, что реально будет делать система? Это и есть ее суть.

Рассматривая функциональную возможность, спросите себя: «Если бы это не было реализовано, была бы система тем, чем ей положено быть?» И если ответ будет отрицательным, значит, вы нашли то, что относится к сути. В системе Гэри мы решили, что игра **не** будет игрой без доски и юнитов. Некоторые дополнительные примеры представлены во врезке в нижней части страницы.

В: Если вы не до конца понимаете смысл какого-то пункта списка, не является ли это признаком плохих требований?

О: Нет, но это признак того, что вам, возможно, придется сформулировать дополнительные требования или, по крайней мере, уточнить информацию. На ранней стадии проектирования некоторые подробности можно опустить просто для того, чтобы получить общее представление о системе. Но сейчас пробелы пора заполнить; собственно, в этом и заключается смысл второго из основных вопросов архитектуры.

В: В ходе работы над новой системой я, скорее всего, не буду знать, как реализовать некоторые пункты списка функциональных возможностей. Не значит ли это, что третий вопрос применим абсолютно всегда?

О: Вовсе нет. Даже если вы никогда не писали код, определяющий, ввел ли пользователь команду «q» или «x», вы умеете писать простейшие команды `if/else`, и с получением ввода с клавиатуры трудностей не будет. Таким образом, если вы не знаете, как реализуется та или иная функциональная возможность, или никогда не писали код для выполнения именно этой операции, все обычно сводится к нескольким подробностям.

Но если вам нужно написать многопоточный чат-сервер, а вы ни разу не писали многопоточные и сетевые программы, — вот тогда вы действительно не знаете, что делать. Именно на такие пункты следует обращать внимание: особо сложные задачи, решение которых вам практически неизвестно.

В: В итоге все сводится к субъективной оценке?

О: Во многих случаях — да. Но если вы начнете работу с того, что вам кажется наиболее важным для системы, то успешное начало вам обеспечено.

А вот чего определенно не следует делать, так это начинать с пунктов, которые кажутся вам знакомыми (если вы уже решали похожую задачу в другом проекте). Начинайте с важнейших компонентов системы, даже если они на первый взгляд кажутся относительно сложными, и вы сделаете первый шаг к успеху.

Суть системы — то, что она делает на самом общем уровне.

Мозговой штурм

Как вы думаете, в чем выражается суть следующих систем:

- Метеорологическая станция?
- Пульт для управления домашней электроникой?
- Приложение для микширования музыки и управления ритмом?

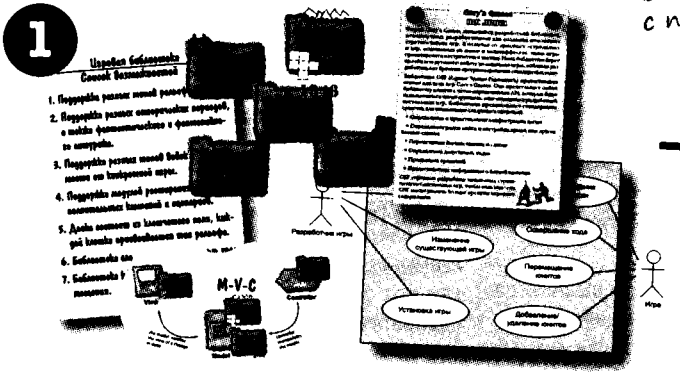
Хаоса стало чуть меньше...

Используя три основных вопроса архитектуры, мы начали постепенно наводить порядок в хаосе, с которого все началось:

Помните все это? Все начиналось с полного хаоса...

Но потом мы сосредоточились на том, чтобы заставить систему делать то, что задумано.

1



2

Игровая библиотека
Список возможностей

1. Поддержка разных типов рельефа.
2. Поддержка разных исторических периодов, а также фантастического и фантезийного сюжета.
3. Поддержка разных типов войск в зависимости от конкретной игры.
4. Поддержка модулей расширения для дополнительных компаний и сценариев.
5. Доска состоит из клеточного поля, каждой клетке приписывается тип рельефа.
6. Библиотека следит за очередностью хода.
7. Библиотека координирует базовые переменные.

3

Игровая система
КЛЮЧЕВЫЕ функциональные возможности

1. **Доска для игры — суть системы.**
2. **Юниты для конкретной игры — суть системы. Что это означает?**
3. **Координация перемещений: что это и как это сделать?**

В результате мы выбрали несколько ключевых возможностей, на которых мы сосредоточимся на первой стадии работы.

Конечно, я люблю парней в форме, но все равно есть из кого выбирать...

...но сделать предстоит еще много

Мы ограничили систему Гэри тремя ключевыми функциональными возможностями, но остается большой вопрос: с чего начать?

Спор ~~Разговор~~ в офисе

Джим: О чем вы вообще думаете? Как можно начинать с того, что не относится к сути системы?

Джо: Это просто смешно. Суть сутью, но мы должны выяснить, что это за «юниты, предназначенные для конкретной игры». Если это сложнее, чем мы думаем, на уточнение информации могут уйти недели!

Фрэнк: Возможно... Но мы точно знаем, что с координацией перемещений будут проблемы, мы понятия не имеем, как за нее взяться! Как можно начинать с чего-то другого, если мы знаем, что с перемещением будут сложности?

Джо: Но и с типами войск тоже могут возникнуть сложности! Мы пока этого не знаем, я об этом и говорю! Нужно разобраться с частями системы, о которых мы ничего не знаем, или нас ждут настоящие неприятности!

Джим: Если хотите, занимайтесь поддержкой перемещений и разными типами войск. А я займусь доской, потому что... ну... мне что-то подсказывает, что Гэри не представляет свою игровую систему без доски. И я не буду оставлять доску на потом... Я займусь ею сразу.

Фрэнк: Вы оба рехнулись. Хотите откладывать трудные задачи на будущее? Пожалуйста! А я постараюсь как можно раньше выяснить, как делается то, о чем я не имею ни малейшего понятия.

И кто, по *Вашему* мнению, прав?

- ☐ Джим (игровое поле)
- ☐ Джо (типы войск)
- ☐ Фрэнк (координация перемещений)

Поставьте галочку, если согласны с этой точкой зрения.

Пускай мальчики спорят. Я думаю, что **ВСЕ** они по-своему правы... Проблема не в том, с чего начинать, проблема — в **РИСКЕ**!

Джилл работает с Фрэнком, Джимом и Джо, и ей не привыкать вмешиваться в их споры.

Все эти функциональные возможности важны по одной причине: все они вводят РИСК в ваш проект. Неважно, с чего вы начнете, важно, чтобы вы двигались в направлении снижения РИСКОВ.

Давайте еще раз посмотрим на наши ключевые функциональные возможности:

Игровая система

Ключевые функциональные возможности

1. **Доска для игры — суть системы.**
2. **Юниты для конкретной игры — суть системы. Что это означает?**
3. **Координация перемещений: что это и как это сделать?**

Если центральные функциональные возможности системы не будут реализованы, увеличивается РИСК, что система не понравится заказчику.

Так как мы не знаем, что это означает, этот пункт может скрывать большой объем работы, сопряженной с РИСКОМ для графиков и расписаний.

Мы не уверены в том, как это делается. Возникает РИСК того, что мы не сможем разобраться в этом пункте или что он займет слишком много времени.

Ваша цель — СОКРАЩЕНИЕ РИСКА, а не споры о том, с какой из ключевых возможностей лучше начать. Начинайте с ЛЮБОЙ, при условии, что вы неуклонно стремитесь к построению заказанной системы.

Возьми в руку карандаш

Найдите риски в проекте.

Вспомните любой проект, которым вы занимаетесь на своей основной работе. Теперь запишите *первое*, с чего вы начали работу над проектом:

Подумайте о трех основных вопросах архитектуры, которые рассматривались на с. 356. Представьте, что эти вопросы относятся к вашему проекту. Запишите несколько функциональных возможностей, которые, на ваш взгляд, были бы значимы с точки зрения архитектуры:

Внимательно присмотревшись к списку, вы, вероятно, увидите, что все его пункты сопряжены с РИСКОМ. Все они могут создать проблемы или задержать выполнение проекта. Запишите, над какими из функциональных возможностей следует работать в первую очередь и почему. Какие риски они создают? Какие риски удастся устранить, если поработать над ними в первую очередь?

Архитектурная головоломка

Работа над игровой системой Гэри начнется с модуля доски. Вам поручено написать интерфейс **Board**, который будет использоваться и расширяться разработчиками для создания их собственных игр.

Ситуация:

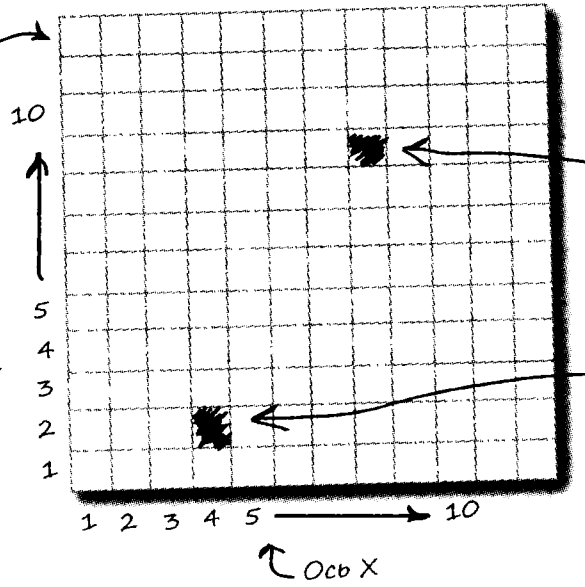
Нужно создать базовый тип **Board**, который будет использоваться разработчиками для создания новых игр. Ширина и высота доски задаются разработчиками. Кроме того, доска должна уметь возвращать поле в заданной позиции, добавлять юниты на поля и возвращать все юниты в позиции с заданными координатами X-Y.

Относительно подробные требования, полученные нами от Гэри и его группы.

Ваша задача:

1. Создайте новый класс с именем **Board.java**.
2. Добавьте конструктор **Board**, получающий высоту и ширину и создающий доску с заданными параметрами. Конструктор должен разбивать доску на клетки, характеризующиеся парой координат X-Y.
3. Напишите метод, возвращающий поле в позиции с заданными координатами X-Y.
4. Напишите методы, добавляющие юниты на клетку с координатами X-Y.
5. Напишите метод, возвращающий все юниты с указанного поля X-Y.

Пример доски. Эта доска имеет форму квадрата, хотя не все доски будут такими.



Ось Y

Это поле имеет координаты (8, 9).

А это поле находится в позиции (4,2).

Ответы на с. 370

Откровенно о вариантах использования

Интервью недели: Сценарии способствуют сокращению рисков

HeadFirst: Привет. Мы рады, что вы сегодня с нами.

Сценарий: И я тоже рад, особенно когда мы встречаемся в главе, посвященной не только вариантам использования.

HeadFirst: По правде говоря, мы немного удивились, когда нам сообщили, кто будет нашим гостем. Здесь речь идет о архитектуре и о функциональных возможностях, направленных на снижение рисков.

Сценарий: Безусловно! Это очень хороший метод решения больших задач.

HeadFirst: Ну да, конечно... Тогда почему вы здесь?

Сценарий: О! Извините, я думал, вы в курсе. Я здесь тоже из-за сокращения рисков.

HeadFirst: Но мы полагали, что вы — всего лишь один из конкретных путей в варианте использования. А ведь мы даже не написали ни одного варианта!

Сценарий: Неважно, я все равно могу помочь. Послушайте, давайте будем откровенны: многие разработчики не считают нужным потратить немного времени на написание вариантов использования. В конце концов, в главе 6 нам понадобилось четыре страницы только для того, чтобы убедить людей в пользе диаграмм вариантов использования, а ведь нарисовать диаграмму куда проще, чем написать вариант использования!

HeadFirst: Да, пожалуй... Программисты не любят писать варианты использования. Но ведь варианты использования действительно помогают, они нас сильно выручили в деле с дверью для Тодда и Джины.

Сценарий: Да, я согласен! Но когда у разработчиков нет времени или варианты использования слишком формальны, я могу предоставить многие преимущества вариантов использования без лишней волокиты.

HeadFirst: Хмм, звучит заманчиво. Расскажите подробнее.

Сценарий: Возьмем хотя бы доску, над которой вы сейчас работаете. Допустим, вы хотите сократить риски от того, что Гэри увидит ее и вспомнит что-то важное, что вы забыли добавить...

HeadFirst: Ах, да! Забытое важное требование — это всегда риск!

Сценарий: И тогда вы пишете простой сценарий использования своей доски (здесь-то я вам и пригожусь), а потом убеждаетесь, что доска работает в соответствии с написанным сценарием.

HeadFirst: Но ведь у нас нет варианта использования... Какие шаги мы выберем в созданном сценарии?

Сценарий: Не обязательно подходить формально. Вы можете использовать формулировку «Разработчик создает новую доску 8 на 10 клеток» или «Игрок 1 уничтожает войска игрока 2 на клетке (4, 5), доска удаляет войска игрока 2 с этого поля».

HeadFirst: А, то есть просто короткие описания использования доски?

Сценарий: Точно! Потом вы проходитесь по каждому описанию и убеждаетесь в том, что доска обрабатывает все случаи. Получается не так полно, как с вариантом использования, но я помогу вам проверить, что ни одно важное требование не было забыто.

HeadFirst: Потрясающе! Мы вернемся после небольшого перерыва.

ПУТАНИЦА

Напишите сценарий для только что написанного интерфейса Board.

Снижение рисков — самое главное в этой главе. Вы написали интерфейс Board на основании полученных требований, но теперь нужно проверить, не забыли ли мы чего-нибудь, до того, как Гэри увидит вашу работу и обнаружит ошибку.

Риск, который вы пытаетесь снизить или устранить с помощью сценария.

Возьмите фрагменты, расположенные в нижней части страницы, и разместите их на доске в правильном порядке. Сценарий должен воспроизводить реалистичную часть игры. Когда все будет готово, посмотрите, не забыли ли вы чего-нибудь в интерфейсе Board, и если забыли, добавьте недостающую функциональность в код. Возможно, некоторые фрагменты останутся неиспользованными, удачи!

Игровая система

Пример использования доски

И2 перемещает танки на поле (4, 5).

Запрос типа рельефа на поле (2, 2).

Мы поставили пару фрагментов, чтобы немного упростить вашу задачу.

Используйте оба столбца.

Расставьте фрагменты сценария на этой доске.

И1 сражается с И2.

Войска И1 проигрывают.

И2 перемещает армию на поле (4, 5).

Разработчик создает доску с заданной высотой и шириной.

Запрос типа рельефа на поле (4, 5).

Войска И1 удаляются с поля (4, 5).

Разработчик создает новую доску.

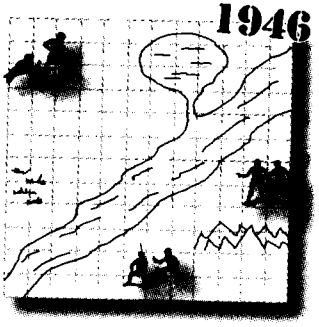
Войска И2 выигрывают сражение.

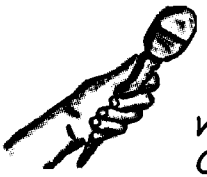
Запрос информации о войсках на поле (4, 5).

И1 ставит артиллерию на поле (4, 5).

И1 ставит подводные лодки на поле (2, 2).

Разработчик игры задает высоту и ширину.





Откровенно о вариантах использования

Интервью недели: Сценарии способствуют сокращению рисков (продолжение)

HeadFirst: И мы снова в эфире. Нам звонят слушатели, не могли бы вы ответить на несколько вопросов?

Сценарий: Конечно, с радостью.

HeadFirst: Замечательно. Вот вопрос, который задается чаще всего: «Значит, нам больше не нужно перебирать варианты использования? Будет достаточно сценария?»

Сценарий: Спасибо, я действительно часто слышу этот вопрос. Я твердо верю, что там, где это возможно, следует применять варианты использования. Я хорошо подхожу для простых задач и определения самых общих требований, но не забывайте: я – всего лишь один из представителей большого семейства Сценариев. Альтернативных путей много, поэтому не нужно ограничиваться единственным способом, вы можете упустить некоторые важные требования.

HeadFirst: Все верно, у нас уже была передача с участием Счастливого Пути и Альтернативного Пути.

Сценарий: Откровенно говоря, это всего лишь специализированные версии меня. Мы стараемся не обсуждать взаимоотношения в семье, каждый хочет найти собственное место в этом мире. Но все мы в действительности входим в семейство Сценариев. И если вы хотите, чтобы ваша система была построена правильно, вам понадобимся мы все. А в самом начале работы, когда построение вариантов использования кажется преждевременным, можно обойтись мной – это станет хорошей отправной точкой.

HeadFirst: Следующий вопрос: «Вы говорите, что способствуете снижению рисков. Ненавижу риски... Можно уточнить, как именно вы помогаете избежать их?»

Сценарий: Тоже хороший вопрос. Вспомните: при определении требований – с вариантами использования, диаграммой вариантов использования, сценарием – вы стараетесь убедиться в том, что создаваемая система делает то, что хочет заказчик. Без хороших требований появляется риск того, что ваша система будет работать неправильно, а это вызовет законное раздражение у заказчика.

HeadFirst: Значит, вы сокращаете риски в этапе требований?

Сценарий: Да, обычно так. Это происходит, когда вы пишете варианты использовании, объединяя требования заказчика предусматривая все возможные пути.

HeadFirst: Но ведь вы также приносите пользу и в архитектуре больших проектов, верно? Именно поэтому мы сейчас проводим это интервью?

Сценарий: Безусловно. Иногда проектировщик не имеет полного списка требований и набора вариантов использования, но для понимания того, как должна работать система, ему все равно нужен пример выполнения некоторой операции. Именно это и происходит: сценарий дает представление о базовой работе модуля или фрагмента кода, чтобы вы могли расставить базовые структурные элементы приложения по своим местам.

HeadFirst: Так вы, выходит, мастер на все руки?

Сценарий: Хотелось бы надеяться. Я помогаю не только в сборе требований и проверке полноты вариантов использования, но и в области архитектуры, где я сокращаю риски и проясняю основную схему работы модуля или блока программного кода.

ПУТАНИЦА. РЕШЕНИЕ

Напишите сценарий для только что запрограммированного интерфейса Board.

Ниже приведен сценарий, который получился у нас. Ваш ответ может отличаться, но в нем должны присутствовать как минимум операции создания доски разработчиком, сражение между игроком 1 и игроком 2 и добавление/удаление юнитов с доски.

Игровая система

Сценарий доски

Разработчик создает доску с заданной высотой и шириной.

И2 перемещает танки на поле (4, 5).

И2 перемещает армию на поле (4, 5).

И1 ставит артиллерию на поле (4, 5).

Запрос информации о войсках на поле (4, 5).

И1 сражается с И2.

Запрос типа рельефа на поле (4, 5).

Войска И2 выигрывают сражение.

Войска И1 удаляются с поля (4, 5).

И1 ставит подводные лодки на поле (2, 2).

Запрос типа рельефа на поле (2, 2).

Эта часть описывает сражение между игроком 1 и игроком 2.

Продолжение во втором столбце.

Эта часть не является обязательной, но она показывает, что доске, вероятно, потребуется получать тип рельефа для базовых запросов на перемещение.

Войска И1 проигрывают.

Разработчик создает новую доску.

Разработчик игры задает высоту и ширину.

В нашем примере эти фрагменты не используются.

Часто задаваемые вопросы

В: Откуда взялись требования для головоломки на с. 364?

О: Они были получены от Гэри — с добавлением толики здравого смысла. Если поразмыслить над тем, чего хочет Гэри (игровая система), а потом вернуться к разговору с заказчиком из главы 6, вы, вероятно, сможете сформулировать ряд собственных требований. Мы добавили несколько уточнений (скажем, возможность добавления войск на поле с заданными координатами), но это всего лишь результат тщательного продумывания постановки задачи.

В: Почему мы не написали вариант использования для определения требований?

О: Можно и так. Но помните: сейчас мы стараемся не столько полностью определить модуль Board, сколько расставить базовые компоненты по своим местам. Это все, что необходимо для сокращения риска при работе над этим компонентом системы. Более того, чрезмерная детализация может привести к повышению рисков в проекте из-за добавления подробностей, несущественных на текущей стадии.

В: Вы хотите сказать, что варианты использования могут повышать риски? Быть не может!

О: Нет, варианты использования не повышают риски, если они используются в нужный момент. А сейчас у нас имеется несколько ключевых функциональных возможностей, которые могут создать нам проблемы, если мы не разберемся в них. Но это не означает, что интерфейс Board не нужно совершенствовать; просто сейчас нам нужно понять, как он работает, чтобы быстро выявить все проблемные места и избежать проблем в будущем. Следовательно, на этой стадии написание полноценных вариантов использования — это перебор.

Но когда мы определим ключевые функциональные возможности и справимся с основными рисками, мы вернемся к каждому модулю и начнем наполнять его подробностями. И тогда варианты использования станут очень важны.

В: Так вот почему мы использовали сценарий? Чтобы не возиться с лишними подробностями?

О: Точно. Сценарий предоставляет многие преимущества вариантов использования, не заставляя иметь дело с многочисленными подробностями, которые пока для нас несущественны.

Архитектурная головоломка (Повторно)

Чего не хватает в Board.java?

Присмотритесь повнимательнее к сценарию на предыдущей странице. Охватывают ли требования на с. 364 всё в законченном сценарии? Если вы считаете, что чего-то не хватает, добавьте описание внизу и включите в Board.java код, реализующий недостающую функциональность.

_____

_____

Архитектурная головоломка. Решение

Наша работа над игровой системой начнется с модуля Board. Ниже приведен интерфейс, написанный нами для выполнения задач, которые мы сочли основными для игровой доски. Посмотрите, совпадает ли ваше решение с нашим.

```
package headfirst.gsf.board;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import headfirst.gsf.unit.Unit;

public class Board {

  private int width, height;
  private List tiles;

  public Board(int width, int height) {
    this.width = width;
    this.height = height;
    initialize();
  }

  private void initialize() {
    tiles = new ArrayList(width);
    for (int i=0; i<width; i++) {
      tiles.add(i, new ArrayList(height));
      for (int j=0; j<height; j++) {
        ((ArrayList)tiles.get(i)).add(j, new Tile());
      }
    }
  }
```

Класс Board размещается в пакете, относящемся к игровой доске. Так мы поступим со всеми модулями, которые мы определили в главе 6.

Вскоре мы создадим этот класс, необходимый для построения Board.

Этот конструктор был указан в требованиях. Он получает ширину и высоту доски, после чего вызывает initialize() для ее инициализации.

Координатная сетка на доске представляется массивом массивов, при этом ширина и высота определяют размеры массивов.

Принцип проектирования: выделите код инициализации в отдельный метод, чтобы он не усложнял чтение остального кода.

В каждой координатной позиции добавляется экземпляр Tile. Этот вспомогательный класс тоже необходим для работы Board... Определение Tile приведено на следующей странице.

Board.java

Эти методы не нуждаются в пояснениях. Они были определены в составе требований на с. 364.

```
  public Tile getTile(int x, int y) {
    return (Tile)((ArrayList)tiles.get(x-1)).get(y-1);
  }

  public void addUnit(Unit unit, int x, int y) {
    Tile tile = getTile(x, y);
    tile.addUnit(unit);
  }

  public void removeUnit(Unit unit, int x, int y) {
    Tile tile = getTile(x, y);
    tile.removeUnit(unit);
  }

  public void removeUnits(int x, int y) {
    Tile tile = getTile(x, y);
    tile.removeUnits();
  }

  public List getUnits(int x, int y) {
    return getTile(x, y).getUnits();
  }
}
```

Мы решили делегировать выполнение операций добавления и удаления юнитов классу Tile.

Вы должны были заметить из упражнения на с. 368, что нам понадобится механизм удаления юнитов с доски. В исходных требованиях он отсутствовал.

Еще одно место, в котором выполнение операции делегируется классу Tile. Так как класс поля хранит находящиеся на нем юниты, информацию о юнитах должен возвращать класс поля.

Часто задаваемые вопросы

В: Разве массив массивов не ограничивает нас квадратной доской?

О: Нет, хотя он и ограничивает нас доской с координатами (x, y). Например, координаты (x, y) могут использоваться доской с шестигранными полями (при правильно выбранной структуре). Но в основном массив массивов практически идеально подходит для прямоугольных досок с квадратными полями.

В: Но ведь это все равно ограничение? Почему не использовать граф или даже класс Coordinate, чтобы не быть привязанным к координатам (x, y) и прямоугольной доске?

О: Если вам нужна максимальная гибкость, возможно, это хорошая мысль. Но в нашей ситуации координаты (x, y) уже указаны в требованиях (см. с. 364). Поэтому мы выбрали решение не столь гибкое, но, безусловно, более простое. Помните, на этой стадии мы стараемся сократить риски, а не увеличить их за счет использования решения более сложного, чем реально необходимо.

Классы Tile и Unit

Чтобы код **Board** компилировался и работал, необходимо создать классы **Tile** и **Unit**. Вот как выглядят эти классы в нашей реализации:

```
package headfirst.gsf.unit;

public class Unit {

  public Unit() {
  }
}
```

Unit.java

Мы постарались сделать класс Unit как можно более простым. Существует немало подробностей, которые придется добавить позже, но сейчас для работы Board они не нужны.

Tile находится в одном пакете с классом Board... Между ними существует тесная связь.

Экземпляр Tile хранит список находящихся на нем юнитов.

Методы, используемые Board для операций с юнитами, являются защищенными, то есть обращаться к ним могут только классы из headfirst.gsf.board.

```
package headfirst.gsf.board;

import java.util.LinkedList;
import java.util.List;

import headfirst.gsf.unit.Unit;

public class Tile {

  private List units;

  public Tile() {
    units = new LinkedList();
  }

  protected void addUnit(Unit unit) {
    units.add(unit);
  }

  protected void removeUnit(Unit unit) {
    units.remove(unit);
  }
}
```

Tile.java

Правильный выбор цели

Сейчас вам не нужно беспокоиться обо всем, что классы **Tile** и **Unit** должны будут делать когда-нибудь в будущем. Сейчас важно заставить работать класс **Board** и его ключевые возможности, а не написать законченную версию **Tile** или **Unit**. Вот почему мы оставили класс **Unit** таким пустым и добавили минимальный набор методов в **Tile**.

Занимайтесь одной функциональной возможностью за раз, чтобы снизить риски.

Не отвлекайтесь на функциональные возможности, которые не помогут снизить риски.

Часто задаваемые вопросы

В: Если класс Tile выполняет операции добавления и удаления юнитов и мы можем получить от Board поле по координатам при помощи метода getTile(), зачем включать в Board методы addUnit() и removeUnit()? Почему бы не вызвать getTile(), а потом использовать экземпляр Tile для этих операций?

О: Да, вы можете пойти по этому пути и выполнять все операции с юнитами при помощи объекта `Tile`, возвращенного `getTile()`. Мы решили включить методы, относящиеся к `Unit`, в класс `Board` и сделать `Board` основной точкой входа для разработчиков игр. Как вы увидите на следующей странице, методы `Tile` объявлены защищенными, так что `addUnit()` и `removeUnit()` могут вызываться напрямую только классами из одного пакета с `Tile`, например `Board`. Таким образом, для работы с полями (а в конечном итоге и с рельефом) будет использоваться объект `Board`.

В: Я все равно считаю, что сейчас можно немного забежать вперед и включить методы, которые понадобятся в Unit и Tile. Почему бы не уделить этим классам больше внимания сейчас?

О: Сейчас мы не пытаемся запрограммировать всю игровую систему; наша цель — реализовать несколько ключевых функциональных возможностей и сократить общие риски проекта. Время, потраченное на написание класса `Unit` или наполнение класса `Tile`, не поможет сократить общие риски. Сделайте ровно столько, сколько необходимо для работы класса `Board`, потому что мы решили, что именно класс `Board` относится к сути системы и отсутствие этого компонента создает риск для системы в целом.

Когда вы разберетесь с ключевыми функциональными возможностями и сократите (или устраните) серьезные риски для проекта, у вас будет достаточно времени для работы над другими аспектами системы, такими как класс `Unit`. Но на данной стадии не стоит тратить время на то, что не способствует сокращению рисков для успеха проекта.

В Сети

Код классов, относящихся к классу Board из игровой системы Гэри, можно найти на сайте http://www.headfirstlabs.com. Найдите эту книгу и перейдите по ссылке Gary's Game System — Board classes.

Больше порядка, меньше хаоса

Архитектура и список ключевых функциональных возможностей системы помогли нам реализовать базовую версию **Board**, а также убедиться в том, что мы правильно представим суть системы заказчику. Давайте вернемся к списку ключевых возможностей:

Игровая система
КЛЮЧЕВЫЕ функциональные возможности

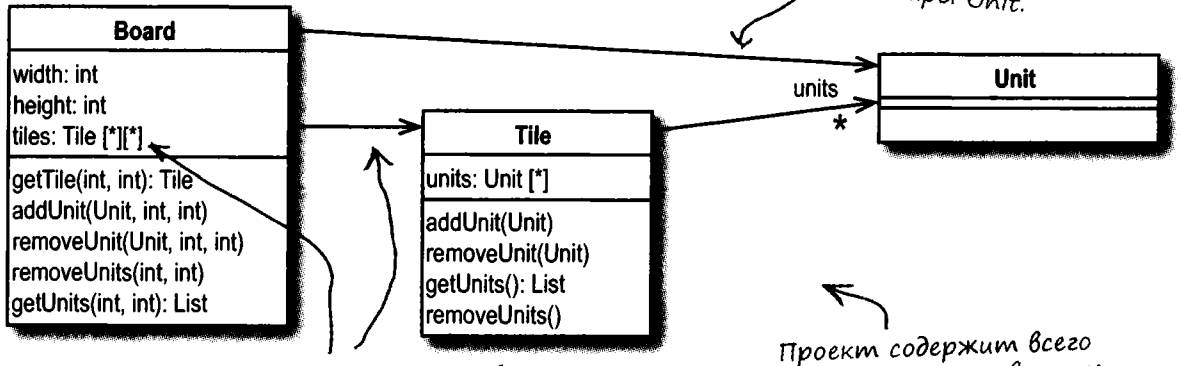
1. Доска для игры — суть системы.
2. Юниты для конкретной игры — суть системы. Что это означает?
3. Координация перемещений: что это и как это сделать?

Простейшая реализация Board написана, теперь можно переходить к следующему пункту списка.

И структура постепенно формируется...

Что еще лучше, в проекте появились некоторые важные классы, и в ходе работы над следующей ключевой функциональной возможностью мы можем продумать ее место в этой структуре.

И хотя класс Board не содержит переменных типа Unit, он все равно ассоциирован с Unit, поскольку его методы получают экземпляры Unit.



В UML не существует удобного представления многомерных массивов, поэтому мы используем обычную ассоциацию.

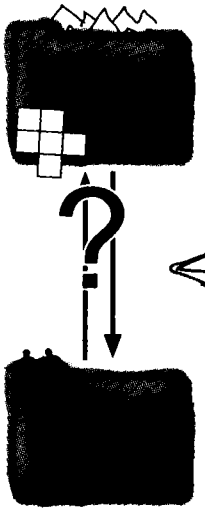
Проект содержит всего три класса, но и в этом виде он структурирован намного лучше, чем прежде.

Чем займемся теперь?

Класс Unit у нас уже есть, почему бы не взяться за «юниты, предназначенные для конкретной игры»? Заодно посмотрим, как взаимодействуют Board и Unit.

Старайтесь строить новые возможности на основе того, что уже сделано.

Когда у вас нет ничего, кроме требований и диаграмм, нужно только выбрать, с чего начать. Но теперь, когда в системе появился код и классы, проще всего выбрать другую ключевую возможность, связанную с тем, что уже построено. Помните наше определение архитектуры?



Архитектурой называется

организационная структура системы,

которая выделяет

важнейшие

части приложения и

отношения

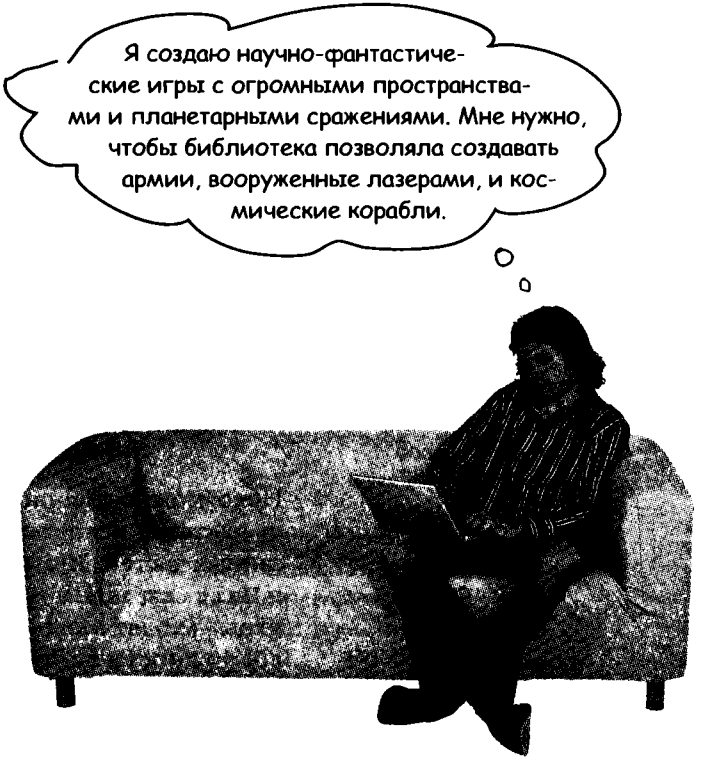
между ними.

После написания важных классов можно переходить к рассмотрению взаимодействий между классами, а также начинать строить новую функциональность на основе того, что уже сделано.

Невозможно говорить об отношениях между частями, если у нас нет двух взаимосвязанных частей. Мы знаем, что классы **Board** и **Unit** связаны, а «юниты для конкретной игры» входят в ключевые функциональные возможности системы. Очевидно, что на следующем этапе работать нужно именно над ними.

Юниты для конкретной игры... что это означает?

Чтобы понять, что подразумевается под этим выражением, проще всего поговорить с потенциальными клиентами Гэри — разработчиками игр, которые будут использовать его системы. Давайте послушаем, что они скажут:



Наши клиенты помешаны на воздушных сражениях, так что наземные войска нам вообще не понадобятся. Пусть библиотека помогает создавать разнообразные типы самолетов с разными скоростями, вооружением и тому подобным.

Мы создаем реалистичные масштабные игры... И даже учитываем возраст и отношения между персонажами.

Не бывает хороших военных игр без оружия... множества типов оружия. А для пущего интереса наша техника может нести сразу два вида вооружений.

Возьми в руку карандаш

Что такое «юниты, предназначенные для конкретной игры»?

Итак, вы выслушали разработчиков игр, которые собираются использовать игровую библиотеку Гэри. Вероятно, теперь вы достаточно хорошо представляете, в чем заключается смысл второй ключевой функциональной возможности. Напишите внизу, как вы представляете себе «юниты, предназначенные для конкретной игры».

_____

_____

_____

_____

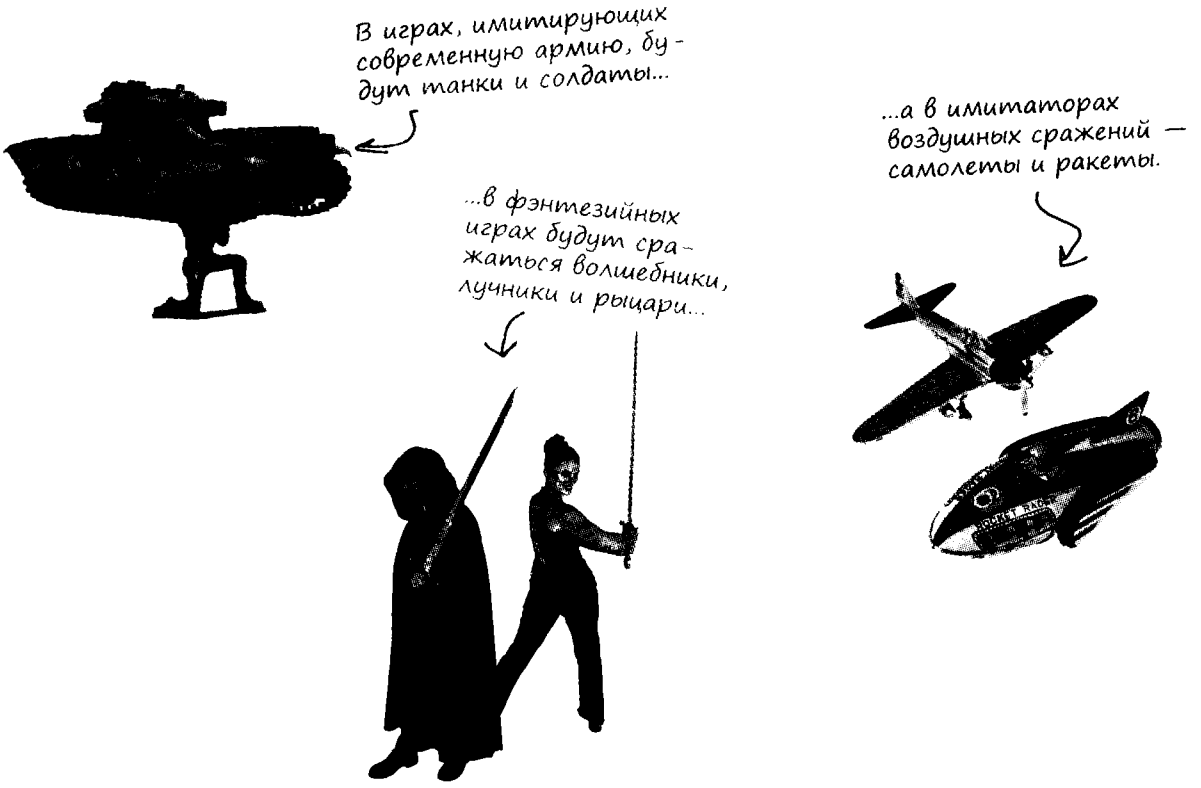
А когда выполните задание, сравните с нашими ответами на следующей странице.

Возьми в руку карандаш
Решение

Что такое «юниты, предназначенные для конкретной игры»?

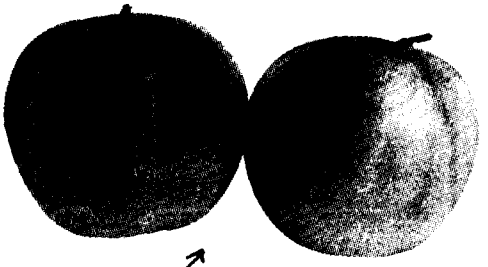
Итак, вы выслушали разработчиков игр, которые собираются использовать игровую библиотеку Гэри. Вероятно, теперь вы достаточно хорошо представляете, в чем заключается смысл второй ключевой функциональной возможности. Напишите внизу, как вы представляете себе «юниты, предназначенные для конкретной игры».

В каждой игре, созданной на основе библиотеки, используются разные типы юнитов, с разными атрибутами и возможностями. Следовательно, библиотека должна представлять свойства, отличающиеся от игры к игре, и поддерживать множественные типы данных для таких свойств.

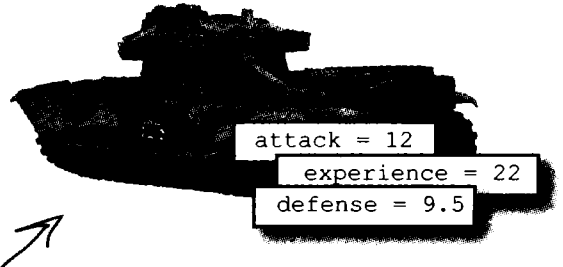


Возвращаемся к общности

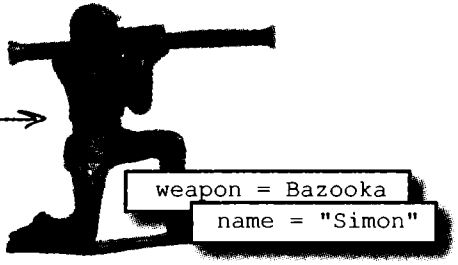
Постепенно становится понятно, чего от нас хочет Гэри, но мы все равно должны определить, как включить поддержку этой функциональной возможности в игровую библиотеку. Для начала рассмотрим разные типы юнитов, упомянутые разработчиками игр, и попробуем определить, что же у них общего.



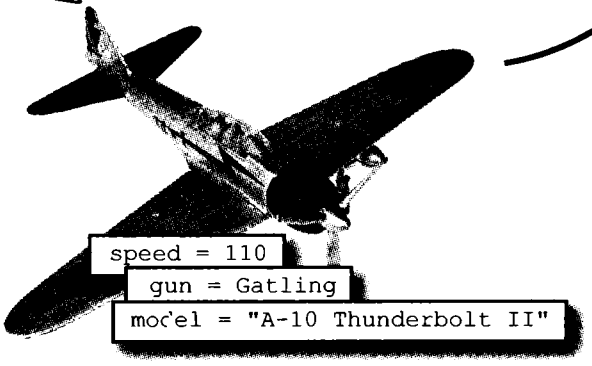
Мы говорили об общности на с. 287 главы 6, когда занимались сбором базовых требований системы. Однако этот прием также применим к меньшим задачам, таким как адаптация юнитов к конкретной игре.



Некоторые виды юнитов, упоминавшиеся на паре последних страниц, и их свойства.

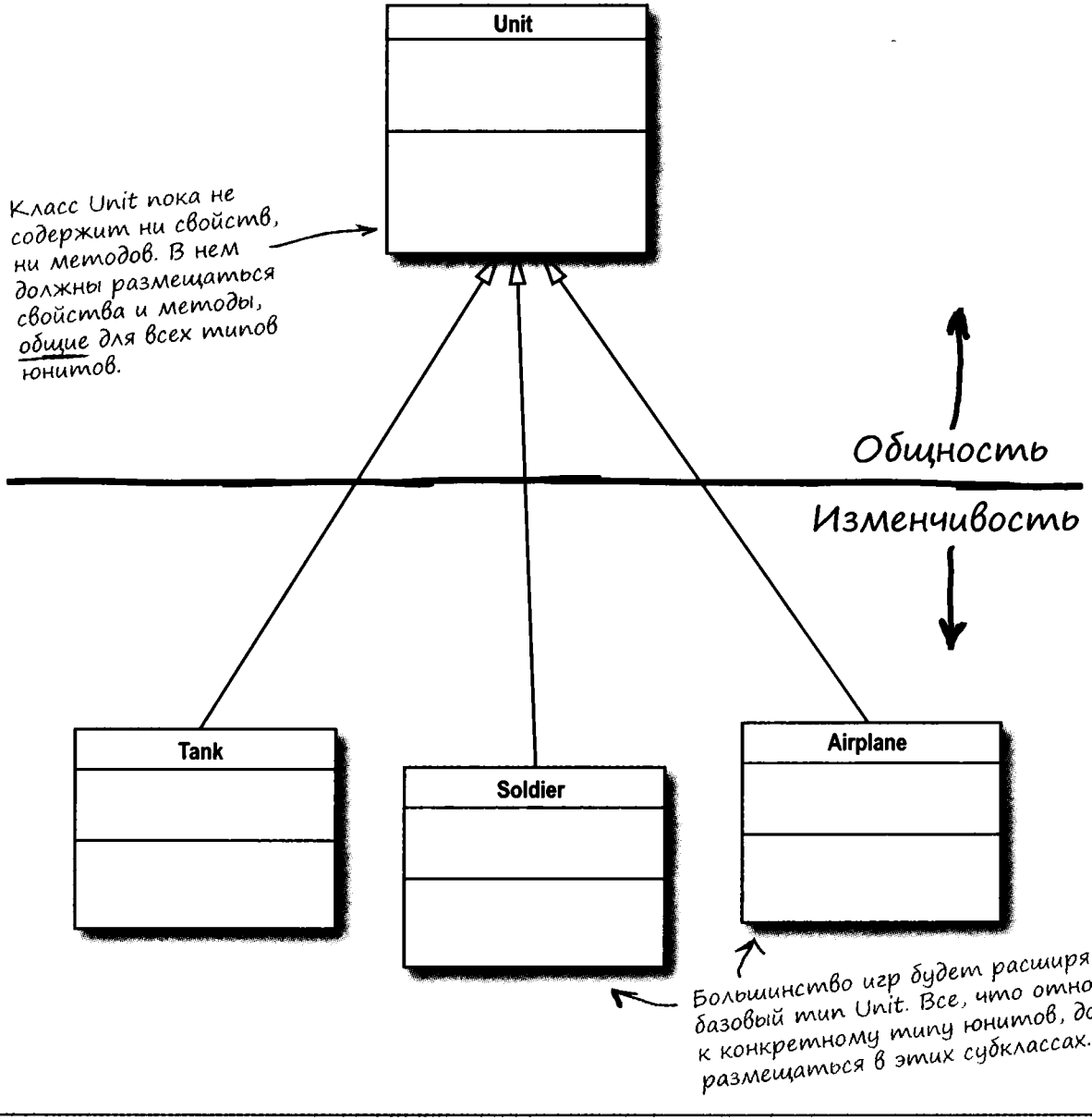


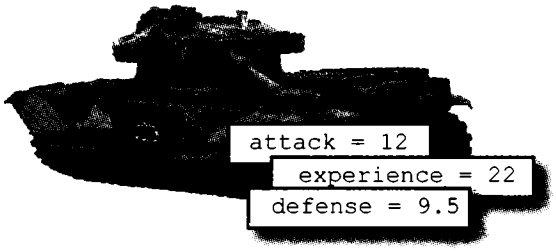
Что общего между разными типами юнитов? Какие свойства будут применимы к юнитам любой игры?



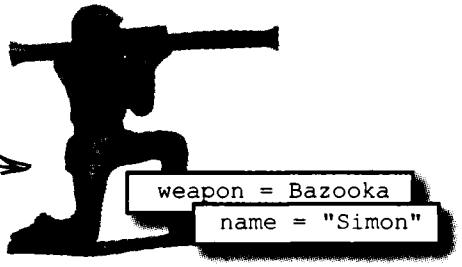
Головоломка

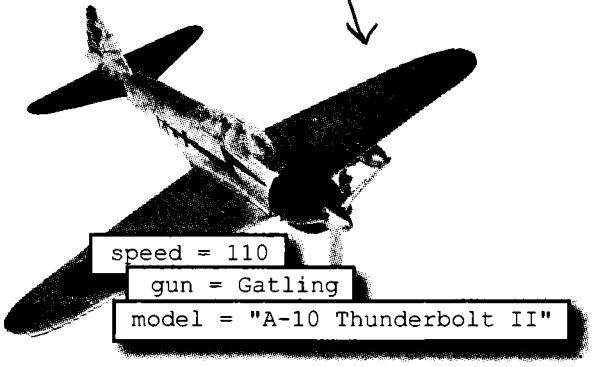
Определите, какие аспекты юнитов являются общими (а следовательно, должны стать частью базового класса Unit), а какие изменяются (и должны принадлежать субклассам Unit, создаваемым для конкретной игры). Запишите свойства и методы, которые, как вы полагаете, должны принадлежать Unit, на следующей диаграмме классов. Затем добавьте в нижней части свойства и методы, место которых в субклассах конкретных игр.





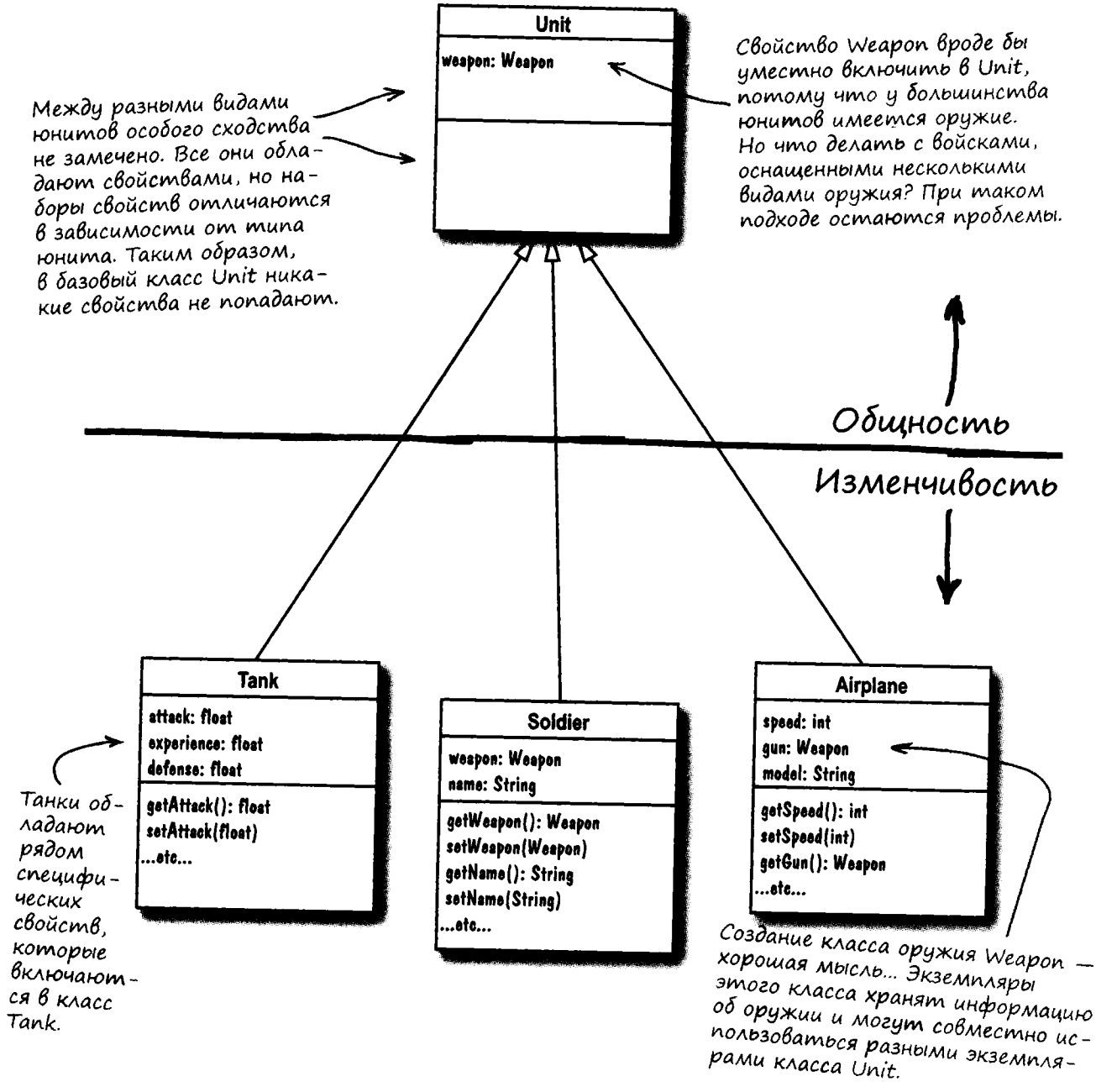
Какие из этих свойств присущи всем типам юнитов? Какие следует разместить в субклассах Unit для конкретных игр?





Решение № 1: Все разное!

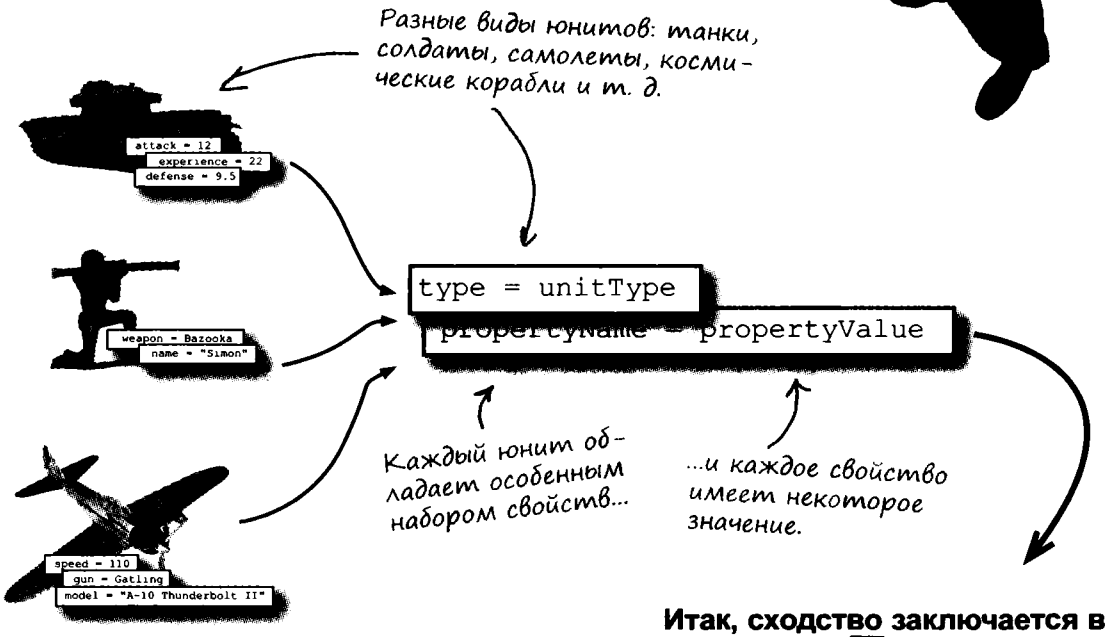
Первое решение, которое приходит в голову, выглядит примерно так:



Ерунда какая-то... Зачем возиться с этой общностью, если у разных видов войск нет ничего общего? Пустая трата времени.

Общность не сводится к именам свойств... Постарайтесь заглянуть поглубже.

На первый взгляд между юнитами, используемыми в разных играх, нет ничего общего, но не стоит ограничиваться именами свойств. Какие общие характеристики реально присутствуют у разных юнитов?

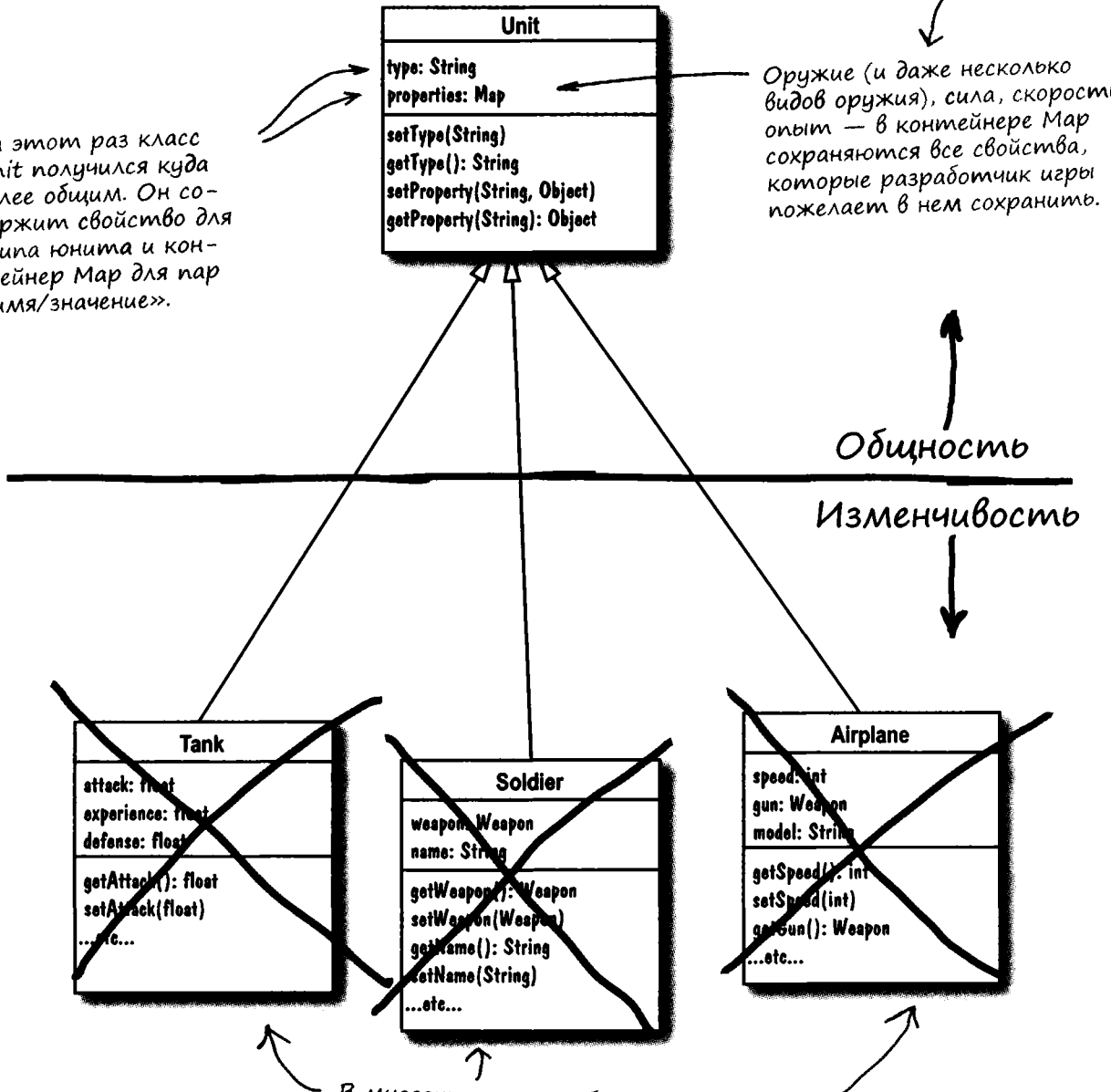


Итак, сходство заключается в том, что юниты имеют тип и набор свойств, каждое из которых представляет собой простую пару «имя/значение».

Решение № 2: Все одинаковое!

Еще одно решение, которое сразу приходит в голову, выглядит примерно так:

Очень похоже на то, как мы хранили свойства инструментов в главе 5.



Оружие (и даже несколько видов оружия), сила, скорость, опыт — в контейнере Map сохраняются все свойства, которые разработчик игры пожелает в нем сохранить.

На этот раз класс Unit получился куда более общим. Он содержит свойство для типа юнита и контейнер Map для пар «имя/значение».

В многочисленных субклассах Unit больше нет необходимости... Мы просто используем класс Unit с разными наборами свойств.

Это просто смешно. Сначала все было **разное**, а теперь все **одинаковое**? И как это поможет мне сократить риски или написать более качественный код?

Анализ общности: путь к гибкости кода

Вас интересует, зачем мы потратили столько времени на анализ общности? Посмотрите на первое решение на с. 382, потом на второе решение слева. Заполните следующую таблицу, чтобы понять, что дает нам общность в отношении структуры кода:

Количество типов юнитов	Количество классов юнитов — решение № 1	Количество классов юнитов — решение № 2
3		
5		
10		
25		
50		
100		

Первая строка относится к примеру, рассмотренному выше: 3 разных типа юнитов.

Может показаться, что 100 разных юнитов — это перебор, но в масштабных военных играх это не так уж далеко от действительности.

Как вы думаете, какое решение лучше? _____

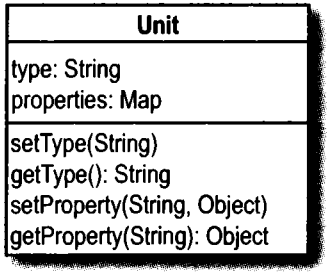
Почему? _____

Количество типов юнитов	Количество классов юнитов — решение № 1	Количество классов юнитов — решение № 2
3	4	1
5	6	1
10	11	1
25	26	1
50	51	1
100	101	1

С решением № 2 один класс Unit поддерживает все типы юнитов с разными наборами свойств и атрибутов.

В решении №1 всегда используется базовый класс Unit с субклассами для всех типов юнитов.

Мы выделили общие характеристики классов и поместили их в базовый класс Unit. Результат: разработчикам игр теперь приходится иметь дело с ОДНИМ классом юнитов вместо 25, 50 или 100!



Один хорошо спроектированный класс Unit позволяет поддерживать произвольное количество типов юнитов.

Хорошая структура кода всегда сокращает риск.

Часто задаваемые вопросы

В: Я вижу, почему это решение улучшает структуру кода, но как оно поможет с сокращением риска?

О: Хорошая структура кода всегда сокращает риск. Заранее продумав структуру класса `Unit`, мы сможем правильно реализовать его с первого раза... До того, как работа над игровой библиотекой будет в самом разгаре и нам вдруг потребуется вносить в `Unit` серьезные изменения, которые отразятся во многих местах по всей кодовой базе.

Мы не только разобрались, что понимается под «юнитами для конкретной игры», но и определили базовый класс Unit. Теперь другие классы, такие как Board, смогут работать с ним, не беспокоясь о возможных серьезных изменениях его структуры на середине или на завершающей стадии цикла разработки проекта.

Порядка стало еще больше...

Мы определили еще одну ключевую функциональную возможность и дополнительно сократили риски в проекте. В списке остался всего один пункт.

Игровая система
КЛЮЧЕВЫЕ функциональные возможности

1. ✓ Доска для игры — суть системы.
2. ✓ Юниты для конкретной игры — суть системы. Что это означает?
3. Координация перемещений: что это и как это сделать?

Погодите... Мы не не написали ни строки кода для класса Unit. Разве не нужно сделать это до того, как мы займемся последним пунктом?

Сейчас мы занимаемся только тем, что приведет к сокращению риска.

Помните: главная задача архитектуры – **сокращение риска** и **упорядочение**. В приложении есть много других аспектов, над которыми необходимо поработать, однако пункты списка – то, чем нужно заняться в первую очередь, когда вы будете примерно представлять себе структуру своего приложения, для сведения основных рисков до контролируемого уровня.

Мы постарались разобраться в структуре класса Unit и в смысле выражения «юниты для конкретной игры», и у нас это получилось:

Unit
type: String properties: Map
setType(String) getType(): String setProperty(String, Object) getProperty(String): Object

Эта диаграмма классов — то, что нам необходимо на данный момент. Она описывает структуру класса Unit и отвечает на вопрос: «Что такое "юниты для конкретной игры"»?

Часто Задаваемые Вопросы

В: Работая над Board, мы запрограммировали класс Board, а теперь вы говорите, что программировать класс Unit не нужно. Почему?

О: На этой стадии проекта всегда следует задавать себе один вопрос: «Сократит ли это риски для успешного завершения моего проекта?» Если ответ будет положительным, продолжайте; если отрицательным, задачу, вероятно, лучше оставить до более поздней стадии проекта.

В случае с Board было необходимо хотя бы в общих чертах понимать, как работает доска, поэтому мы запрограммировали базовую реализацию. Но для Unit оказалось достаточно диаграммы классов и понимания базовой функциональности. В обоих случаях нашей целью было сокращение рисков для проекта, а не программирование некоторого класса или пакета.

В: Тогда почему мы не могли построить диаграмму классов для Board, как это было сделано для Unit, и остановиться на этом?

О: Возможно, мы могли ограничиться диаграммой классов. Это вопрос субъективной оценки. Если вы чувствуете, что ваши действия направлены на сокращение рисков в проекте, ничто не мешает вам остановиться на диаграмме классов или же спуститься на уровень-два вглубь.

В: А разве правильно спрашивать у заказчика и пользователей, что должна делать система? Не могут ли они отвлечь нас от нужного направления?

О: Обычно в обращении к заказчику нет ничего плохого, ведь это его систему вы строите. Да, заказчик может вас запутать или направить по ошибочному пути, но только в том случае, если вы не понимаете, над чем работаете. Если вы вступаете в разговор, четко представляя свои цели, и прислушиваетесь к конкретной информации, вы сможете отфильтровать всю отвлекающую и несущественную информацию.

В: Не думаю, что я бы самостоятельно догадался до использования Map для хранения свойств класса Unit.

О: Это нормально; для этого и нужны такие инструменты, как общность и три основных вопроса архитектуры. Они помогут прийти к решениям, до которых вы бы не додумались самостоятельно, притом способом, подходящим для любого типа проектов. В случае с классом Unit важно не то, что мы использовали Map для хранения свойств. Важно понять, что все юниты по существу представляют собой базовый тип с набором пар «имя/значение». Когда это стало понятно, разобраться в подробностях реализации хранения пар «имя/значение» стало проще простого.

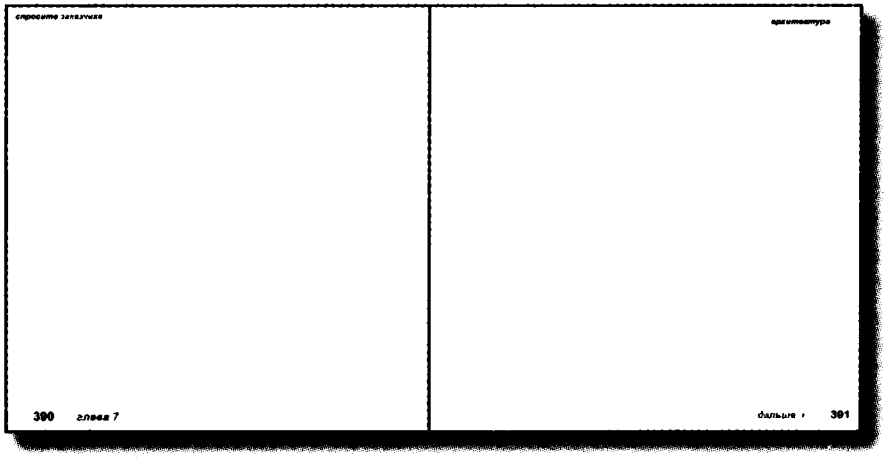
В: Получается, ООАП почти не имеет отношения к программному коду?

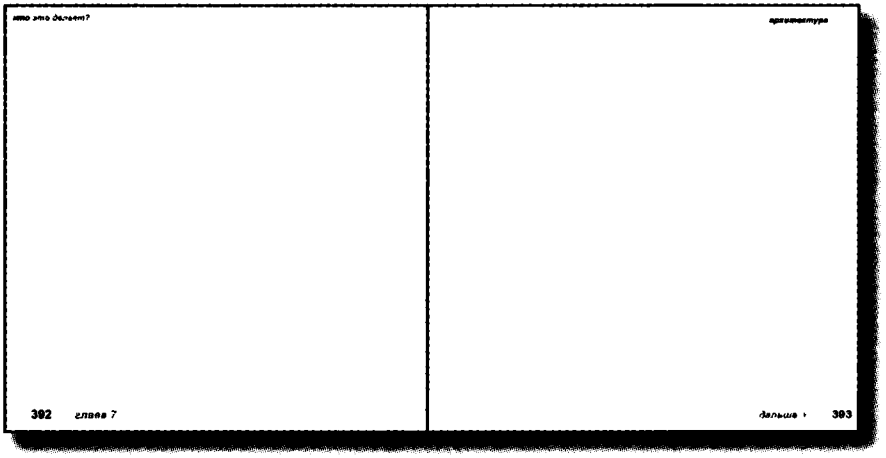
О: ООАП имеет отношение только к программному коду. Целью методологии является стабильное написание хороших программ. Однако чтобы создать хорошую программу, вы не всегда садитесь и пишете ее с первого раза. Иногда для создания хорошей программы написание кода лучше отложить на столько, на сколько возможно. Планирование, организация, структурирование, выяснение требований, сокращение рисков... После всего этого написать код будет уже несложно.

Иногда для создания хорошей программы написание кода лучше отложить на столько, на сколько это возможно.

Стань автором

Остается лишь координировать перемещения юнитов. Но что Вы сделаете теперь, чтобы определить эту ключевую функциональную возможность? Попробуйте подобрать материаал для следующих нескольких страниц книги. Как бы Вы подошли к реализации последней ключевой функциональной возможности?



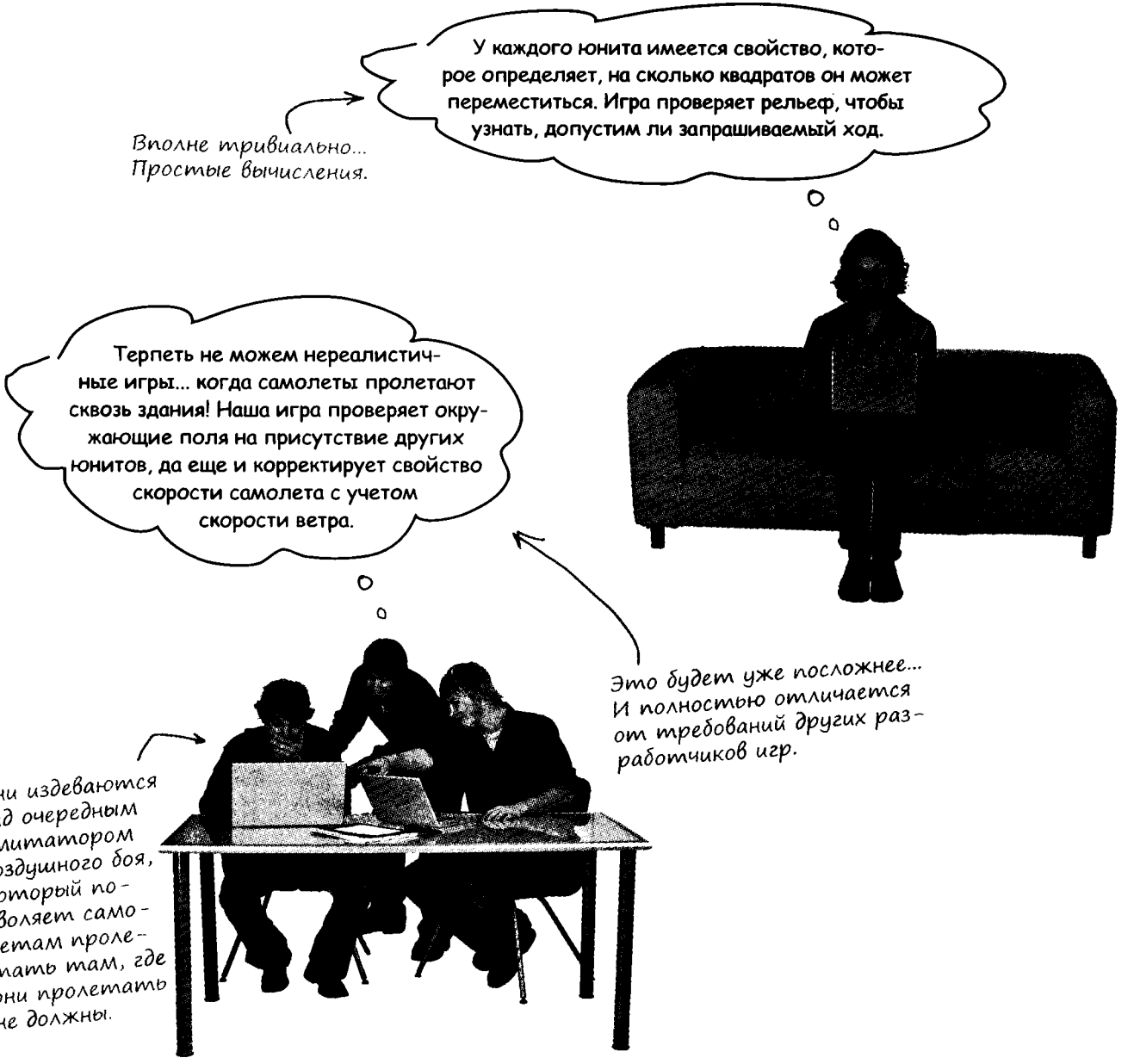


*Подсказка: вспомните, как мы поступили с предыдущей ключевой функциональной возможностью, когда было непонятно, с чего следует начинать.

Что это означает? Спросите заказчика.

Хотите узнать ответы на упражнение «Стань автором»? Прочитайте следующие четыре страницы и посмотрите, насколько они будут похожи на то, что получилось у вас.

Если вы не уверены в истинном значении той или иной функциональной возможности, самое разумное, что можно сделать, — задать вопрос заказчику. Мы поступили так с юнитами для конкретной игры, теперь давайте попробуем тем же способом узнать, что понимается под «координацией перемещений».



А вы знаете, что такое «координация перемещений»?

После общения с заказчиком вы должны достаточно хорошо понимать, на что похожа третья ключевая функциональная возможность игровой системы Гэри. Запишите, как вы себе ее представляете:

_____

_____

_____

_____

Проведите анализ общности

Далее необходимо определить, что общего у разных сценариев перемещения, о которых говорят заказчики на с. 390. Есть ли у них что-то общее, что применимо ко всем типам перемещений? Если вы считаете, что есть, запишите эти общие характеристики:

_____

_____

_____

_____

И что вам делать дальше?

Если вы поняли, что заказчик имеет в виду под «координацией перемещений», и знаете характеристики последней функциональной возможности, общие для всех игр, то вы должны достаточно хорошо представлять, что нужно сделать с игровой библиотекой для работы этой функциональной возможности. Запишите свои соображения:

_____

_____

_____

_____

Используйте эти три действия каждый раз, когда вы не уверены в смысле той или иной функциональной возможности или не знаете, как реализовать ее в своей системе.

1. Спросите заказчика

Что это означает?

2. Анализ общности

Как реализовать эту функциональную возможность в моей системе?

3. План реализации

Что здесь общего?

По высказываниям клиентов Гэри насчет перемещения мы сформулировали следующие принципы:

Юниты должны уметь перемещаться с одного поля доски на другое. Перемещение определяется расчетами или алгоритмом, специфическим для каждой игры, который иногда учитывает особые характеристики юнита.

Так что же именно общего между разнообразными сценариями перемещений? Помните, что говорили будущие пользователи?

Терпеть не можем нереалистичные игры... когда самолеты пролетают сквозь здания! Наша игра проверяет окружающие поля на присутствие других юнитов, да еще и корректирует свойство скорости самолета с учетом скорости ветра.

В этой ситуации применяется алгоритм, проверяющий допустимость перемещения, и другой алгоритм, определяющий дальность перемещения юнита (которое отчасти зависит от свойств этого юнита).

У каждого юнита имеется свойство, которое определяет, на сколько квадратов он может переместиться. Игра проверяет рельеф, чтобы узнать, допустим ли запрашиваемый ход.

В данном случае проверяется допустимость перемещения, а также значение свойства movement юнита.

Что остается общим?	Что изменяется?
Перед перемещением проверяется его допустимость.	Алгоритм проверки допустимости хода зависит от конкретной игры.
Для определения дальности перемещения используются свойства юнита.	Количество и набор свойств зависят от конкретной игры.
На перемещение могут влиять внешние факторы.	Факторы, влияющие на перемещение, зависят от конкретной игры.

Видите закономерность?

Здесь учитываются такие обстоятельства, как скорость ветра.

Это «...зависит от конкретной игры»

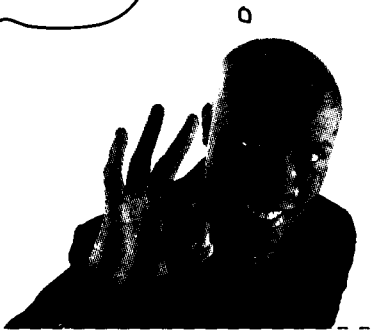
Вы заметили, что присутствует в каждом пункте нашего списка? Каждый раз, когда мы находим какую-то общность, в столбце изменчивости присутствуют одни и те же слова: «...зависит от конкретной игры».

Когда различий в функциональной возможности больше, чем сходства, такое решение не может считаться хорошим и универсальным.

Если в системе Гэри невозможно предоставить универсальное решение какой-либо задачи, эта задача не должна быть частью библиотеки.

Гэри, мы все продумали. Нужно предоставить разработчикам игр возможность самостоятельно реализовывать перемещение. Все, что мы сделаем для этого в библиотеке, только усложнит им жизнь.

Ладно, похоже, вы действительно все продумали, так что я не против. В конце концов, разработчики игр всегда предпочитают максимум контроля над реализацией.



Часто задаваемые вопросы

В: И чем это отличается от «юнитов для конкретной игры»?

О: С юнитами мы нашли некоторую общность: у каждого юнита есть тип и свойства «имя/значение». С перемещением реализация каждой игры отличалась от всех остальных. Поэтому было логично оставить перемещение на усмотрение разработчиков игр вместо того чтобы разрабатывать решение, которое было бы настолько общим, что никакой пользы от него все равно бы не было.

В: Но ведь есть и общие аспекты, верно? Алгоритм перемещения, проверка допустимости перемещения?

О: Да, есть. Так что теоретически мы могли написать интерфейс `Movement` с методом типа `move()`, который получал бы аргументы `MovementAlgorithm` и `LegalMoveCheck` или что-нибудь аналогичное. А потом каждый разработчик игры должен бы был расширять `MovementAlgorithm` и `LegalMoveCheck`. Если вам представилось что-нибудь в этом роде, поздравляем! Вы схватываете все на лету.

Но спросите себя: что это реально дает? Разработчики игр должны будут изучать ваши интерфейсы, и если проверка допустимости в игре не используется, они могут передать в прааметре `LegalMoveCheck` значение `null`... А как будет выглядеть интерфейс `MovementAlgorithm`?.. Пожалуй, вы только добавляете сложность, вместо того чтобы сокращать ее.

Ваша задача — сокращение риска и сложности, а не их увеличение. Мы решили, что будет проще доверить перемещение разработчикам игр и просто изменять положение юнитов на доске (с использованием методов Board, которые мы реализовали за них).

Замечательно. Теперь у нас есть бумажка с пометками, несколько незаконченных классов и много UML-диаграмм. И вы хотите меня убедить, что именно так пишутся хорошие программы?

Заказчики платят не за оригинальный код, а за хорошие программы.

Да! Не забывайте, что хорошие программы — не только хороший код.

Хороший код хорошо спроектирован и обычно работает так, как ему положено. Но хорошие программы не только хорошо спроектированы, они выпускаются вовремя и делают то, чего от них хочет заказчик.

В этом и заключаются цели архитектуры: сокращение рисков того, что продукт будет сдан с нарушением сроков или будет делать не то, чего хочет заказчик. Наши списки ключевых функциональных возможностей, диаграммы классов, незаконченные классы — все это помогает убедиться в том, что мы создаем не только хороший код, но и хорошую программу.

Сокращение рисков помогает писать хорошие программы

Разобравшись со всеми тремя ключевыми функциональными возможностями, мы взяли под контроль основные риски для успеха нашего проекта. Посмотрим, как каждый из шагов, описанных в этой главе, способствовал сокращению рисков для проекта:

Игровая система

Ключевые функциональные возможности

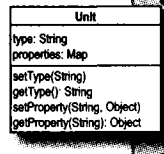
1. Доска для игры — суть системы.
2. Юниты для конкретной игры — суть системы. Что это означает?
3. Координация перемещений: что это и как это сделать?

С этого все началось. Мы только примерно знали, что нам нужно построить... и не более того.

Мы определили основные классы Board, но написали объем кода, достаточный только для того, чтобы сократить риски ошибочной реализации доски.

Unit.java
Tile.java
Board.java

Затем мы разобрались, что означает требование «юниты для конкретной игры», и спланировали реализацию этой функциональной возможности с использованием диаграммы классов.



Наконец, при помощи принципа общности мы поняли, что реализацию перемещений лучше оставить разработчикам игр... Тем самым мы избавились еще от одного серьезного риска.

Огромный Риск-О-Метр

Ни при каких условиях не уложимся в срок.

Один шанс из ста, что все будет сделано правильно.

Серьезных проблем будет не так уж много.

Уверены в том, что все будет сделано вовремя!

Кода пока не так уж много, но мы можем быть уверены в том, что проект будет сдан вовремя и с правильной функциональностью.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Архитектура помогает преобразовать диаграммы, планы и списки функциональных возможностей в четко организованное приложение.
- Функциональные возможности системы, имеющие наибольшую важность для проекта, обладают архитектурной значимостью.
- В первую очередь займитесь функциональными возможностями, которые относятся к сути вашей системы, смысл которых вам недостаточно понятен или которые вы не знаете, как реализовать.
- Все, что вы делаете на архитектурных стадиях проекта, должно вести к сокращению рисков для успешного завершения проекта.
- Если все подробности варианта использования вам не нужны, для быстрого сбора требований можно написать сценарий с описанием использования вашего продукта.
- Если вы не уверены в том, что собой представляет та или иная функциональная возможность, спросите у заказчика, а потом постарайтесь обобщить полученные ответы для более глубокого понимания.
- Используйте анализ общности для построения гибких программных решений.
- Заказчики заинтересованы в том, чтобы программа делала то, что им нужно, и была сдана вовремя, а не в том, чтобы написанный вами код выглядел (по вашему мнению) круто.

Возьми в руку карандаш
Решение

Чего не хватает в Board.java?

Присмотритесь повнимательнее к сценарию на предыдущей странице. Охватывают ли требования на с. 364 всё в законченном сценарии? Если вы считаете, что чего-то не хватает, добавьте описание внизу и включите в Board.java код, реализующий недостающую функциональность.

В тексте говорится об удалении юнитов, но в требованиях на с. 364 об удалении ничего не сказано.

Для реализации этого требования мы добавили в Board.java методы removeUnit() и removeUnits().

Не стремитесь к оригинальности

Имитация — самая искренняя форма проявления ума. Ничто не доставляет столько удовлетворения, как создание совершенно нового, оригинального решения задачи, которая не давала вам покоя несколько дней. Пока не выяснится, что кто-то уже решил ту же задачу задолго до вас, да еще и сделал это лучше вас! В этой главе рассматриваются принципы проектирования, которые были созданы за прошедшие годы, и то, как они помогают вам стать более квалифицированным программистом. Откажитесь от мысли «сделать по-своему», в этой главе вы узнаете, **как сделать умнее и быстрее.**

Кратко о принципах проектирования

До сих пор наше внимание было сосредоточено на том, что делается до начала написания кода. Сбор требований, анализ, построение списков функциональных возможностей, рисование диаграмм вариантов использования... Конечно, в какой-то момент вам придется заняться написанием кода. И здесь-то в игру вступают принципы проектирования.

***Принцип проектирования** — общий инструмент или прием, который может применяться при проектировании или написании кода для улучшения гибкости, удобства сопровождения и расширения программы.*

Принципы проектирования уже встречались вам в предыдущих главах:

ОО-принципы

Инкапсулируйте то, что изменяется.

Программируйте для интерфейса, а не для реализации.

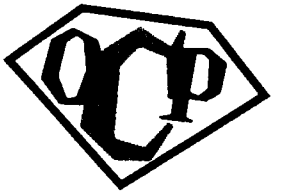
Каждый класс приложения должен иметь только одну причину для изменения.

Классы создаются ради поведения и функциональности.

В этой главе мы рассмотрим еще несколько ключевых принципов и их положительное влияние на структуру и реализацию кода. Вы увидите, что в некоторых ситуациях приходится выбирать между двумя принципами проектирования... Впрочем, мы забегаем вперед. Давайте начнем с первого принципа проектирования.

Применение проверенных принципов ОО-проектирования улучшает гибкость, удобство сопровождения и расширения программы.

Принцип № 1: принцип открытости/закрытости



Наш первый принцип проектирования — принцип открытости/закрытости (OCP, Open-Closed Principle). Он **допускает изменения**, но только те, которые предотвращают необходимость переписывания существующего кода. Обычно мы определяем OCP следующим образом:

Принцип открытости/закрытости

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

Классы закрываются, чтобы никто не мог изменить ваш работающий код.

Закрыто для изменения...

Предположим, у вас имеется класс с некоторым поведением, и вы хотите запрограммировать это поведение так, как вам нужно. Позаботьтесь о том, чтобы никто другой не мог изменить код класса, и это поведение станет ***закрытым для изменения***. Иначе говоря, никто не сможет изменить это поведение, потому что вы зафиксировали его в классе, который заведомо не будет изменяться.

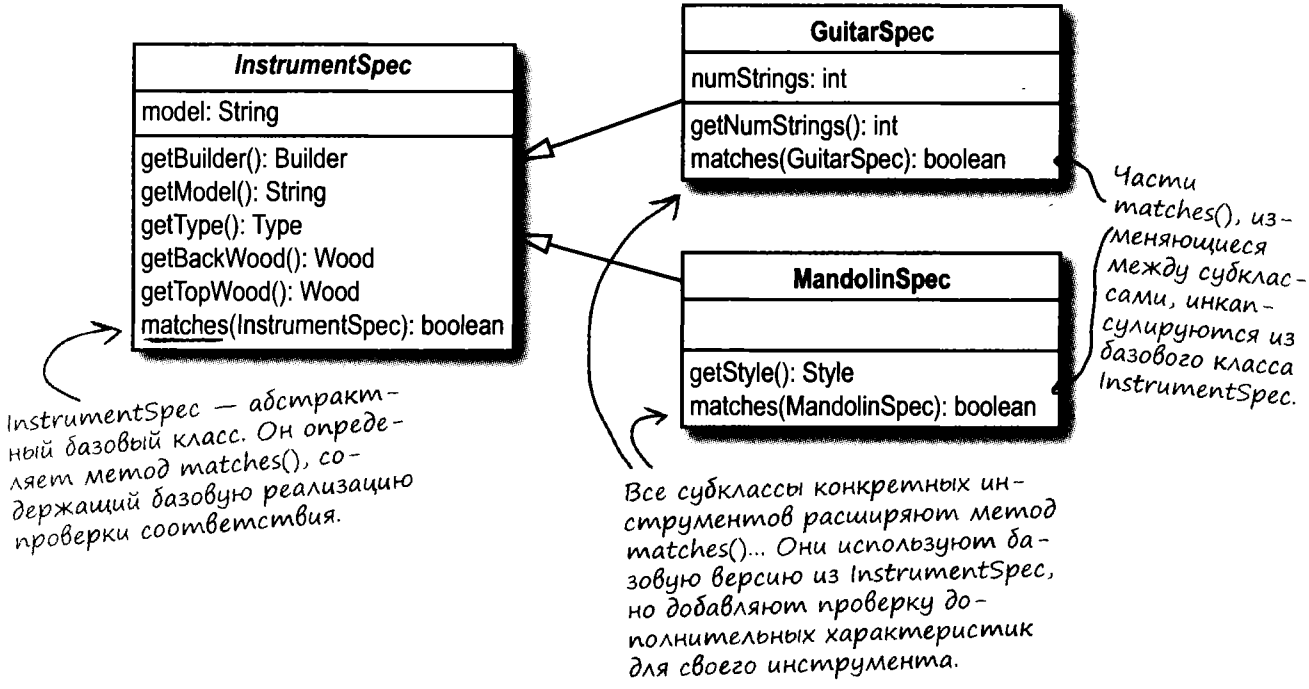
...но открыто для расширения

А потом кому-то потребовалось это поведение изменить. Вы не хотите, чтобы он возился с вашим идеальным кодом, который прекрасно работает... почти всегда. Но при этом вы хотите, чтобы ваш код можно было использовать и расширять. Вы разрешаете субклассирование, и внешние пользователи переопределяют ваш метод, чтобы он лучше подходил для их целей. Итак, хотя работающий код не изменился, вы оставили свой класс ***открытым для расширения***.

Классы открываются, чтобы их можно было субклассировать и расширять.

Помните нашу работу над каталогом музыкальных инструментов?

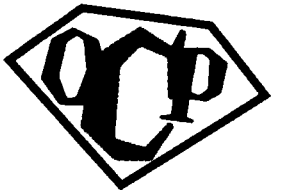
Вероятно, вы этого не заметили, но мы использовали принцип открытости/закрытости при написании классов InstrumentSpec для каталога Рика в главе 5:



Класс InstrumentSpec закрыт для изменения; метод matches( ) определяется в базовом классе и не изменяется.

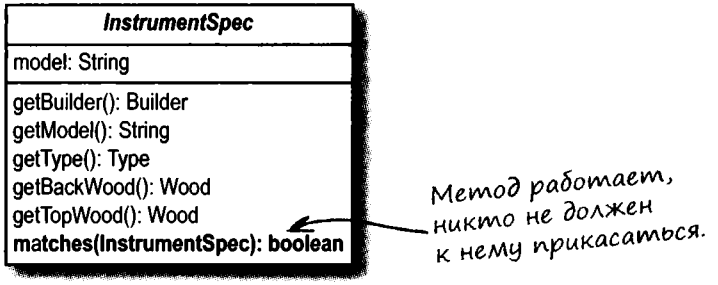
Но при этом он открыт для расширения, потому что любой субкласс может изменить поведение matches( ).

OCP шаг за шагом



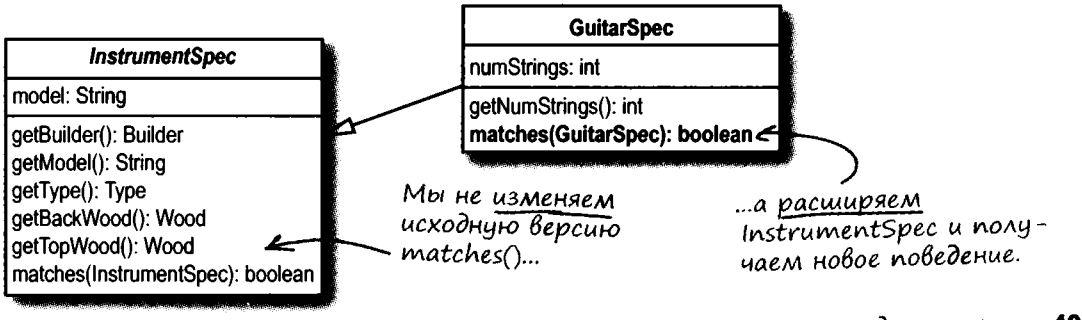
Давайте вернемся к тому, что мы сделали в главе 5, и последовательно рассмотрим происходящее в контексте OCP.

1. **Мы запрограммировали метод matches() в InstrumentSpec.java и закрыли его для изменения.**
   Эта версия `matches()` прекрасно работает, и мы не хотим, чтобы кто-нибудь с ней экспериментировал. Другими словами, после того, как мы запрограммируем `InstrumentSpec` и эту версию `matches()`, они не должны изменяться.



2. **Но нам понадобилось изменить modify matches() для классов конкретных инструментов.**
   Хотя метод `matches()` прекрасно работает с другими объектами `InstrumentSpec`, для гитар и мандолин он делает не совсем то, что нужно. Хотя метод `matches()` закрыт для корректировки, нам понадобится механизм его расширения и изменения... Иначе класс `InstrumentSpec` окажется негибким, а это плохо.

3. **И тогда мы расширили класс InstrumentSpec и переопределили matches() для изменения его поведения.**
   Изменение кода InstrumentSpec недопустимо, но мы можем расширить его в классах `GuitarSpec` и `MandolinSpec`, а затем переопределить `matches()` во всех этих классах для добавления поведения, присущего конкретным инструментам.



Главное содержание OCP — гибкость — выходит за рамки простого наследования.

Безусловно, наследование является простым примером принципа открытости/закрытости, но здесь речь идет о чем-то большем, чем простое субклассирование и переопределение метода. Каждый раз, когда вы пишете рабочий код, следует приложить максимум усилий к тому, чтобы этот код *оставался рабочим*... А это означает, что он не должен изменяться другими людьми.

Но в будущем может возникнуть необходимость в адаптации этого кода, возможно, для одной-двух особых ситуаций. Вместо того чтобы заставлять вас погружаться в код и вносить множество изменений, OCP позволяет ***расширить*** рабочий код без его ***изменения***.

Задача может решаться разными способами. И хотя наследование чаще всего оказывается самым простым в реализации, безусловно, это не единственный вариант. Другое отличное решение будет представлено позднее в этой главе, когда речь пойдет о композиции.

Возьми в руку карандаш

Найдите следы OCP в своем проекте.

Подумайте над проектом, над которым вы работаете в настоящее время. Найдете ли вы в нем места, в которых вы уже использовали OCP? Если найдете, опишите, как вы использовали OCP:

_____

_____

_____

_____

Теперь подумайте, нет ли в проекте места, в котором следовало бы использовать принцип открытости/закрытости (но вы еще его не использовали). Запишите, что, по вашему мнению, необходимо сделать для того, чтобы применить принцип OCP в вашем текущем проекте:

_____

_____

_____

_____

Часто задаваемые вопросы

В: А что плохого в изменении кода уже написанного класса?

О: Если у вас имеется класс, который работает и активно используется, вносить изменения в него без необходимости не стоит. Но помните: ИЗМЕНЕНИЕ — единственная великая константа в разработке ПО. С принципом OCP мы разрешаем изменение посредством расширения (вместо модификации уже написанного кода). Субклассы могут дополнять и расширять поведение базового класса без возни с кодом, который заведомо работает и устраивает заказчика.

В: Разве принцип OCP не является разновидностью инкапсуляции?

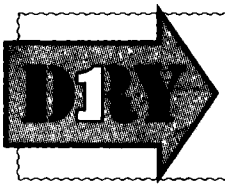
О: Скорее это сочетание *инкапсуляции* и *абстракции*. Вы находите поведение, которое остается неизменным, абстрагируете его в базовый класс, а потом закрываете этот код от изменений. Когда вам понадобится новое или другое поведение, изменения реализуются в субклассах, расширяющих базовый класс. Здесь и проявляется инкапсуляция: то, что изменяется (поведение в субклассах), отделяется от того, что остается неизменным (общее поведение в базовом классе).

В: Единственный способ использования OCP — расширение другого класса?

О: Нет. В любой ситуации, в которой ваш код закрыт для изменения, но открыт для расширения, используется принцип OCP. Например, если класс содержит несколько приватных методов, эти методы закрыты для изменения — другой код не сможет их использовать. Но при этом в класс включаются открытые методы, которые вызывают эти приватные методы. Таким образом, вы расширяете поведение приватных методов без их изменения. Это еще один пример практической реализации OCP.

Принцип № 2: не повторяйтесь

Далее идет принцип «не повторяйтесь», часто называемый DRY (сокращение от Don't Repeat Yourself). Этот принцип тоже выглядит очень просто, но он играет важнейшую роль в написании кода, простого в сопровождении и повторном использовании.



Не повторяйтесь

Избегайте дублирования кода, абстрагируя общие аспекты и размещая их в одном месте.

Где применяется принцип DRY

Вы уже видели принцип DRY в действии, даже если и не осознали этого. Он использовался нами в главе 2, когда Тодд и Джина захотели, чтобы открывшаяся дверь автоматически закрывалась.

```
public void pressButton() {
  System.out.println(
    "Pressing the remote control button...");
  if (door.isOpen()) {
    door.close();
  } else {
    door.open();

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      public void run() {
        door.close();
        timer.cancel();
      }
    }, 5000);
  }
}
```

Помните: код класса Remote автоматически закрывал открывшуюся дверь?

Remote.java

Дуг предложил включить тот же код в BarkRecognizer... Но принцип DRY говорит, что это ПЛОХАЯ идея.

```
public void recognize(String bark) {
  System.out.println("   BarkRecognizer: " +
    "Heard a '" + bark + "'");
  door.open();

  final Timer timer = new Timer();
  timer.schedule(new TimerTask() {
    public void run() {
      door.close();
      timer.cancel();
    }
  }, 5000);
}
```

BarkRecognizer.java

1. Абстрагируем общий код.

В соответствии с принципом DRY сначала следует выделить код, общий для **Remote** и **BarkRecognizer**, и разместить его в одном месте. Как мы определили в главе 2, лучшим местом для размещения этого кода является класс **DogDoor**:

Используя принцип DRY, мы извлекаем этот код из Remote и BarkRecognizer и размещаем его в ОДНОМ месте: в классе DogDoor. Нет дублирующегося кода — нет кошмаров сопровождения.

```
public class DogDoor {
  public void open() {
    System.out.println("The dog door opens.");
    open = true;

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      public void run() {
        close();
        timer.cancel();
      }
    }, 5000);
  }
```

DogDoor.java

2. Теперь удаляем код из других мест...

3. ...и обращаемся к коду из шага № 1.

Следующие два шага происходят одновременно. Удалите весь код, который был размещен в одном месте на шаге № 1, а затем при необходимости включите в программу явную ссылку на абстрагированный код:

Сначала удаляем этот фрагмент... Теперь все находится в методе open() класса DogDoor.

```
  public void recognize(String bark) {
    System.out.println("   BarkRecognizer: " +
      "Heard a '" + bark + "'");
    door.open();

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      public void run() {
        door.close();
        timer.cancel();
      }
    }, 5000);
  }
```

Включать явные обращения к абстрагированному коду в данном случае не нужно... Это происходит автоматически при вызове door.open().

BarkRecognizer.java

DRY: ОДНО требование в ОДНОМ месте

Абстрагирование дублирующегося кода — хорошее начало для использования DRY, но этим дело не ограничивается. Стремясь в исключению дублирующегося кода, вы в действительности стремитесь к тому, чтобы каждое требование и каждая функциональная возможность вашего приложения реализовались только один раз.

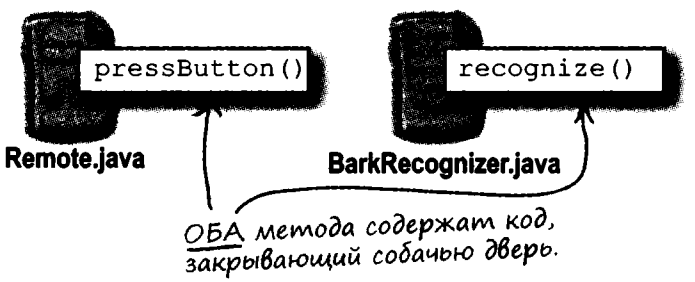
В уже упоминавшемся примере с собачьей дверью мы реализовали функциональную возможность автоматического закрытия двери.

Собачья дверь для Тодда и Джины, версия 2.0
Список требований

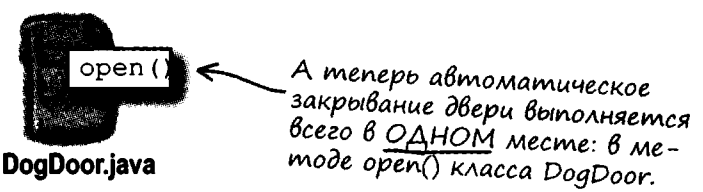
1. Высота дверного проема должна быть не менее 12 дюймов.
2. Кнопка на пульте открывает дверь, если она закрыта, и закрывает ее, если дверь открыта.
3. После того, как дверь откроется, она должна автоматически закрываться (если не была закрыта до этого).

Одно требование, на котором мы сосредоточились.

В исходной версии эта функциональная возможность была реализована в двух местах: `Remote.java` и `BarkRecognizer.java`.



Применение принципа DRY позволило исключить дублирующийся код. Но что еще важнее, реализация требования (автоматического закрывания двери) теперь находится в *одном* месте, а не в двух:



Часто задаваемые вопросы

В: Так сутью принципа DRY не является борьба с дублированием кода и копированием/вставкой?

О: Принцип DRY направлен не только на борьбу с дублированием кода, но и на то, чтобы эта борьба не создавала новых проблем. Его суть не сводится к тому, чтобы упаковать в один класс код, встречающийся в нескольких местах, вы должны позаботиться о том, чтобы каждый блок информации и поведения в системе существовал в одном определенном месте. Тогда ваша система всегда будет знать, куда следует обращаться, если понадобится этот блок информации или поведения.

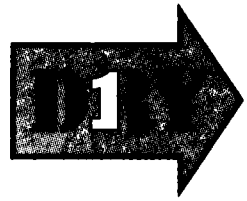
В: Если принцип DRY относится к функциональным возможностям и требованиям, не должен ли он применяться не только к написанию кода, но и к сбору этих функциональных возможностей и требований?

О: Безусловно, отличная идея! И в ходе написания требований, и при разработке вариантов использования, и при программировании необходимо следить за тем, чтобы в системе ничего не дублировалось. Каждое требование должно быть реализовано один раз, варианты использования не должны перекрываться, а код не должен повторяться. Действие принципа DRY распространяется не только на написание кода.

В: И все это для того, чтобы позднее избежать проблем с сопровождением кода?

О: Правильно. Но дело даже не сводится к тому, чтобы избежать необходимости обновления кода в нескольких местах. Помните: суть принципа DRY заключается в наличии единственного источника для некоторого блока информации или поведения. Но к формированию этого единственного источника следует подходить разумно! Ведь вы не захотите, чтобы распознаватель лая был единственным источником для закрытия двери, правда? Дверь же не должна обращаться к распознавателю с запросом на закрытие себя?

Таким образом, принцип DRY не сводится к устранению дубликатов. Он также направлен на принятие разумных решений по поводу разбиения функциональности вашей системы.



Суть принципа DRY в том, что каждый блок информации и поведения в системе должен существовать в одном разумно выбранном месте.

Головоломка

Принцип DRY не сводится к выявлению дублирующегося кода в системе, он применяется к функциональным возможностям и требованиям. Начнем использовать DRY на практике, и не только на уровне программного кода.

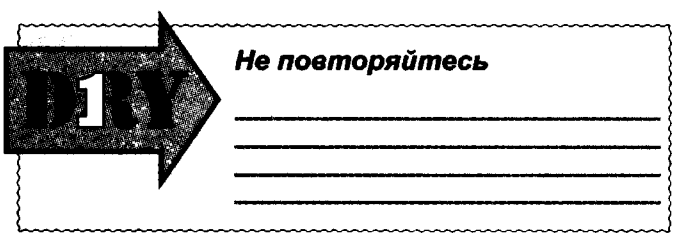
Проблема:

Тодд и Джина представили новый список пожеланий. Позаботьтесь о том, чтобы список не содержал дубликатов, а каждая функциональная возможность была реализована в проектируемой системе ровно один раз.

Ваша задача:

1. Внимательно прочитайте список справа. Требования и функциональные возможности, добавленные с момента нашей последней работы над дверью, выделены жирным шрифтом.
2. Посмотрите на новые функциональные возможности и требования. Не найдется ли среди них дубликатов?
3. Пометьте возможные дубликаты в пунктах списка.
4. Запишите дублирующиеся требования в конце списка, но сформулируйте их так, чтобы предотвратить дублирование.
5. Запишите свою версию нового определения принципа DRY. Проследите за тем, чтобы оно не ограничивалось дублированием одного лишь программного кода.

Запишите свое определение принципа DRY, включающее то, что мы узнали на нескольких последних страницах.



Собачья дверь для Тодда и Джины, версия 3.0

Список требований и функциональных возможностей

1. Высота дверного проема должна быть не менее 12 дюймов.
2. Кнопка на пульте открывает дверь, если она закрыта, и закрывает ее, если дверь открыта.
3. После того, как дверь откроется, она должна автоматически закрываться (если не была закрыта).
4. Распознаватель лая должен уметь определять, когда лает собака.
5. Услышав лай, распознаватель должен открыть собачью дверь.
6. Если что-то в доме приблизится к двери на минимальное расстояние, дверь должна сообщить об этом хозяину, чтобы не толкнуть объект при открытии.
7. Дверь должна открываться в определенные часы.
8. Дверь должна быть интегрирована в систему безопасности дома, чтобы сигнализация не срабатывала при открытии и закрытии двери.
9. Собачья дверь должна давать звуковой сигнал, если внешнее препятствие мешает открытию.
10. Дверь должна подсчитывать, сколько раз собака входит и выходит из дома.
11. При закрывании двери система безопасности включается, если она была активна до открытия двери.

Это вы уже видели...

...а это новые требования и функциональные возможности.

Запишите внизу новые или обновленные требования, не содержащие дублирования.

Ответы на следующей странице.

Головоломка. Решение

Принцип DRY не сводится к выявлению дублирующегося кода в системе, он применяется к функциональным возможностям и требованиям. Начнем использовать DRY на практике, и не только на уровне программного кода.

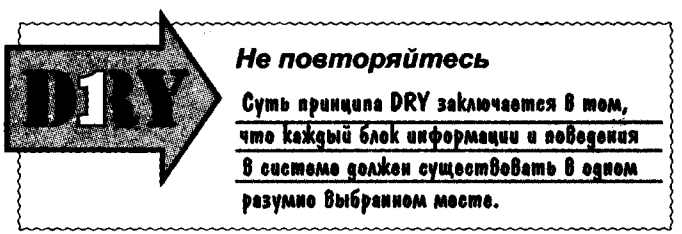
Проблема:

Тодд и Джина представили новый список пожеланий. Позаботьтесь о том, чтобы список не содержал дубликатов, а каждая функциональная возможность была реализована в проектируемой системе ровно один раз.

Ваша задача:

1. Внимательно прочитайте список справа. Требования и функциональные возможности, добавленные с момента нашей последней работы над дверью, выделены жирным шрифтом.
2. Посмотрите на новые функциональные возможности и требования. Не найдется ли среди них дубликатов?
3. Пометьте возможные дубликаты среди пунктов списка.
4. Запишите дублирующиеся требования в конце списка, но сформулируйте их так, чтобы предотвратить дублирование.
5. Запишите свою версию нового определения принципа DRY. Проследите за тем, чтобы оно не ограничивалось дублированием одного лишь программного кода.

Определение DRY, которое написали мы:



Собачья дверь Тодда и Джины, версия 3.0

Список требований и функциональных возможностей

1. Высота дверного проема должна быть не менее 12 дюймов.
2. Кнопка на пульте открывает дверь, если она закрыта, и закрывает ее, если дверь открыта.
3. После того, как дверь откроется, она должна автоматически закрываться (если не была закрыта).
4. Распознаватель лая должен уметь определять, когда лает собака.
5. Услышав лай, распознаватель должен открыть собачью дверь.
6. ~~Если что-то в доме приблизится к двери на минимальное расстояние, дверь должна сообщить об этом хозяину, чтобы не толкнуть объект при открытии.~~
7. Дверь должна открываться в определенные часы.
8. Дверь должна быть интегрирована в систему безопасности дома, чтобы сигнализация не срабатывала при открытии и закрытии двери.
9. ~~Собачья дверь должна давать звуковой сигнал, если внешнее препятствие мешает открытию.~~
10. Дверь должна подсчитывать, сколько раз собака входит и выходит из дома.
11. При закрывании двери система безопасности включается, если она была активна до открытия двери.

Дверь сообщает хозяину, если снаружи или внутри дома имеется препятствие, мешающее ее работе.

Когда дверь открывается, сигнализация отключается, а при закрытии двери она снова включается (если была активна ранее).

В список требований были внесены некоторые изменения.

Пункты 6 и 9 почти идентичны: в одном отслеживаются внутренние препятствия, в другом внешние, но базовая функциональность почти одинакова.

Пункты 8 и 11 относятся к домашней сигнализации... По сути, они тоже являются дубликатами одной базовой функциональности

Требования 7 и 10 остались неизменными.

Так мы объединили и переписали пункты 6 и 9.

Новое требование, сформулированное на базе пунктов 8 и 11.

Принцип № 3: принцип единственной обязанности

Принцип единственной обязанности, или SRP (Single Responsibility Principle), определяет, что какие объекты делают в системе. Каждый спроектированный вами объект должен иметь только одну обязанность, и если в этой обязанности что-то изменится, вы будете точно знать, где следует вносить изменения в ваш код.

Принцип единственной обязанности

Каждый объект в системе должен иметь единственную обязанность, и все действия объекта должны быть сосредоточены на ее выполнении.

Принцип единственной обязанности реализован правильно, если каждый из ваших объектов имеет только одну причину для изменения.

Часто задаваемые вопросы

В: Принцип SRP кажется мне очень похожим на DRY. В обоих случаях речь идет о классе, который делает что-то одно, что ему положено делать. Разве не так?

О: Да, эти принципы связаны и часто используются вместе. Принцип DRY определяет, что блок функциональности должен находиться в одном месте, например в классе; принцип SRP указывает, что класс должен делать что-то одно, и делать это хорошо.

В хороших приложениях один класс делает что-то одно, делает это хорошо, и при этом его поведение не дублируется в других классах.

В: Разве требование «класс должен делать что-то одно» не является ограничением?

О: Нет, если учесть, что «что-то одно» может быть довольно большим. Например, класс `Board` в игровой системе Гэри выполняет множество мелких задач, но все они относятся к одной большой задаче: управлению доской в игре. Класс Board делает только это, и ничего более; это хороший пример применения SRP.

В: И с принципом SRP мои классы становятся меньше, потому что они будут делать что-то одно, верно?

О: Вообще-то с принципом SRP ваши классы часто становятся больше. Так как функциональность не распределяется среди многих классов (как поступают многие программисты, не знакомые с SRP), в одном классе часто размещается больше кода.

Однако применение SRP часто приводит к уменьшению количества классов, а от этого обычно существенно упрощается управление и сопровождение приложения.

В: Формулировка SRP очень похожа на определение сцепления. Это одно и то же?

О: Сцепление в действительности является синонимом SRP. Если вы пишете приложения с высоким уровнем сцепления, это означает, что вы правильно применяете SRP.

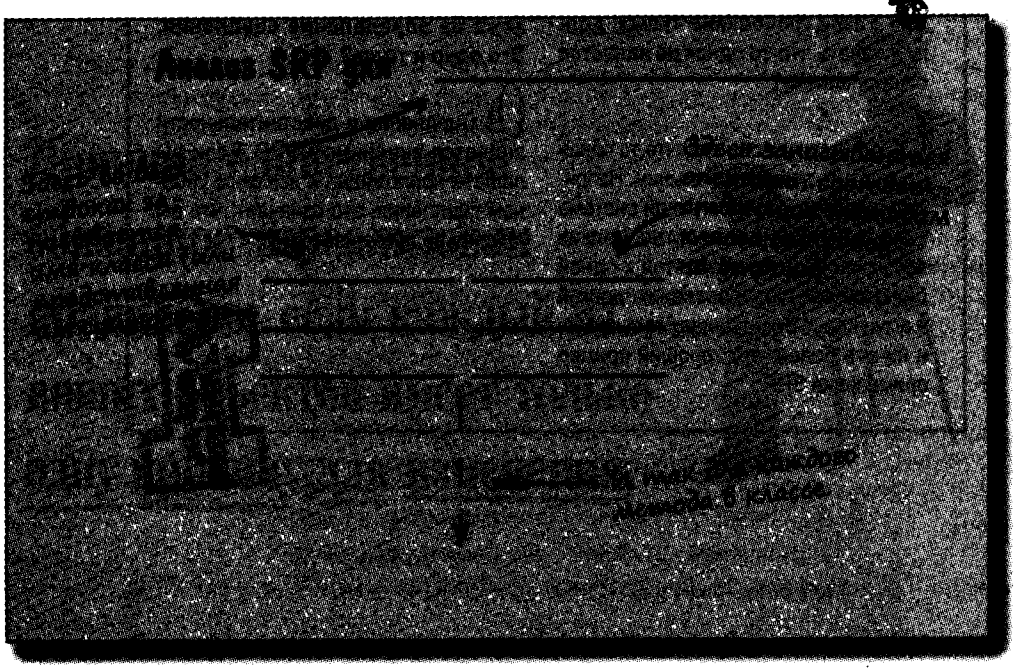
Выявление множественных обязанностей

В большинстве случаев для обнаружения классов, не использующих SRP, можно задействовать простую проверку:

1. Запишите на листке бумаги набор строк вида [имя класса] [операция, выполняемая с объектом класса] – по одной строке для каждого метода класса, проверяемого на соблюдение принципа SRP.
2. Прочитайте каждую строку вслух (при необходимости добавьте связки или слова, чтобы текст читался нормально). Имеет ли смысл то, что вы прочитали? Действительно ли у вашего класса есть обязанности, обозначенные этим методом?

Если то, что вы прочитали, не имеет смысла, вероятно, метод нарушает принцип SRP. Возможно, метод должен принадлежать другому классу... Рассмотрите возможность его перемещения.

Так должен выглядеть листок для анализа SRP.



Возьми в руку карандаш

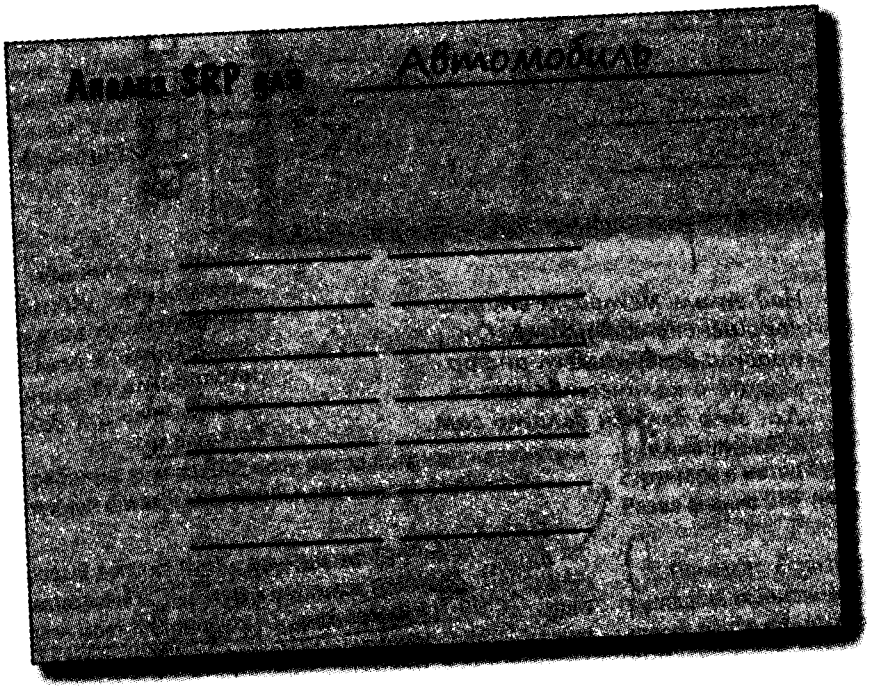
Примените принцип SRP к классу Automobile.

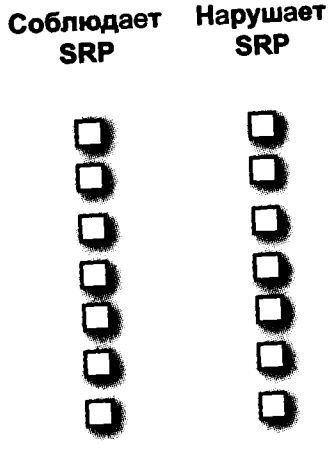
Проведите анализ SRP для класса Automobile, приведенного ниже. Заполните список методами класса Automobile. Решите, имеет ли смысл присутствие каждого метода в классе Automobile, и расставьте галочки.

Этот класс уже встречался нам в главе 5.

Automobile
start()
stop()
changeTires(Tire [*])
drive()
wash()
checkOil()
getOil(): int

Да, вы можете смухлевать и подсмотреть методы в главе 5, но мы доверяем вам и надеемся, что вы этого не сделаете. Попробуйте решить упражнение самостоятельно и только потом обратитесь к главе 5, если у вас ничего не получится.





Если какая-либо строка получается бессмысленной, то, вероятно, метод в этой строке нарушает SRP.

Возьми в руку карандаш
Решение

Примените принцип SRP к классу Automobile.

Проведите анализ SRP для класса Automobile, приведенного ниже. Заполните список методами класса Automobile. Решите, имеет ли смысл присутствие каждого метода в классе Automobile, и расставьте галочки.

Логично, что запуск и остановка двигателя являются обязанностью самого автомобиля.

Смена шин, мытье и проверка масла НЕ относятся к обязанностям автомобиля.

	Соблюдает SRP	Нарушает SRP
	☑	☐
	☑	☐
	☐	☑
	☐	☑
	☐	☑
	☐	☑
	☑	☐

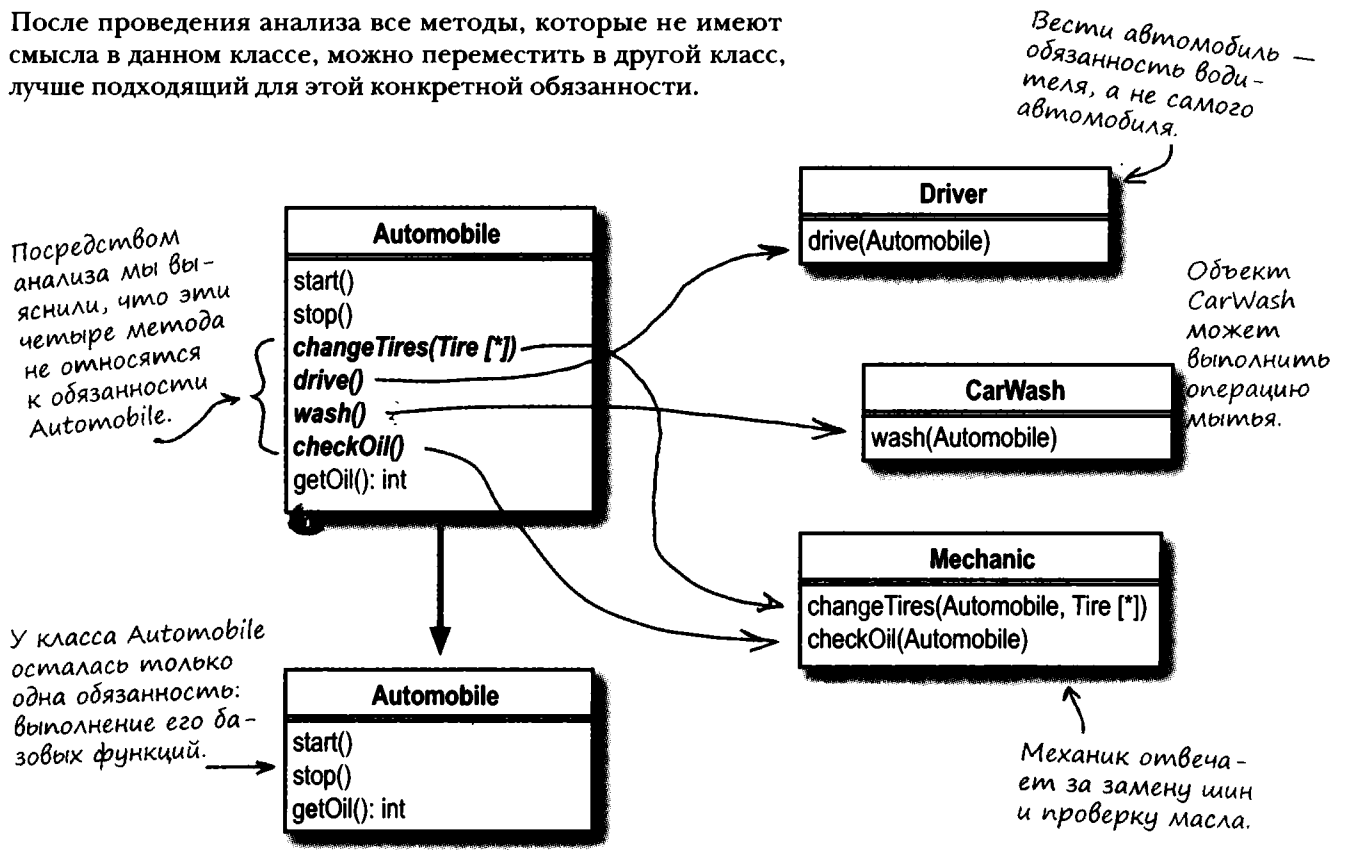
Над этим методом стоило хорошенько подумать. Он просто возвращает информацию о количестве масла... Да, это должен делать сам автомобиль.

Здесь небольшая тонкость... Хотя автомобиль может запускать и останавливать двигатель, управлять им должен водитель.

Подобные случаи наглядно показывают, почему анализ SRP — всего лишь рекомендация, а не набор жестких правил. Некоторые решения все равно придется принимать, руководствуясь здравым смыслом и собственным опытом.

От множественных обязанностей к единственной

После проведения анализа все методы, которые не имеют смысла в данном классе, можно переместить в другой класс, лучше подходящий для этой конкретной обязанности.



Часто задаваемые вопросы

В: Как работает анализ SRP, если метод получает параметры как в ситуации с wash(Automobile) для класса CarWash?

О: Хороший вопрос! Чтобы ваш анализ SRP имел смысл, необходимо включить параметр метода в описание. Например, «Автомойка моет [автомобиль]». Такая формулировка выглядит логично (с параметром `Automobile`), поэтому метод следует оставить в классе `CarWash`.

В: А если CarWash будет получать параметр Automobile в конструкторе и метод будет вызываться просто в формате wash()? Разве анализ SRP не приведет к ошибочному результату?

О: Приведет. Если параметр, с которым вызов метода начинает выглядеть осмысленно (как `Automobile` для метода `wash()` класса on `CarWash`), передается в конструкторе класса, анализ SRP может дать неверный результат. Именно поэтому данные анализа SRP необходимо дополнять значительной долей здравого смысла и знания системы.

По следам SRP

Принцип SRP уже несколько раз встречался в нашей работе; теперь, когда вы познакомились с ним поближе, пришло время разобраться, где и как он применялся. Посмотрите на следующую страницу и разберитесь, как и почему в этой ситуации применялся принцип SRP.

Из примера с собачьей дверью (глава 3).

Обновление класса двери

Код, закрывающий дверь, перемещается из класса Remote в класс DogDoor:

DogDoor.java

Также необходимо добавить директивы import для java.util.Timer и java.util.TimerTask.

```
public class DogDoor {
  public void open() {
    System.out.println("The dog door opens.");
    open = true;

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      public void run() {
        close();
        timer.cancel();
      }
    }, 5000);
  }

  public void close() {
    System.out.println("The dog door closes.");
    open = false;
  }
}
```

Этот код прежде находился в Remote.java.

Теперь дверь будет закрываться автоматически. Даже если добавятся новые устройства, которые смогут ее открывать. Отлично!

Упрощение класса пульта

Перенесенный код исключается из класса Remote, потому что дверь теперь автоматически закрывает себя:

```
public void pressButton() {
  System.out.println("Pressing the remote control button...");
  if (door.isOpen()) {
    door.close();
  } else {
    door.open();

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      public void run() {
        door.close();
        timer.cancel();
      }
    }, 5000);
  }
}
```

Remote.java

дальше ›

Как вы думаете, как в случае с собачьей дверью применялся принцип единственной обязанности? Запишите свой ответ:

Попробуйте найти в книге пару примеров применения принципа SRP для улучшения гибкости и структуры кода. Вы найдете их в системе управления собачьей дверью, программе поиска музыкальных инструментов Рика и в игровой системе Гэри. Запишите найденные примеры и укажите, как в них применялся принцип SRP.

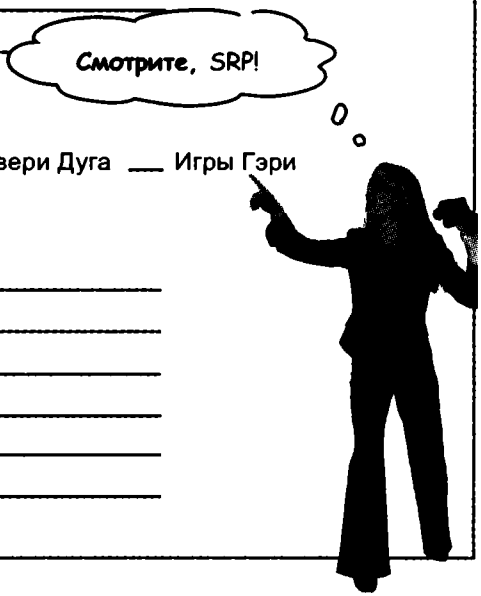
Первый пример

Приложение: ___Инструменты Рика ___Собачьи двери Дуга ___Игры Гэри

Как использовался принцип SRP:

Пометьте приложение, в котором вы нашли пример использования SRP.

Напишите, как, по вашему мнению, принцип SRP использовался в этом примере.



Второй пример

Приложение: ___Инструменты Рика ___ Собачьи двери Дуга ___ Игры Гэри

Как использовался принцип SRP:

По следам SRP — нашли!

Давайте вспомним, когда мы уже применяли принцип SRP в наших программах. Ниже мы привели свой вариант списка; посмотрите, получилось ли у вас что-нибудь похожее.

Из примера с собачьей дверью (глава 3).

Обновление класса двери

Код, закрывающий дверь, перемещается из класса Remote в класс DogDoor:

Также необходимо добавить директивы import для java.util.Timer и java.util.TimerTask.

```
public class DogDoor {
  public void open() {
    System.out.println("The dog door opens.");
    open = true;

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      public void run() {
        close();
        timer.cancel();
      }
    }, 5000);
  }

  public void close() {
    System.out.println("The dog door closes.");
    open = false;
  }
}
```

Этот код прежде находился в Remote.java.

Теперь дверь будет закрываться автоматически... Даже если добавятся новые устройства, которые могут ее открывать. Отлично!

Упрощение класса пульта

Перенесенный код исключается из класса Remote, потому что дверь теперь автоматически закрывает себя:

```
public void pressButton() {
  System.out.println("Pressing the remote control button...");
  if (door.isOpen()) {
    door.close();
  } else {
    door.open();

    ~~final Timer timer = new Timer();~~
    ~~timer.schedule(new TimerTask() {~~
      ~~public void run() {~~
        ~~door.close();~~
        ~~timer.cancel();~~
      ~~}~~
    ~~}, 5000);~~
  }
}
```

дальше ›

Как вы думаете, как в случае с собачьей дверью применялся принцип единственной обязанности? Запишите свой ответ:

Мы вывели код закрывания собачьей двери из класса Remote.java и предотвратили его дублирование в классе BarkRecognizer (привет принципу DRY!). Также мы проследили за тем, чтобы класс DogDoor выполнял все операции, связанные с функционированием собачьей двери, — это его единственная обязанность.

Здесь мы написали, как принцип SRP (и DRY) помог нам в реализации собачьей двери.

Попробуйте найти в книге пару примеров применения принципа SRP для улучшения гибкости и структуры кода. Вы найдете их в системе управления собачьей дверью, программе поиска музыкальных инструментов Рика и в игровой системе Гэри. Запишите найденные примеры и укажите, как в них применялся принцип SRP.

Первый пример

Приложение: X Инструменты Рика ___ Собачьи двери Дуга ___ Игры Гэри

Как использовался принцип SRP:

Мы создали метод matches() в классе InstrumentSpec — вместо того, чтобы оставлять код сравнения инструментов в методе search() класса Inventory. Таким образом, класс InstrumentSpec делает все, что относится к свойствам инструментов, — этот код не «размазывается» по другим классам. Принцип SRP в действии!

Возможно, ваши примеры будут отличаться от наших. Главное — убедитесь, что принцип SRP применен аналогично и вы правильно поняли суть упражнения.

Второй пример

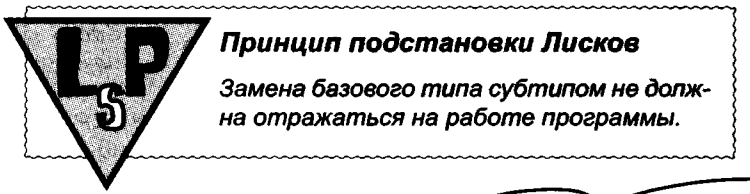
Приложение: ___ Инструменты Рика ___ Собачьи двери Дуга X Игры Гэри

Как использовался принцип SRP:

Используя контейнер Map для хранения свойств всех типов юнитов в классе Unit, мы использовали SRP. Чтобы избежать ситуации, при которой экземпляры Unit конкретной игры работают со своими свойствами, а базовый класс Unit при этом поддерживает другой набор свойств, мы переместили всю функциональность, относящуюся к свойствам, в класс Unit. Таким образом, вся работа со свойствами выполняется в ОДНОМ месте — в классе Unit.

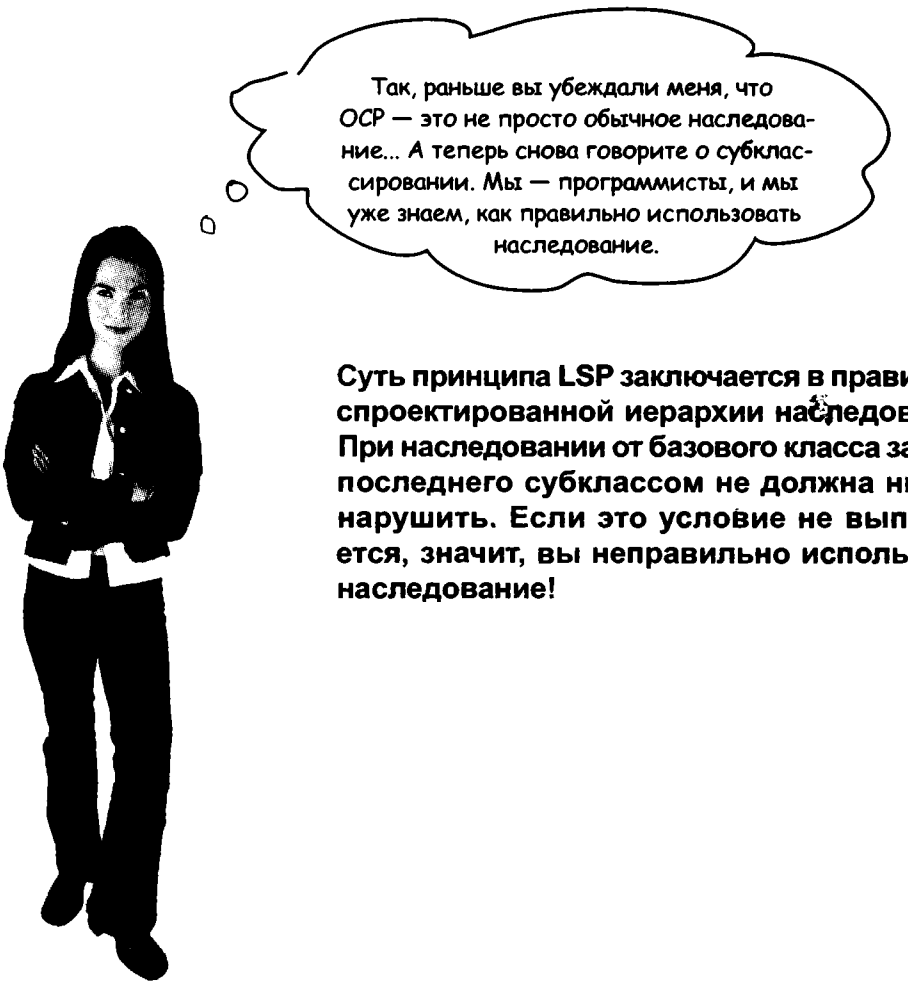
Принцип №4: Принцип подстановки Лисков

Следующую позицию нашего хит-парада принципов проектирования занимает принцип подстановки Лисков, или LSP (Liskov Substitution Principle). Его определение выглядит предельно просто:



Принцип подстановки Лисков

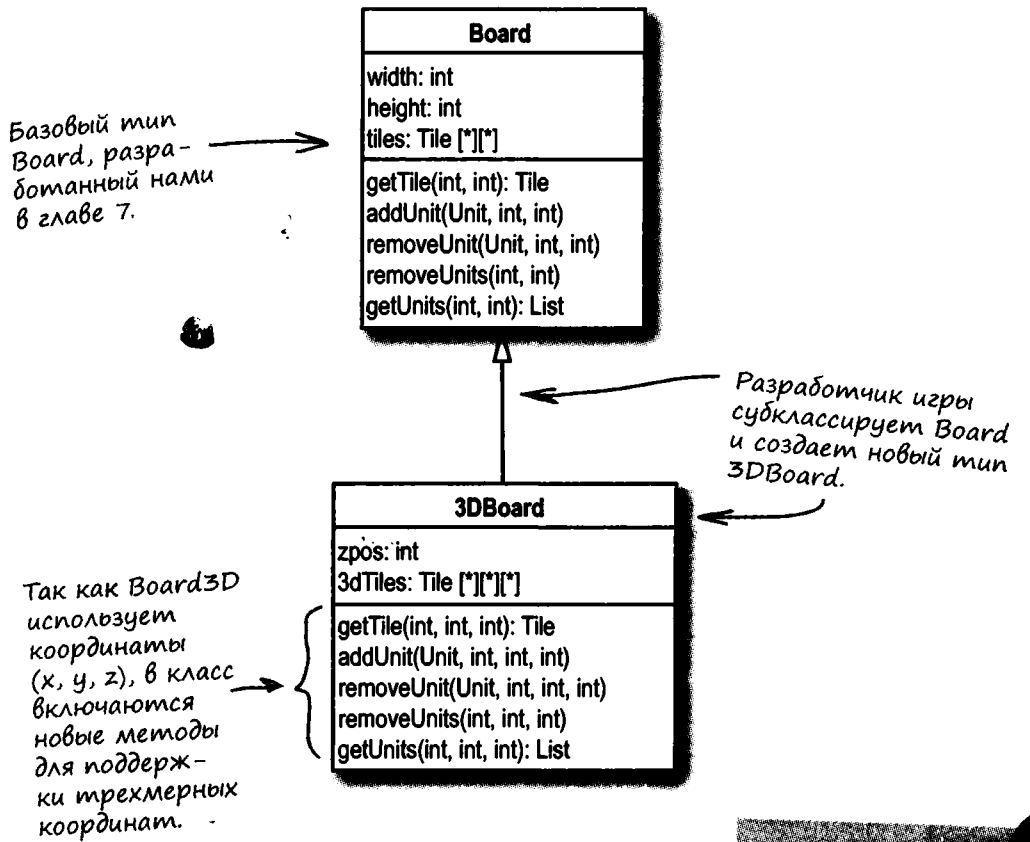
Замена базового типа субтипом не должна отражаться на работе программы.

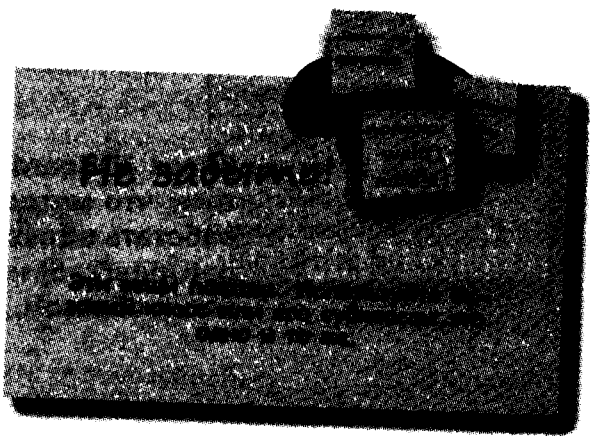


Суть принципа LSP заключается в правильно спроектированной иерархии наследования. При наследовании от базового класса замена последнего субклассом не должна ничего нарушить. Если это условие не выполняется, значит, вы неправильно используете наследование!

Пример неправильного использования наследования

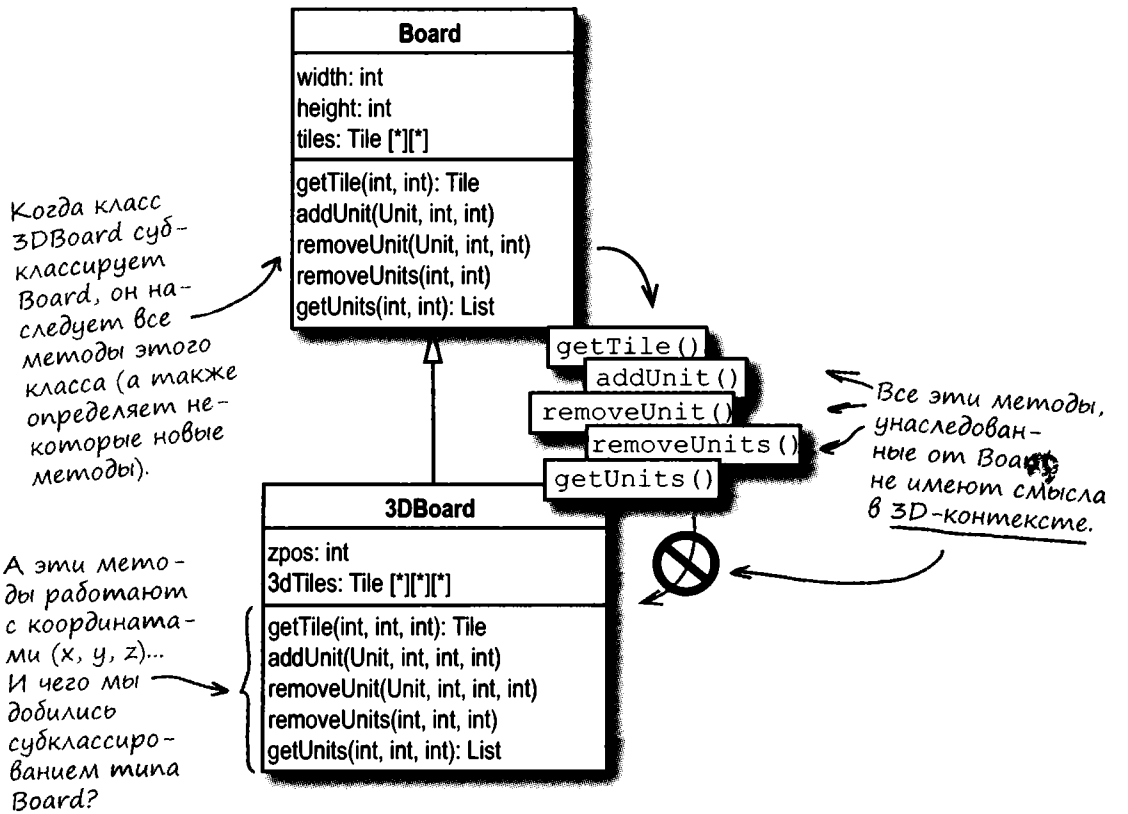
Допустим, у Гэри появился новый клиент, который хочет использовать игровую библиотеку для создания воздушных сражений времен II Мировой войны. Клиент расширяет базовый тип **Board** для поддержки трехмерной доски, представляющей небо. Вот как это выглядит:





Принцип LSP выявляет скрытые проблемы в структуре наследования structure

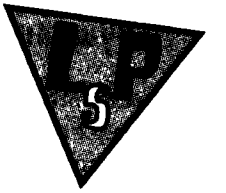
На первый взгляд идея субклассирования **Board** и использование наследования кажется отличной. Но если присмотреться повнимательнее, можно заметить проблемы, которые создает такое решение:



Класс 3DBoard не может заменить Board, потому что методы Board не будут правильно работать в трехмерном контексте. Вызов вида getUnits(2, 5) не имеет смысла для 3DBoard. Таким образом, данная структура нарушает LSP.

Что еще хуже, мы вообще не знаем, что означает для 3DBoard передача координат вида (2,5). Перед нами пример неудачного использования наследования.

«Замена базового типа субтипом не должна отразиться на работе программы»



Как уже было сказано ранее, принцип LSP требует, чтобы замена базового типа субтипом не отражалась на работе программы. Но что это означает на практике? С технической точки зрения проблем вроде бы нет:

```
Board board = new 3DBoard();
```

С точки зрения компилятора класс 3DBoard может использоваться вместо Board.

Но когда вы начинаете реально *использовать* этот экземпляр **3DBoard** как **Board**, ситсуация моментально усложняется:

```
Unit unit = board.getUnits(8, 4);
```

Не забывайте: board здесь фактически является экземпляром субтипа 3DBoard.

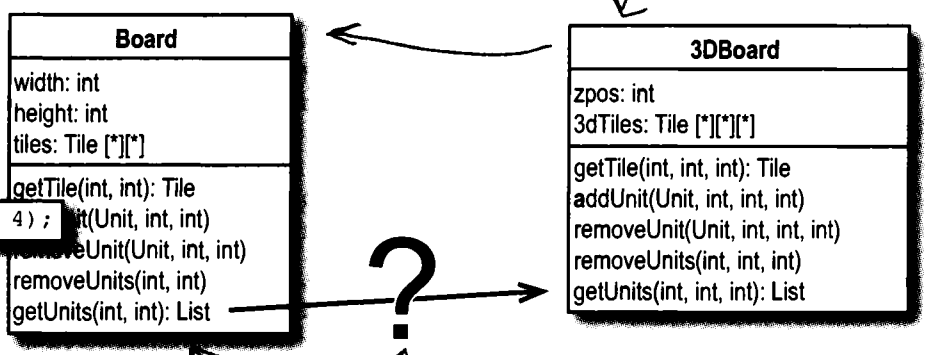
Но что этот метод означает для 3DBoard?

Итак, хотя класс **3DBoard** является субклассом **Board**, он не может заменить **Board**... Смысл методов, унаследованных **3DBoard**, изменился по отношению к суперклассу. Что еще хуже, вообще неясно, *какой* смысл имеют эти методы!

Наследование (и LSP) означает, что любой метод Board должен использоваться для экземпляра 3DBoard... То есть класс 3DBoard может заменить Board без всяких проблем.

Некоторые фрагменты кода вызывают метод из Board, но для экземпляра 3DBoard:

```
Unit unit = removeUnits(8, 4);
```

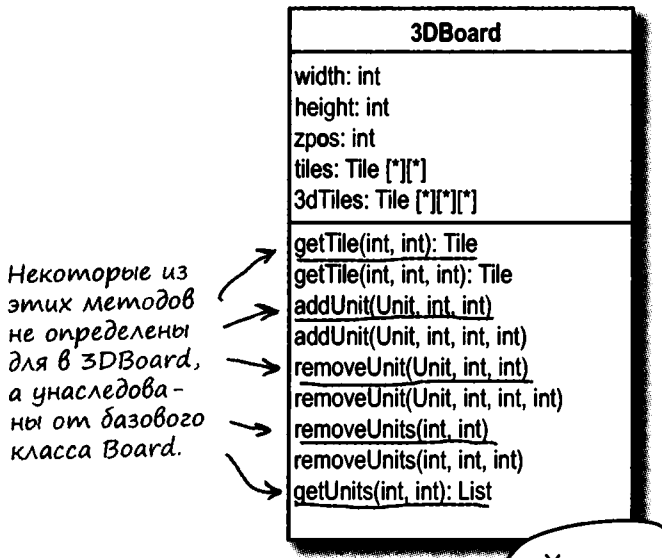


Но что эти методы означают для 3DBoard? Да наверное, ничего не означают!

Нарушение LSP усложняет код

Может показаться, что это не такая уж серьезная проблема, но код, нарушающий LSP, может оказаться весьма запутанным, а его отладка превращается в сущий кошмар. Попробуем представить, как неопытный пользователь будет работать с плохо спроектированным классом **3DBoard**.

Вероятно, для начала он просмотрит список методов класса:



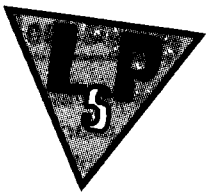
Хмм, я что-то не пойму, какую версию getTile() или addUnit() нужно использовать. Может, эти методы получают координаты X и Y для текущей доски... Даже не знаю.

Трудно понять код, в котором неправильно используется наследование.

При использовании наследования ваш субкласс получает все методы своего суперкласса, даже если они вам не нужны. И при неудачном использовании наследования класс будет содержать множество лишних методов, которые, скорее всего, не имеют смысла в вашем субклассе.

Что можно сделать, чтобы этого не произошло? Прежде всего убедитесь в том, что субклассы могут подставляться вместо своего базового типа, то есть попросту проверьте соблюдение принципа LSP. А потом исследуйте *альтернативы* для применения наследования в программном коде...

Решение задачи 3DBoard без применения наследования



Мало понять, что наследование здесь не подходит... Необходимо разобраться, что же *следовало* сделать. Давайте еще раз проанализируем классы **Board** и **3DBoard** и посмотрим, нельзя ли создать трехмерную доску без применения наследования.

Класс Board содержит функциональность, нужную классу 3DBoard, но не является базовым типом для 3DBoard.

Board

width: int
height: int
tiles: Tile [*][*]

getTile(int, int): Tile
addUnit(Unit, int, int)
removeUnit(Unit, int, int)
removeUnits(int, int)
getUnits(int, int): List

boards *

Вместо расширения используется ассоциация. Таким образом, 3DBoard может использовать поведение Board без наследования и нарушения LSP.

3DBoard может хранить массив объектов Board, то есть трехмерная доска строится как коллекция двумерных.

3DBoard

zpos: int
3dTiles: Tile [*][*][*]

getTile(int, int, int): Tile
addUnit(Unit, int, int, int)
removeUnit(Unit, int, int, int)
removeUnits(int, int, int)
getUnits(int, int, int): List

Методы 3DBoard используют координату zpos для выбора используемого экземпляра Board в массиве, после чего передают координаты (x,y) делегированным операциям Board.

Это разновидность делегирования. Класс 3DBoard делегирует большую часть своей функциональности отдельным экземплярам Board.

Эти методы очень похожи на методы Board, но они используют функциональность Board вместо ее расширения. Таким образом, наследование плохо подходит для этой ситуации.

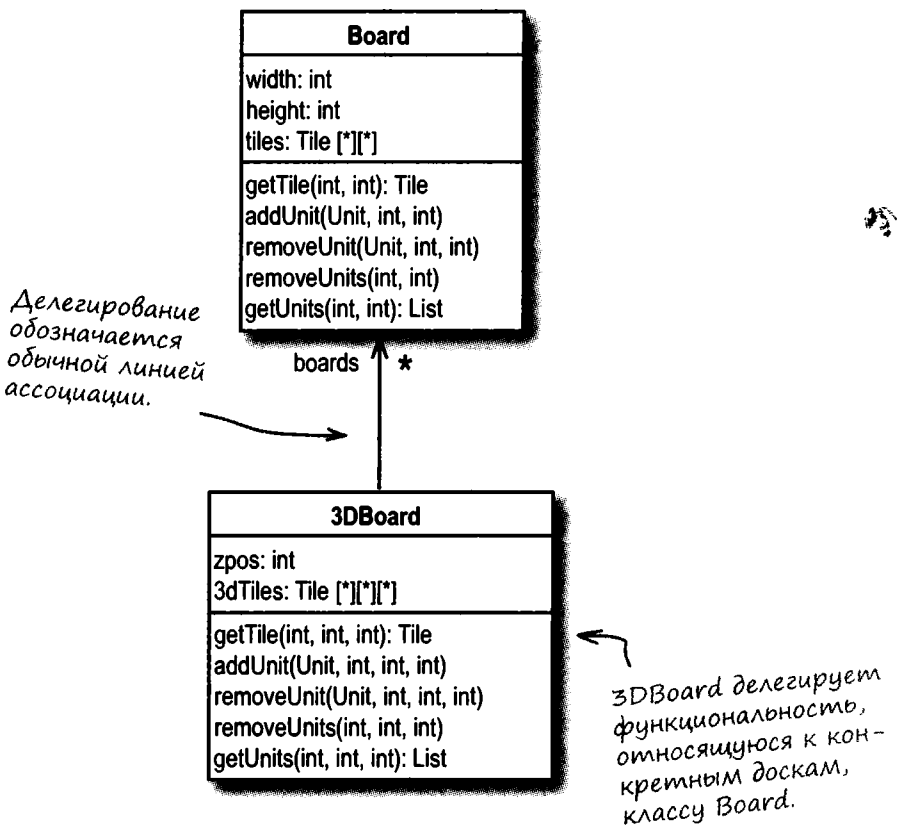
Так что можно использовать вместо наследования?

Делегирование функциональности другому классу

Мы уже знаем, что делегированием называется поручение операции другому классу. Делегирование – всего лишь одна из возможных альтернатив наследованию.

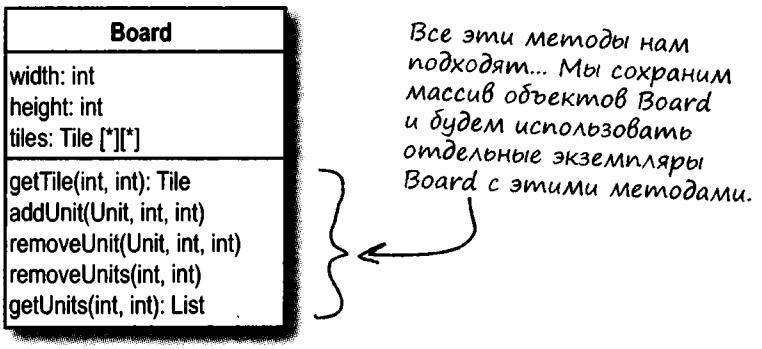
***Делегированием** называется поручение выполнения некоторой операции другому классу или методу.*

Для решения рассматриваемой задачи **3DBoard** мы воспользовались делегированием, не прибегая к наследованию:

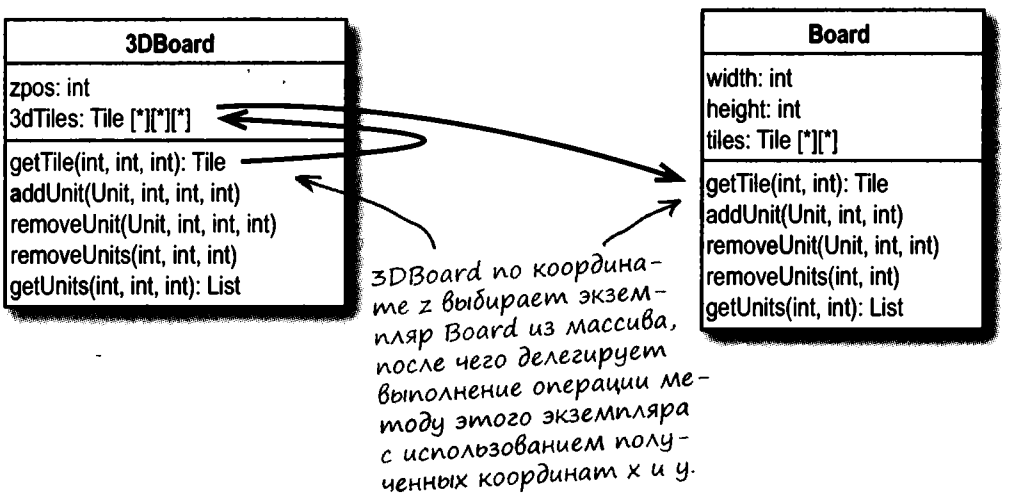


Когда применяется делегирование

Делегирование лучше всего применять в ситуациях, в которых нужно использовать функциональность другого класса «как есть», без каких-либо изменений. В случае с **3DBoard** мы хотели использовать методы класса **Board**:



Так как мы не собираемся изменять существующее поведение, но хотим использовать его, можно создать между **3DBoard** и **Board** отношение делегирования. **3DBoard** хранит набор экземпляров **Board** и делегирует им выполнение задач, относящихся к конкретным доскам.

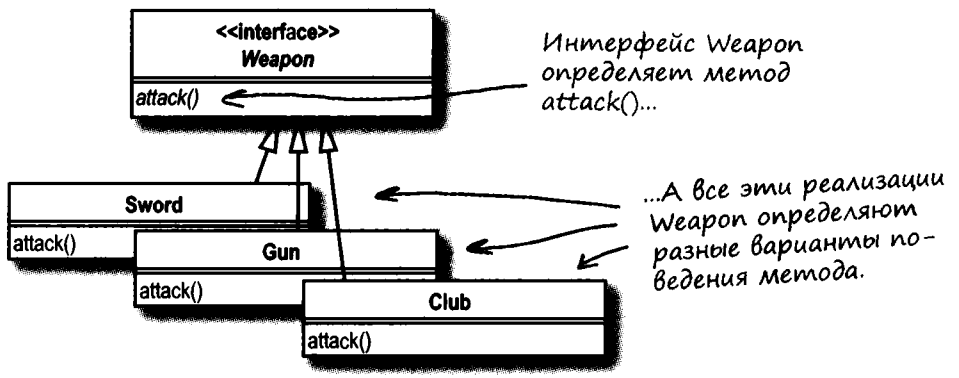


Если вам потребуется использовать функциональность другого класса без ее изменения, рассмотрите возможность использования делегирования вместо наследования.

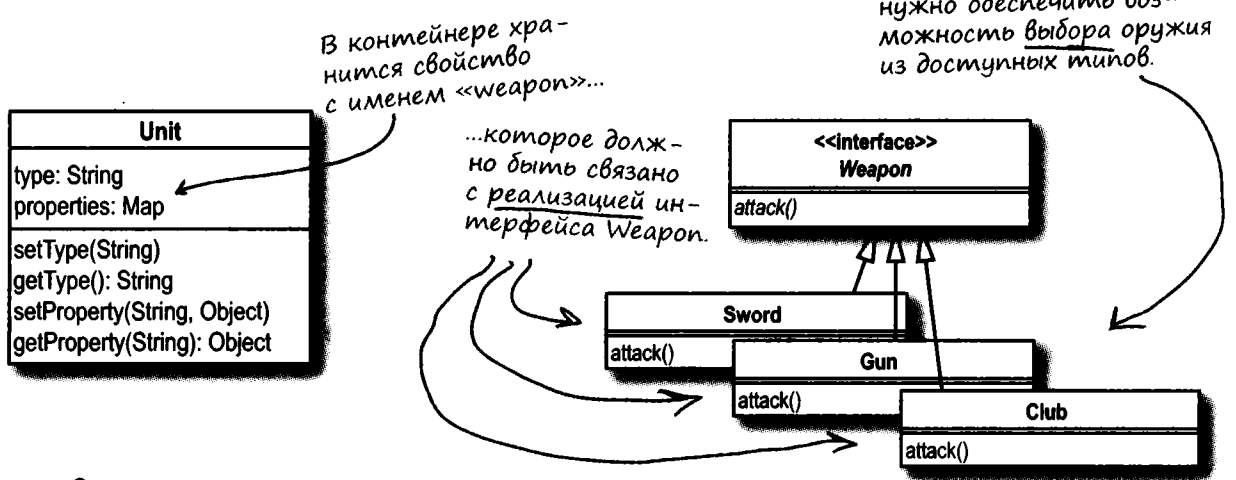
Использование композиции для объединения поведения других классов

Иногда оказывается, что делегирование – не совсем то, что вам нужно. При делегировании поведение объекта, которому делегируется выполнение операций, остается неизменным. **3DBoard** всегда использует экземпляры **Board**, а поведение методов **Board** не изменяется.

Но в некоторых случаях желательно иметь выбор из нескольких вариантов поведения. Например, допустим, что мы захотели разработать интерфейс вооружения **Weapon** и создали несколько реализаций этого интерфейса с разным поведением:

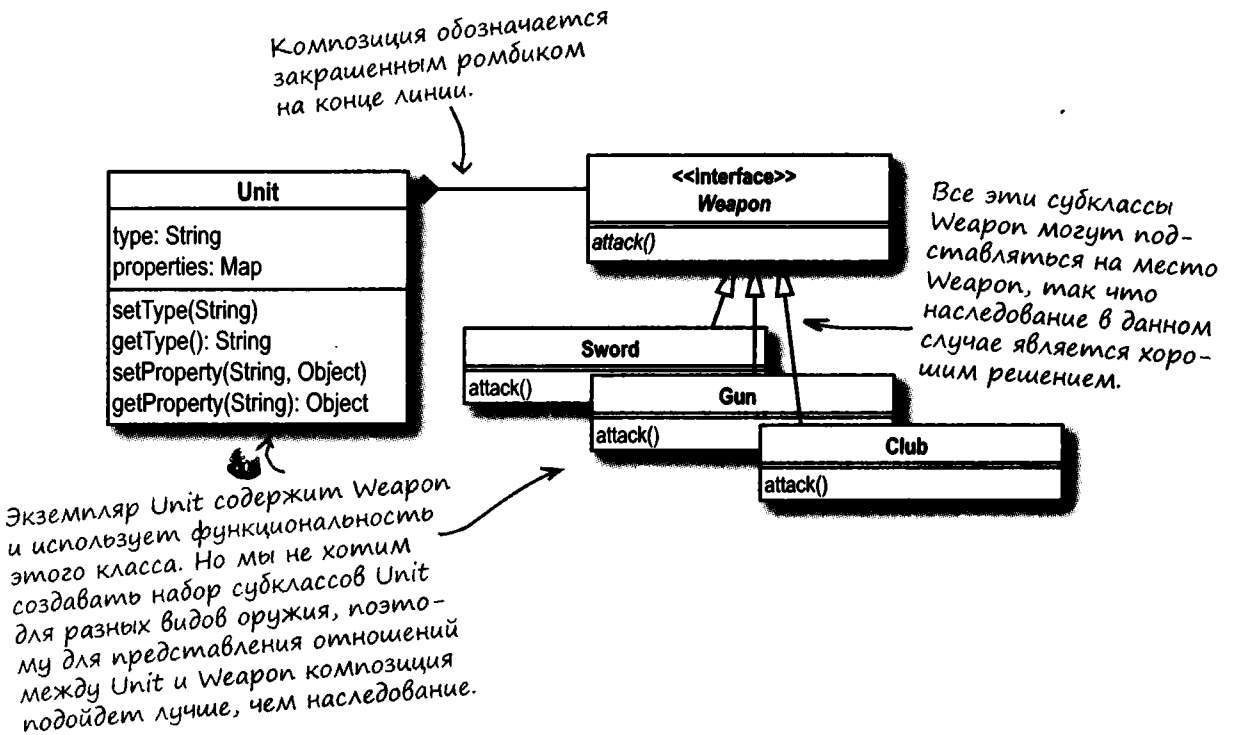


Теперь необходимо использовать поведение из этих классов в классе **Unit**. Среди прочих свойств в контейнере **Map** хранится свойство weapon, значением которого должна быть реализация класса **Weapon**. Но класс **Unit** может менять вооружение, поэтому свойство weapon не должно быть привязано к конкретной реализации **Weapon**; мы хотим, чтобы каждый экземпляр **Unit** мог обращаться к **Weapon** независимо от того, какая реализация **Weapon** должна использоваться.

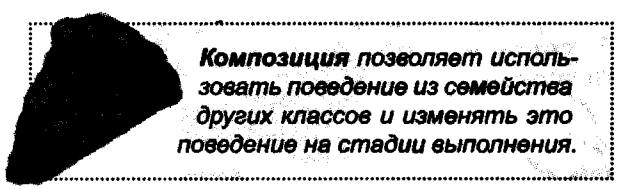


Когда используется композиция

Для выбора поведения из семейства возможных вариантов (как в классе Unit) используется композиция. На диаграммах UML она обозначается следующим образом:



Композиция особенно эффективна тогда, когда вы хотите использовать поведение, определенное в интерфейсе, с возможностью выбора из разных реализаций этого интерфейса, как во время компилирования, так и на стадии выполнения.



Композиция позволяет использовать поведение из семейства других классов и изменять это поведение на стадии выполнения.

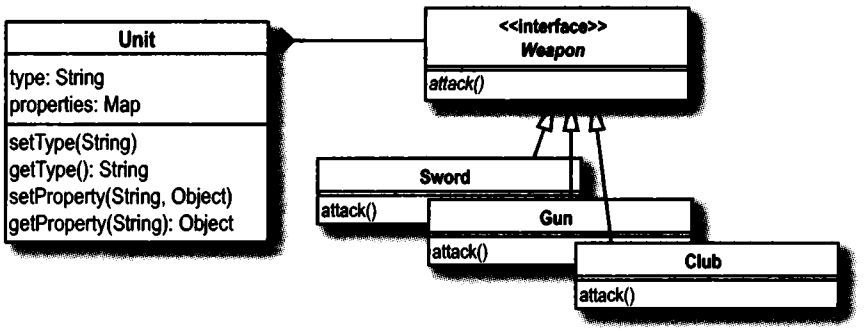
Пицца — отличный пример композиции: она состоит из разных ингредиентов, но ингредиенты можно заменять без каких-либо последствий для всего куска.

[От отдела маркетинга: вы сравниваете принципы программирования с пиццей и хотите, чтобы к нам относились серьезно?]

Когда пиццы не остается, то не остается и ингредиентов...

Говоря о композиции, мы не упомянули один важный момент. Если один объект содержит другие объекты, то при уничтожении объекта-владельца *объекты, образующие композицию, тоже будут уничтожены.* Звучит непонятно, так что давайте разберемся, что это на самом деле означает.

Это наш класс **Unit**, связанный отношениями композиции с интерфейсом **Weapon** и его реализациями:



Предположим, мы создали новый экземпляр **Unit** и присвоили его свойству weapon экземпляра **Sword**:

```
Unit pirate = new Unit();
pirate.setProperty("weapon", new Sword());
```

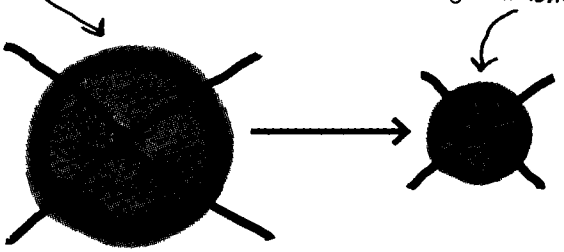
Экземпляр Unit содержит экземпляр Sword.

Объект Sword не существует вне этого конкретного экземпляра Unit.

Что произойдет при уничтожении **Unit**? Очевидно, переменная **pirate** уничтожается, но и экземпляр **Sword**, на который она ссылается, тоже будет уничтожен. Он не существует *за пределами* объекта **pirate**.

При уничтожении объекта pirate типа Unit...

...связанный с pirate объект Sword тоже будет уничтожен.



При использовании композиции объект, содержащий варианты поведения других объектов, является их владельцем. Когда этот объект уничтожается, то вместе с ним уничтожаются все композитные варианты поведения.

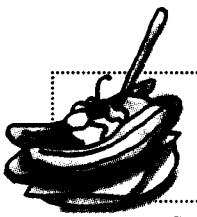
Варианты поведения, используемые в композиции, не существуют за пределами самой композиции.

МОЗГОВОЙ ШТУРМ

Можете ли вы привести пример того, как аспект владения объектами при композиции играет отрицательную роль в приложении? Когда может возникнуть необходимость в существовании внедренных объектов за пределами композитного класса?

Агрегирование: композиция с продолжением существования

Что делать, если вы хотите использовать все преимущества композиции – гибкость в выборе поведения, соблюдение принципа LSP, – но внутренние объекты должны существовать вне главного объекта? На помощь приходит агрегирование.



***Агрегирование** — использование одного класса как части другого с возможностью существования за его пределами.*

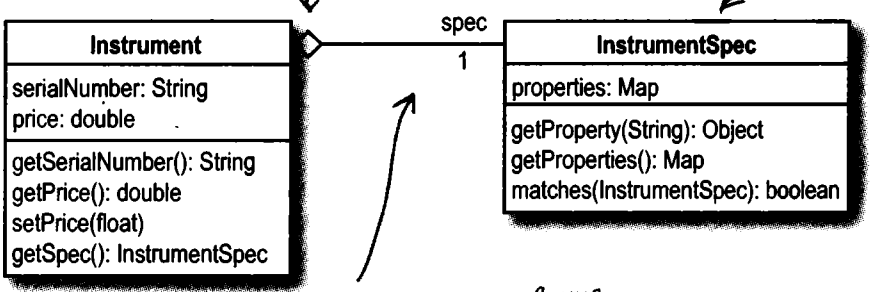
Мороженое, бананы и вишни существуют за пределами бананового десерта. Даже если контейнер пропадает, отдельные компоненты все равно остаются.

Мы уже использовали агрегирование

Агрегирование уже использовалось в приложении для Рика из главы 5:

InstrumentSpec используется как часть Instrument, но спецификация также может существовать вне определения инструмента (например, когда она заполняется покупателем для поиска).

Агрегирование обозначается пустым ромбиком на конце линии.



Применение агрегирования позволило нам обойтись без субклассов конкретных инструментов.

Агрегирование и композиция

Часто бывает трудно понять, когда стоит применить композицию, а когда агрегирование. Проще всего спросить себя: *существует ли объект, поведение которого я хочу использовать, за пределами объекта, использующего его поведение?*

Если самостоятельное существование объекта имеет смысл, используйте агрегирование, а если нет — композицию. Но будьте внимательны! Иногда даже незначительные изменения в использовании объекта приводят к очень серьезным последствиям.

Маленькая головоломка

Джоэл откинулся на спинку, потянулся и снова подумал, что было бы неплохо купить новое кресло, если подвернется премия. Программирование игр было тяжелой работой, и Джоэл снова задерживался в офисе позже всех.

«Наша новинка произведет фурор», — подумал он. Джоэл пролистал руководство пользователя игровой библиотеки Гэри и задумался над тем, как лучше реализовать ковбоев для новой игры — одна из последних проблем, которые ему предстояло решить. Внезапно взгляд Джоэла остановился на классе Unit, и он понял, что ковбоев можно представить объектами Unit, а для представления лассо, револьверов и устройств для клеймения скота вполне подойдет интерфейс Weapon.

Джоэл создал классы Lasso, Revolver и BrandingIron и позаботился о том, чтобы они реализовали интерфейс Weapon. Он даже добавил в класс Building свойство Weapon, чтобы ковбои после долгого трудового дня могли повесить свое снаряжение в доме.

«Хорошее, изящное решение... Немного композиции, и рядом с моим именем в титрах будет указано: "Ведущий проектировщик"». Джоэл быстро набросал диаграмму классов для утренней встречи, закрасил ромбик композиции на линии, соединяющей классы Unit и Weapon, и отправился на ужин в кафе.

Но утром его ждал неприятный сюрприз: вместо поздравлений начальник устроил ему разнос...

Что Джоэл сделал не так?

Ответы на с. 442

Наследование — всего лишь одна из возможностей

Этот раздел начинался с LSP и общей идеи о том, что замена базового класса субклассами не должна создавать проблем. Но что еще важнее, мы рассмотрели несколько способов повторного использования поведения других классов, кроме наследования.

Делегирование

Делегируйте выполнение операций другому классу в тех ситуациях, когда вы не хотите изменять поведение, но самостоятельная реализация этого поведения не входит в обязанность вашего объекта.

Композиция

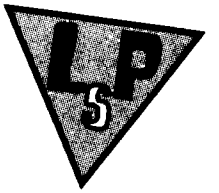
Композиция позволяет повторно использовать поведение одного или нескольких классов (и особенно семейств классов). Ваш объект полностью владеет всеми внедряемыми объектами, и они не существуют вне контекста своего использования в вашем объекте.

Агрегирование

Агрегирование предоставляет все преимущества композиции, но при этом позволяет использовать поведение другого объекта, самостоятельно существующего за пределами вашего объекта.

Все три приема позволяют повторно использовать поведение других объектов без нарушения LSP.

Если вы будете использовать делегирование, композицию и агрегирование вместо наследования, ваши программы будут получаться более гибкими и простыми в сопровождении, расширении и повторном использовании.



Часто задаваемые вопросы

В: Я думал, что субклассирование — это хорошо. А теперь вы говорите, что это плохо?

О: Нет, субклассирование и наследование играют ключевую роль в любом нормальном языке ООП. Однако принцип LSP говорит не о субклассировании вообще, а о том, когда следует применять субклассирование. Если ваш субкласс действительно заменяет базовый класс, то, скорее всего, вы правильно используете наследование. Если же субкласс не может быть подставлен на место базового типа, вам стоит присмотреться к другим решениям, таким как агрегирование и делегирование.

В: И если класс не должен расширять другой класс, то вместо наследования можно воспользоваться делегированием, композицией или агрегированием?

О: Конечно. Принцип LSP не распространяется на агрегирование или делегирование, потому что эти два механизма прекрасно подходят для решения проблем с деревьями наследования, не соответствующими LSP. Правильное использование LSP идет рука об руку с возрастанием роли делегирования, композиции и агрегирования.

В: И нам обязательно было привлекать LSP, чтобы это выяснить? Разве речь идет не о простом написании хороших ОО-программ?

О: Достаточно часто для написания хорошего кода не нужно беспокоиться о формальном названии принципа проектирования. Например, взгляните на пример `Board` на с. 423: чтобы заставить `3DBoard` расширять `Board`, пришлось изменить все методы! Это верный признак того, что наследование создает проблемы.

В: В тексте встречается множество условных обозначений UML. Как мне запомнить, что все они означают?

О: Вам вообще не нужно запоминать условные обозначения. Хотя UML предоставляет специальные обозначения для агрегирования и композиции, они всего лишь являются особыми видами ассоциации. Следовательно, как и в случае с делегированием, для обозначения композиции и агрегирования можно использовать обычную линию со стрелкой.

В: Но ведь мы собьем с толку разработчиков, которые не будут знать, какой тип ассоциации следует использовать?

О: Не исключено, но зато и гибкость существенно увеличивается. Предположим, позднее вы решите, что при уничтожении армии входящие в нее подразделения не должны автоматически уничтожаться. Тогда отношение между армией и подразделением преобразуется из композиции в агрегирование.

Если вы используете базовую стрелку ассоциации, диаграмму классов даже не придется изменять. Кроме того, у разработчиков появляется больше свободы для применения собственных идей по реализации ассоциации.

В условных обозначениях агрегирования и композиции нет ничего плохого, но вам не стоит слишком сильно полагаться на них, особенно на ранней стадии цикла разработки. Никогда не знаешь, что может измениться позднее, а гибкая структура всегда лучше жесткой.

Компания классов-участников реализации принципов ООП, облаченных в маскарадные костюмы, развлекается игрой «Кто я?» Игрок дает подсказку, а остальные на основании сказанного им пытаются угадать, кого он изображает. Будем считать, что игроки всегда говорят правду о себе. Если сказанное ими может относиться сразу к нескольким персонажам, перечислите всех, к кому может относиться их высказывание. Заполните пропуски именами одного или нескольких участников. Первый правильный ответ за нами.

Кто я?

Сегодняшние участники:

Субкласс **Делегируемый класс** **Агрегированный класс**

Делегирующий класс **Композитный класс**

Меня можно подставлять на место моего базового типа. _____

Я поручаю другим работать за себя. _____

Мое поведение используется как часть поведения другого класса. _____

Я изменяю поведение другого класса. _____

Я не изменяю поведение другого класса. _____

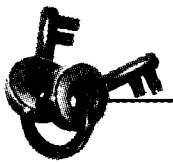
Я могу объединять поведение нескольких других классов. _____

Я остаюсь, даже если остальные классы уходят. _____

Я получаю мое поведение и функциональность от базового типа. _____

→ Ответы на с. 441

КЛЮЧЕВЫЕ МОМЕНТЫ



- Принцип открытости/закрытости (OCP) обеспечивает гибкость и возможность повторного использования кода — классы открыты для расширения, но закрыты для изменения.
- Соблюдение классами принципа единственной обязанности упрощает применение OCP в программном коде.
- Когда вы пытаетесь определить, относится ли метод к обязанности класса, спросите себя: *Входит ли в задачу класса выполнение этой конкретной операции?* Если нет, переместите метод в другой класс.
- Когда ОО-код будет почти завершен, проверьте выполнение принципа DRY. Избегайте дублирования кода и проследите за тем, чтобы каждый аспект поведения в коде находился в единственном месте.
- Принцип DRY применим не только к коду, но и к требованиям: каждая функциональная возможность и требование должны быть реализованы в единственном месте.
- Принцип подстановки Лисков (требование возможности замены базового типа субтипом) обеспечивает правильность использования наследования.
- При обнаружении кода, нарушающего LSP, рассмотрите возможность замены наследования делегированием, композицией и агрегированием.
- Если вам понадобилось поведение другого класса, но вы не хотите изменять его, просто делегируйте выполнение операций этому классу.
- Композиция позволяет выбрать поведение из набора вариантов поведения, часто определяемых разными реализациями одного интерфейса.
- При использовании композиции внешний объект является владельцем используемых объектов поведения, причем последние прекращают существование сразу же при уничтожении внешнего объекта.
- Агрегирование позволяет использовать аспекты поведения из другого класса без ограничения их срока жизни.
- Агрегированные объекты продолжают существовать даже после уничтожения агрегирующего объекта.

Инструментарий ООАП

В этой главе наш инструментарий пополнился несколькими новыми ОО-принципами. Добавим то, что мы узнали здесь, в наши заметки. И не забывайте: эти принципы лучше использовать вместе, а не по отдельности!

Требования

Хорошие требо
диться в том,
работает так

Требования долж
на все шаги вар
системы.

Из вариант
узнать то
рассказать

Анализ вар
жет выяви
ствующие

Требования
стут) со в

Анализ и проектирование

Хорошо спроектированный код
ко изменяется и расширяется.

Используйте базовые ОО-

Решение больших задач

Слушайте заказчика и поймите, какая система ему нужна.

Постройте список функциональных возможностей на языке, понятном заказчику.

что функциональные
ли соответствуют
азчика.

общие планы системы
диаграмм вариантов
я (и вариантов ис-

ьшую систему на
ебольших частей

паттерны проекти-
ементам системы.

ООАП при работе
ли.

ОО-принципы

Инкапсулируйте то, что изменяется.

Программируйте для интерфейса, а не для реализации.

Каждый класс приложения должен иметь только одну причину для изменения.

Классы создаются ради поведения и функциональности.

Классы должны быть открыты для расширения, но закрыты для изменения (OCP).

Избегайте дублирования кода, абстрагируя общие аспекты классов и размещая их в одном месте (принцип DRY).

Каждый объект в системе должен иметь одну обязанность, и вся работа объекта должна быть сосредоточена на выполнении этой единственной обязанности (SRP).

Замена базового типа субтипом не должна отразиться на работе программы (LSP).

Кто я? Решение

Компания классов-участников реализации принципов ООП, облаченных в маскарадные костюмы, развлекается игрой «Кто я?» Игрок дает подсказку, а остальные на основании сказанного им пытаются угадать, кого он изображает. Будем считать, что игроки всегда говорят правду о себе. Если сказанное ими может относиться сразу к нескольким персонажам, перечислите всех, к кому может относиться их высказывание. Заполните пропуски именами одного или нескольких участников. Первый правильный ответ за нами.

Сегодняшние участники:

Субкласс **Делегируемый класс** **Агрегированный класс**

Делегирующий класс **Композитный класс**

Меня можно подставлять на место моего базового типа. субкласс

Я поручаю другим работать за себя. делегирующий класс, композитный класс

Это базовое определение делегирования, но класс, использующий композицию, использует поведение других классов.

Мое поведение используется как часть поведения другого класса. агрегированный класс

Я изменяю поведение другого класса. субкласс

Я не изменяю поведение другого класса. делегированный класс, агрегированный класс, делегирующий класс, композитный класс

Только субкласс изменяет поведение другого класса.

Я могу объединять поведение нескольких других классов. композитный класс, делегирующий класс

Я остаюсь, даже если остальные классы уходят. агрегированный класс, делегирующий класс

Я получаю мое поведение и функциональность от базового типа. субкласс

При агрегировании и делегировании экземпляры объектов связываются, но их существование не зависит друг от друга.

Маленькая головоломка. Решение

Ошибка Джоэла кроется в этой строке:

"...Он даже добавил в класс Building свойство Weapon, чтобы ковбои после долгого трудового дня могли повесить свое снаряжение в доме".

Тем самым Джоэл принял решение, что классы Lasso, Revolver и BrandingIron смогут существовать вне конкретного экземпляра Unit. Ковбой не является владельцем снаряжения; он просто временно использовал его поведение.

Так как реализации Weapon существуют вне конкретного ковбоя, Джоэлу следовало применить агрегирование вместо композиции. Разные ковбои могли бы использовать одну реализацию Weapon в разное время, и снаряжение продолжит существовать, даже если ковбоя, который его использует, затопчет бешеный бык.

Программы пишутся для заказчика

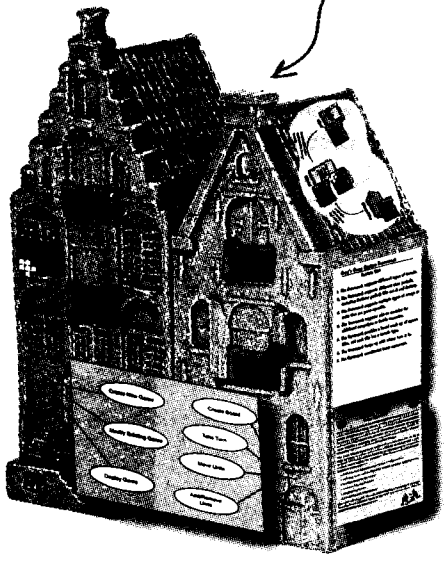
Пришло время показать заказчику, что вы действительно беспокоитесь о его интересах. Назойливое начальство? Обеспокоенные заказчики? Люди, которые постоянно спрашивают: «Ну что, успеете к сроку?» Никакой хорошо структурированный код не порадует вашего заказчика; вы должны показать ему что-то работающее. А теперь, когда вы овладели солидным инструментарием ОО-программирования, пора узнать, как доказать заказчику, что его программа действительно работает. В этой главе мы рассмотрим два способа углубленного анализа функциональности вашего продукта, после которого заказчик с чувством скажет: *«Да, вы определенно идеально подходите для этой работы!»*

Инструментарий постепенно заполняется

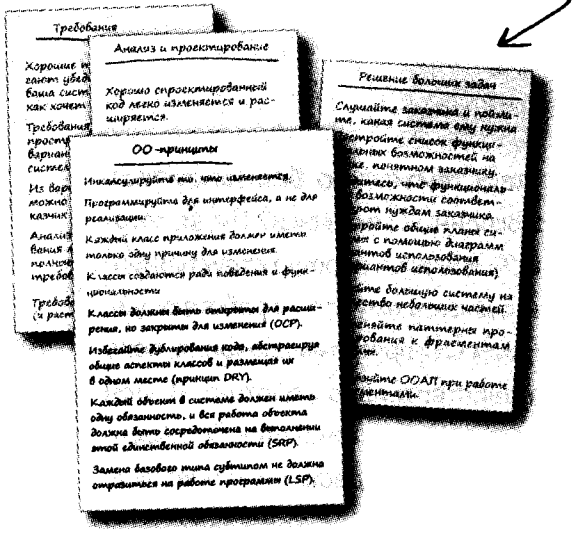
Вы уже довольно много узнали из этой книги, и наш арсенал инструментов проектирования и анализа постепенно пополняется. В предыдущей главе мы даже добавили в него некоторые приемы ОО-программирования:

В главах 6 и 7 мы при помощи диаграмм вариантов использования и списков ключевых функциональных возможностей преобразовали общую концепцию в архитектуру приложения.

ОО-принципы из главы 8 помогают писать хорошо спроектированные, гибкие ОО-программы.



Мы рассмотрели уйму принципов и приемов для сбора требований, анализа и проектирования, а также решения всевозможных проблем из области программирования.



1. **Убедитесь в том, что поведение программы соответствует требованиям заказчика.**

2. **Применяйте базовые ОО-принципы для повышения гибкости.**

3 шага к созданию хороших программ были представлены в главе 1, но с тех пор мы последовательно применяли их в каждой главе.

3. **Постарайтесь создать структуру кода, упрощающую его сопровождение и повторное использование.**

Все это хорошо, конечно, и я уверен, что ты отличный разработчик. Но честно говоря, меня интересует совсем другое... **где мое приложение?**

Но программу-то вы пишете для ЗАКАЗЧИКА!

Инструменты, приемы — замечательно... Но все это не имеет никакого значения, если вы не используете их для создания хороших программ, которыми будет доволен заказчик.

И как правило, заказчика не интересуют ОО-принципы и диаграммы. Он хочет только одного: чтобы программа работала так, как ей положено.

Гэри хочет увидеть свою игровую библиотеку в действии.

Хорошие программы пишутся методом итераций. Поработайте над общей картиной, а потом — над каждым фрагментом приложения.

Джо: Не стоило нам тратить столько времени на диаграммы и архитектурные штучки. Теперь нам нечего показать Гэри, кроме кружков с надписями.

Фрэнк: Бросьте, мы сделали намного больше. Сейчас закончить класс Board будет проще простого, потому что мы уже взялись за написание большой части его функциональности.

Джилл: Я согласна, но ведь это единственный класс, для которого мы написали хоть какой-то код. Как мы будем показывать его Гэри?

Джо: Я думаю, мы легко напишем класс Unit, потому что у нас есть для него готовая диаграмма. Написание кода не должно занять много времени.

Фрэнк: Вот именно. И вообще, мы ведь знаем, как написать все эти классы. Мы просто берем каждый класс или даже целый пакет и применяем к каждому функциональному блоку все принципы объектно-ориентированного анализа и проектирования.

Джилл: Но нам придется работать с функциональностью. У нас не будет времени на рисование картинок и анализ.

Фрэнк: В этом-то и дело, Джил. Нам уже не придется изменять то, что мы делаем, нужно только увеличить глубину итераций.

Джо: Увеличить глубину итераций? Что это значит?

Фрэнк: Только то, что мы будем выполнять те же операции анализа и проектирования, но с отдельными частями игровой системы Гэри.

Джилл: И в процессе построения приложения появятся рабочие фрагменты, которые мы сможем показать Гэри, верно?

Джо: И мы применим все известные нам инструменты, чтобы наша программа получилась хорошо спроектированной?

Фрэнк: Вот именно. Но сначала нужно принять одно важное решение...

Углубление итераций: два основных варианта

В области разработки ПО не существует единственного способа итеративной проработки конкретных фрагментов приложения. Вам приходится работать над меньшими функциональными блоками, но при этом существуют два основных способа выбора этих блоков и даже определения того, что считать «меньшим блоком» в контексте вашего приложения.

Можно сосредоточиться на конкретных функциональных возможностях приложения. Вы берете один блок функциональности, необходимой вашему заказчику, и работаете над ним, пока он не будет готов.

Функционально-ориентированная разработка

...вы выбираете конкретную функциональную возможность приложения, планируете, анализируете и разрабатываете ее до завершения.

Также можно сосредоточиться на потоках операций в приложении. В этом варианте вы берете полный путь работы с приложением, с четко определенной начальной и конечной точкой, и реализуете его в своем коде.

Сценарно-ориентированная разработка

...вы выбираете конкретный сценарий в варианте использования и пишете код реализации этого сценария в варианте использования.

Термины «сценарий» и «поток операций» часто используются как синонимы.

Оба метода итераций управляются качественными требованиями.

Так как требования исходят от заказчика, оба метода направлены на реализацию пожеланий заказчика.

Функционально-ориентированная разработка

При использовании **функционально-ориентированной разработки** вы последовательно работаете над отдельными функциональными возможностями. Таким образом, вы реализуете одну функциональную возможность за раз, пока не будет реализована вся функциональность приложения.

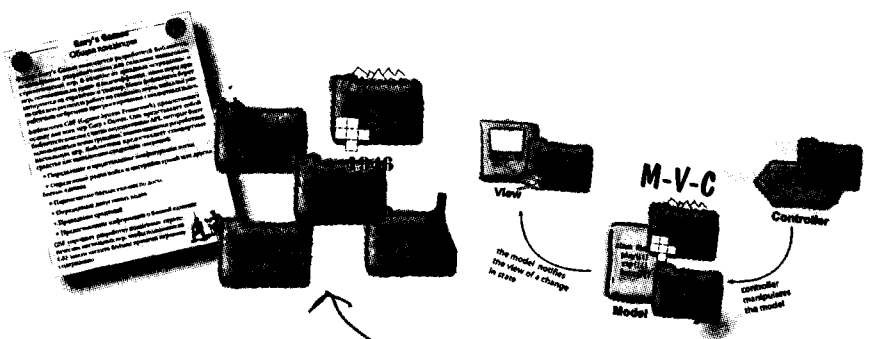
При функционально-ориентированной разработке вы работаете над конкретной функциональной возможностью, а в центре внимания находится список функциональных возможностей приложения.

Игровая библиотека

Список функциональных возможностей

1. Поддержка разных типов рельефа.
2. Поддержка разных исторических периодов, а также фантастического и фантезийного антуража.
3. Поддержка разных типов войск в зависимости от конкретной игры.
4. Поддержка модулей расширения для дополнительных кампаний и сценариев.
5. Доска состоит из квадратных полей, каждому полю присваивается некоторый тип рельефа.
6. Библиотека следит за очередностью хода.
7. Библиотека координирует базовые перемещения.

Таким образом, вы берете функциональную возможность № 1 и начинаете работать над классом Terrain, а также над классом Tile, чтобы обеспечить поддержку разных типов рельефа.



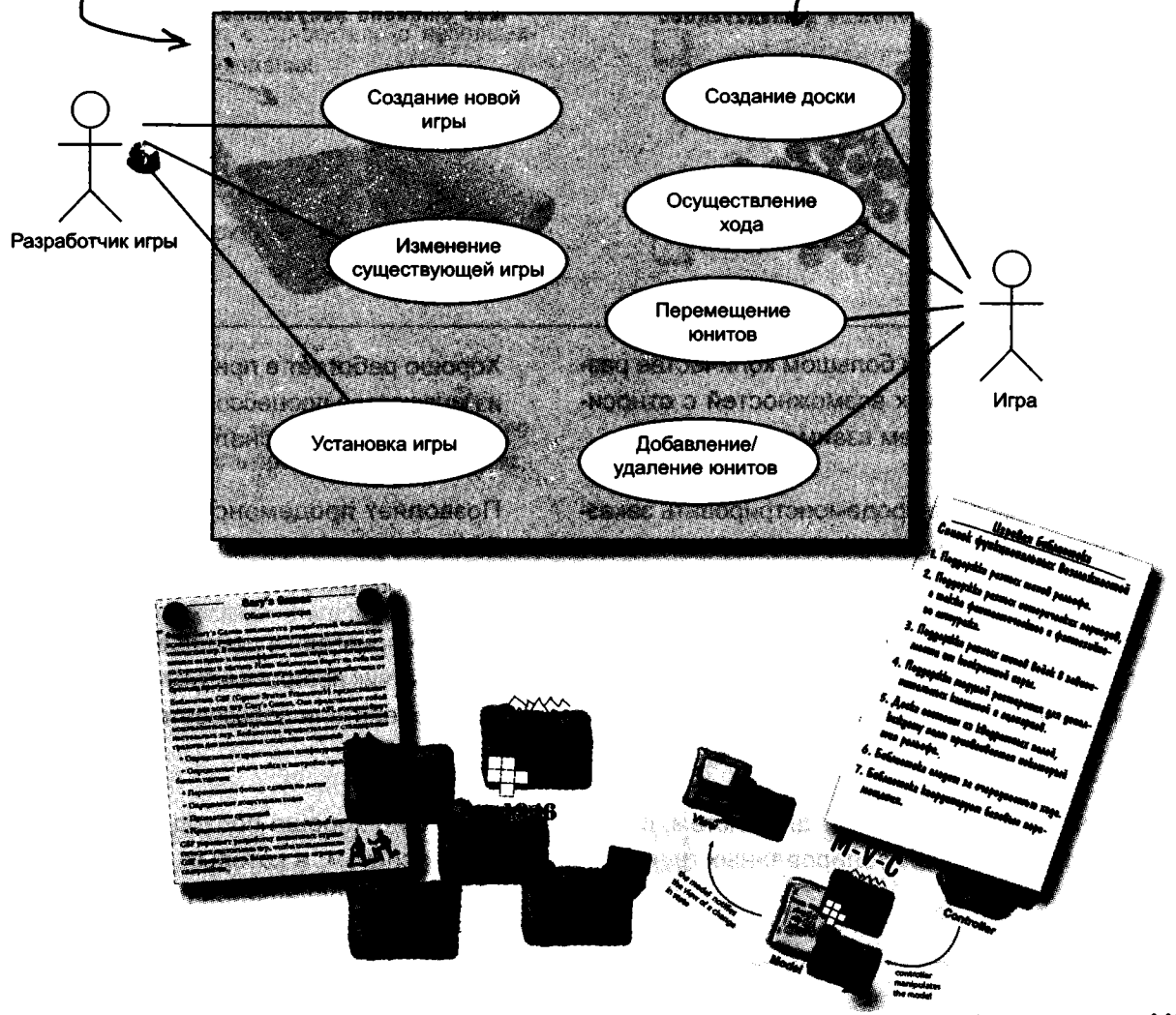
Все эти планы и диаграммы задействованы, но в своей работе вы руководствуетесь списком функциональных возможностей.

Сценарно-ориентированная разработка

При сценарно-ориентированной разработке вы работаете над реализацией отдельного сценария в варианте использования. Затем вы берете следующий сценарий и работаете над ним, пока не будут завершены все сценарии некоторого варианта использования. Далее вы переходите к следующему варианту и т. д., пока не начнут работать все варианты использования.

При сценарно-ориентированной разработке вы руководствуетесь диаграммой сценариев использования, на которой представлены разные сценарии использования в вашей системе.

Например, можно взять сценарий использования «Создание доски» и последовательно написать код реализации всех вариантов этого сценария.

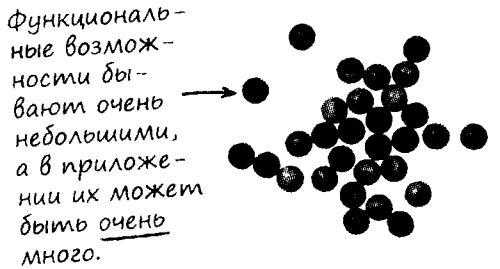


Два метода разработки

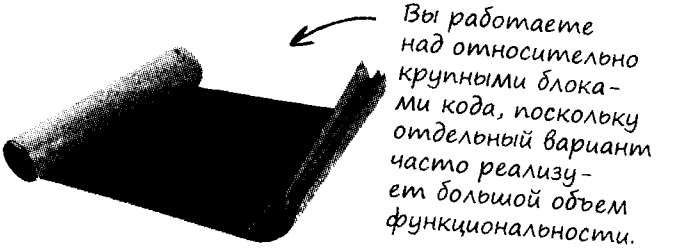
На самом деле существует великое множество разных методов итеративного перебора и построения компонентов вашего приложения. Впрочем, большинство таких методов относится к одной из двух представленных категорий. Так как же решить, какой метод следует использовать в каждом конкретном случае?

Чем функционально-ориентированная разработка отличается от сценарно-ориентированной?

Функционально-ориентированная разработка обладает большей детализацией



Сценарно-ориентированная разработка в большей степени направлена на «общую картину»



Хорошо работает при большом количестве разных функциональных возможностей с относительно низким уровнем взаимодействия.

Позволяет быстрее продемонстрировать заказчику работающий код.

Сильно привязан к функциональности: маловероятно, чтобы при этом методе вы забыли о какой-нибудь функциональной возможности.

Особенно хорошо подходит для систем, состоящих из множества изолированных функциональных блоков.

Хорошо работает в приложениях, состоящих из множества процессов и вариантов (вместо отдельных функциональных блоков).

Позволяет продемонстрировать заказчику более крупные функциональные блоки на каждой стадии разработки.

Сильно привязан к действиям пользователя: разработчик программирует разные варианты взаимодействия пользователя с системой.

Особенно хорошо подходит для транзакционных систем, в которых работа системы в основном определяется продолжительными, сложными процессами.

НАЗОВИТЕ МЕТОД

Ниже приведены утверждения, каждое из которых относится к одному из методов итеративной разработки компонентов вашей системы. Определите, к какому методу относится каждое утверждение. Учтите, что в некоторых случаях утверждение может быть справедливо для обоих методов.

	Сценарно-ориентированный	Функционально-ориентированный
Метод работает с относительно небольшими блоками приложения.	☐	☐
Метод концентрируется на отдельной части приложения.	☐	☐
Метод реализует завершенный процесс (последовательность операций) в приложении.	☐	☐
При использовании этого метода всегда можно проверить, реализована ли часть приложения, над которой вы работаете.	☐	☐
При использовании этого метода вы руководствуетесь диаграммой, а не списком.	☐	☐

НАЗОВИТЕ МЕТОД

Решение упражнения

Ниже приведены утверждения, каждое из которых относится к одному из методов итеративной разработки компонентов вашей системы. Определите, к какому методу относится каждое утверждение. Учтите, что в некоторых случаях утверждение может быть справедливо для обоих методов.

	Сценарно-ориентированный	Функционально-ориентированный
Метод работает с относительно небольшими блоками приложения.	☐	☑
Метод концентрируется на отдельной части приложения.	☑	☑
Метод реализует завершенный процесс (последовательность операций) в приложении.	☑	☐
При использовании этого метода всегда можно проверить, реализована ли часть приложения, над которой вы работаете.	☑	☑
При использовании этого метода вы руководствуетесь диаграммой, а не списком.	☑	☐

Использование функционально-ориентированной разработки

Пока Гэри не вышел из себя, давайте займемся функционально-ориентированной разработкой: мы берем одну функциональную возможность и прорабатываем ее вплоть до завершения. Это не должно занять столько времени, как для написания кода реализации целого варианта использования.

Когда ваш заказчик с нетерпением ожидает результатов, используйте функционально-ориентированную разработку. Начните с той функциональной возможности, над которой вы уже поработали.

Возвращаемся к списку функциональных возможностей из глав 6 и 7.

Игровая библиотека

Список функциональных возможностей

1. Поддержка разных типов рельефа.
2. Поддержка разных исторических периодов, а также фантастического и фэнтезийного антуража.
3. Поддержка разных типов войск в зависимости от конкретной игры.
4. Поддержка модулей расширения для дополнительных кампаний и вариантов.
5. Доска состоит из квадратных полей, каждому полю присваивается некоторый тип рельефа.
6. Библиотека следит за очередностью хода.
7. Библиотека координирует базовые перемещения.

Диаграмма классов для Unit уже готова, давайте напишем код этого класса и вычеркнем пункт № 3.

Unit
type: String properties: Map
setType(String) getType(): String setProperty(String, Object) getProperty(String): Object

Мы также знаем, что большинство других функциональных возможностей зависит от этого класса, что делает его еще более подходящим кандидатом для начала работы.

МОЗГОВОЙ ШТУРМ

А если бы вы решили пойти по пути сценарно-ориентированной разработки, то с чего бы вы начали?

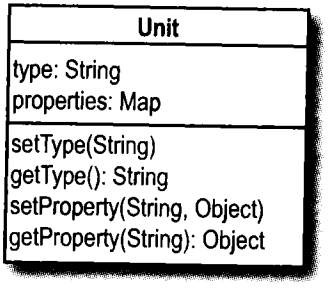
Анализ функциональной возможности

После выбора функциональной возможности, с которой начинается работа, следует провести дополнительный анализ. Начнем с того, что было записано в списке функциональных возможностей:

3. Поддержка разных типов войск в зависимости от конкретной игры.

То, что у нас есть... Но это довольно общее описание того, что нам предстоит запрограммировать.

Также у нас имеется диаграмма классов из главы 7:



Похоже на хорошую заготовку для класса Unit. Так все ли на месте?

Итак, у нас вроде бы есть все необходимое для перехода к программированию? А чтобы лишний раз убедиться в том, что мы ничего не забыли, проведем небольшой текстологический анализ.

У нас нет варианта использования, который можно было бы проанализировать, но мы можем вернуться к общей концепции игровой библиотеки Гэри и проверить, было ли учтено все, что хочет Гэри от своих юнитов.

Так выглядела общая концепция Гэри из главы 6.

Gary's Games

Общая концепция

Фирма Gary's Games занимается разработкой библиотек, используемых разработчиками для создания пошаговых стратегических игр. В отличие от аркадных «стрелялок» и игр, основанных на аудио- и видеоэффектах, наши игры ориентируются на стратегию и тактику. Наша библиотека берет на себя всю рутинную работу по созданию игры, избавляя разработчика от бремени программирования стандартных задач.

Библиотека GSF (Ggame System Framework) предоставляет основу для всех игр Gary's Games. Она представляет собой библиотеку классов с четко определенным API, которые будут использоваться всеми группами, занимающимися разработкой настольных игр. Библиотека предоставляет стандартные средства для выполнения следующих операций:

- Определение и представление конфигурации доски
- Определение родов войск и настройка армий или других боевых единиц
- Перемещение боевых единиц по доске
- Определение допустимых ходов
- Проведение сражений
- Предоставление информации о боевой единице

GSF упрощает разработку пошаговых стратегических настольных игр, чтобы пользователи GSF могли уделить больше времени игровому содержанию.

Сравните диаграмму класса Unit с общей концепций. Не было ли что-то упущено на диаграмме?

Что еще захочет увидеть Гэри, когда вы скажете: «Я дописал код управления юнитами для вашей библиотеки»?

Реализация класса Unit

Фактически наша диаграмма классов показывает лишь то, как должны представляться свойства юнитов. Но в своей общей концепции Гэри предполагает, что в его игровой библиотеке юниты будут способны на нечто большее, чем простая поддержка нестандартных свойств для конкретной игры.

И это вполне логично, потому что ключевые возможности, на которых мы сосредоточились в главе 7, относились не ко всему классу Unit, а только к свойствам Unit для конкретной игры.

Ниже перечислено то, что должны делать юниты в игровой системе Гэри:

1. **Каждый юнит должна обладать свойствами, причем разработчики игры должны иметь возможность добавлять новые свойства юнитов.**

Сейчас наша диаграмма классов ориентирована на этот конкретный аспект класса Unit.

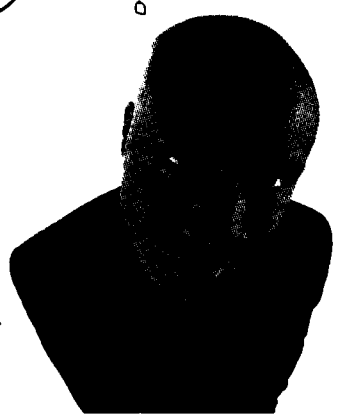
2. **Юниты должны уметь перемещаться с одного поля доски на другое.**

Вы должны примерно представлять, как это делается, после работы над аналогичной ключевой функцией из главы 7.

3. **Юниты должны уметь объединяться в армии.**

Все эти новые возможности сформулированы на основе общей концепции Гэри.

Замечательно: **еще один** список того, что вы собираетесь сделать. Слушайте, я вам доверяю, и все такое, но я должен увидеть нечто большее, чем листок бумаги с рисунком.



Гэри не впечатляют варианты использования и списки... Как бы убедить его, что мы работаем над выполнением его требований?

Демонстрация класса Unit

Мы поработали над поддержкой юнитов, привязанных к конкретной игре, и механизмом поддержки свойств **Unit** в главе 7. Но чтобы убедить Гэри, что мы сделали действительно что-то стоящее, одной диаграммы классов будет недостаточно.

Unit
type: String properties: Map
setType(String) getType(): String setProperty(String, Object) getProperty(String): Object

Возможно, этого будет достаточно для начала работы над реализацией класса Unit, но вряд ли мы убедим Гэри, что в игровой библиотеке уже имеется полноценная реализация юнитов.

Как насчет тестирования? Сможете ли вы показать мне, что юниты имеют свойства, могут перемещаться и объединяться в армии? Я хочу видеть работающий код.

Заказчик хочет видеть то, что ему понятно

Ваши заказчики привыкли видеть программы, выполняемые на компьютере. Диаграммы и списки помогут им убедиться в том, что вы одинаково понимаете требования и ту систему, которую вы строите, но чтобы они поняли, что вы делаете что-то действительно полезное, потребуется нечто большее.

Определите тестовые сценарии, которые можно показать заказчику. Эти сценарии убедят его в том, что ваш код работает, и делает именно то, чего хочет заказчик.

Написание тестовых сценариев

Будьте внимательны... Это не те «сценарии», о которых мы говорили в контексте вариантов использования.

Тестовые сценарии не обязаны быть слишком сложными; они просто демонстрируют заказчику, что функциональность ваших классов работает правильно.

Для свойств юнитов можно начать с простого тестового сценария, который создает экземпляр **Unit** и включает в него новое свойство. Достаточно показать заказчику программу, которая выдает результат следующего вида:

При всей своей простоте этот тест покажет заказчику, что написанный вами код действительно работает.

Начинаем с создания экземпляра Unit...

...затем задаем некоторые свойства...

...наконец, читаем свойства и убеждаемся в том, что прочитанные значения совпадают с заданными.

```
File Edit Window Help ItWorks
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set "type" to "infantry"
...Set "hitPoints" to 25
...Getting unit type: "infantry"
...Getting unit hitPoints: 25

Test complete.
```

Запишите внизу результаты, которые должен увидеть Гэри.

СТАНЬ заказчиком

У нас уже есть один тестовый сценарий. Представьте себя на месте Гэри и придумайте еще два тестовых сценария, которые продемонстрируют, что класс Unit работает как положено. Запишите выходные данные каждого сценария в окне справа.

```
File Edit Window Help Scenario2
%java UnitTester
Testing the Unit class...
_____________________________
_____________________________
_____________________________
_____________________________
_____________________________

Test complete.
```

```
File Edit Window Help Scenario3
%java UnitTester
Testing the Unit class...
_____________________________
_____________________________
_____________________________
_____________________________
_____________________________

Test complete.
```

СТАНЬ заказчиком. Решение

Вам было поручено придумать еще два тестовых сценария, которые продемонстрируют, что класс Unit работает как положено. Вот два тестовых сценария, которые получились у нас:

Свойства юнитов | Группы юнитов

```
File Edit Window Help ItWorks
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set "type" to "infantry"
...Set "hitPoints" to 25
...Getting unit type: "infantry"
...Getting unit hitPoints: 25

Test complete.
```

Первый тестовый сценарий проверяет чтение и запись значений свойств.

Тестовый сценарий № 2: изменение значений свойств

Мы решили протестировать задание значений свойств с их последующим изменением. Например, если задать свойство **hitPoints**, а затем изменить его, то обращение к свойству **hitPoints** должно вернуть более позднее значение этого свойства:

Второй сценарий похож на первый, но он тестирует изменение значения свойства (вместо простого присваивания и чтения).

Тестирование всегда начинается с создания нового экземпляра Unit.

Затем мы задаем значение hitPoints и изменяем его.

```
File Edit Window Help Scenario2
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set "hitPoints" to 25
...Set "hitPoints" to 15
...Getting unit hitPoints: 15

Test complete.
```

В завершение мы убеждаемся в том, что hitPoints содержит обновленное значение, а не исходное значение 25.

Тестовый сценарий № 3: обращение к несуществующему свойству

В третьем сценарии мы решили протестировать, что произойдет при выборке значения свойства, которое не было задано. Такие ошибки встречаются сплошь и рядом, и мы не хотим, чтобы программа «падала» каждый раз, когда разработчик игры допустит незначительную опечатку в коде. Вот как мы протестировали эту ситуацию:

Тест показывает заказчику, что мы не ограничиваемся только «хорошими» случаями, а думаем еще и о том, что делать в непредвиденной ситуации.

Снова начинаем с создания нового экземпляра Unit.

Задаем свойство hitPoints (нормальное использование Unit).

Пытаемся обратиться к свойству, значение которого не задано.

Наконец, убеждаемся в том, что класс Unit корректно обрабатывает обращения к несуществующим свойствам.

```
File Edit Window Help Scenario3
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set "hitPoints" to 25
...Getting unit strength: [no value]
...Getting unit hitPoints: 25

Test complete.
```

Всегда тестируйте свои программы на все возможные варианты использования, которые только сможете придумать. Действуйте творчески!

Не забывайте тестировать ситуации некорректного использования программных продуктов. Вы сможете обнаружить многие ошибки на ранней стадии — заказчики будут довольны.

Часто задаваемые вопросы

В: Но у нас пока не написано ни строки кода. Мы не забегаем вперед, начиная с тестов?

О: Вовсе нет. Если тесты будут известны до написания кода, вам будет проще разобраться, какой код потребуется для прохождения этих тестов. С этими тремя тестовыми сценариями мы легко напишем код класса `Unit`, а тесты точно определят поведение нашего кода.

В: Кажется, это называется «разработкой через тестирование»?

О: В целом — да. Формально разработка через тестирование ориентирована на автоматизированные тесты, и в них обычно используются тестовые системы (например, JUnit). Но основная идея разработки через тестирование заключается как раз в написании тестовых сценариев до написания кода.

В: Так мы используем разработку через тестирование или функционально-ориентированную разработку? Я совсем запутался...

О: И то и другое. Как правило, в качественном анализе и проектировании программных продуктов смешано много разных методов. Можно начать с вариантов использования (сценарно-ориентированная разработка), потом выбрать в этом варианте небольшую функциональную возможность и начать работать над ней (форма функционально-ориентированной разработки). А потом вы можете использовать тесты для определения реализации этой возможности (разработка через тестирование).

В: Почему тесты такие простые? Я ожидал чего-то более интересного.

О: Тесты должны быть простыми, и каждый из них должен тестировать небольшой функциональный блок. Если вы начнете проверять сразу несколько условий, вам будет труднее понять, что именно стало причиной сбоя. Лучше написать больше тестов, но чтобы каждый из них был сконцентрирован на очень четко определенном блоке функциональности.

В: И каждый тест проверяет правильность работы одного метода класса?

О: Нет, каждый тест концентрируется на одном блоке функциональности — это может быть один метод или несколько методов. Например, невозможно протестировать задание значения свойства (метод `setProperty()`) без чтения значения этого свойства (метод `getProperty()`). Таким образом, в одном блоке функциональности — задание свойства — задействованы два метода.

В: Почему мы тестируем чтение свойства, которое не было задано? Ведь это же неправильное использование класса Unit?

О: Протестировать неправильное использование вашего кеда обычно не менее (а то и более) важно, чем протестировать его правильное использование. Разработчики игр могут неправильно записать имя свойства, или написанный ими код будет рассчитывать на то, что свойство было задано в другом месте и запросит несуществующее свойство. Вы должны знать, что будет происходить в таких ситуациях.

В: Тесты спланированы, и мы можем переходить к программированию класса Unit?

О: Вообще-то сначала необходимо решить еще одну структурную проблему...

Разработка через тестирование направлена на реализацию правильного поведения классов.

Я тут посмотрел на нашу диаграмму класса Unit... Не уверен, что это лучшее решение. Мне кажется, что мы немного ошиблись с анализом общности.

Unit
type: String properties: Map
setType(String) getType(): String setProperty(String, Object) getProperty(String): Object

Текущая диаграмма классов, которую обсуждают Фрэнк, Джилл и Джо.

Джо: О чем ты? Мы определили, что каждое свойство в Unit должно иметь имя и значение. И мы решили использовать Map для хранения свойств.

Фрэнк: И разработчики игр могут добавлять новые свойства. Для этого достаточно определить новое имя свойства и занести пару «имя/значение» в контейнер Map методом setProperty().

Джилл: Верно. А также мы добавили свойство type, потому что у всех юнитов должен быть тип — общая характеристика...

Джо: Вот именно. Так что с анализом общности все в порядке.

Джилл: ...но мы также знаем, что юниты могут объединяться в группы — армии, флоты и так далее. И что произойдет, если в одной группе окажутся два юнита одного типа... Как нам их различить?

Фрэнк: Ты хочешь сказать, что нам понадобится какой-то идентификатор?

Джилл: Да, вероятно. Или хотя бы имя... Но даже в этом случае свойство name не предотвратит возможного совпадения, верно?

Джо: Согласен, но это не означает, что нам придется менять структуру. Свойство ID можно просто добавить в контейнер Map. Для обращения ко всем свойствам существует хороший единый механизм — метод getProperty().

Фрэнк: Он прав, Джил. А поскольку подробности реализации имен свойств инкапсулируются в контейнере Map, мы даже можем переключиться с идентификатора на имя или выбрать совсем другой признак, и код, использующий класс Unit, от этого особенно не изменится... Нужно будет только указать новое имя свойства в getProperty(). Элегантное решение!

Джилл: А как насчет общности? Если идентификатор действительно является общим для всех разновидностей Unit, разве не нужно вывести его из Map, — по аналогии с тем, как было сделано со свойством type?

Джо: Вот оно что... Инкапсуляция *или* общность. Тяжелый выбор... Похоже, мы не можем выбрать одно, не нарушив другого.

Возьми в руку карандаш

Уточнение анализа общности

Ниже перечислены разные свойства, которыми могут обладать разные юниты. Запишите на листке «Общность» те свойства, которыми, по вашему мнению, должны обладать все юниты независимо от типа. На листке «Изменчивость» должны быть перечислены свойства, присущие только некоторым, отдельным разновидностям юнитов.

Потенциальные свойства юнитов

name	weapon	allegiance	gear
type	hitPoints	wingspan	lastName
strength	weight	landSpeed	id
speed	intelligence	experience	groundSpeed
stamina	firstName	weapons	hunger

Здесь записываются общие свойства, относящиеся ко всем юнитам без исключения.

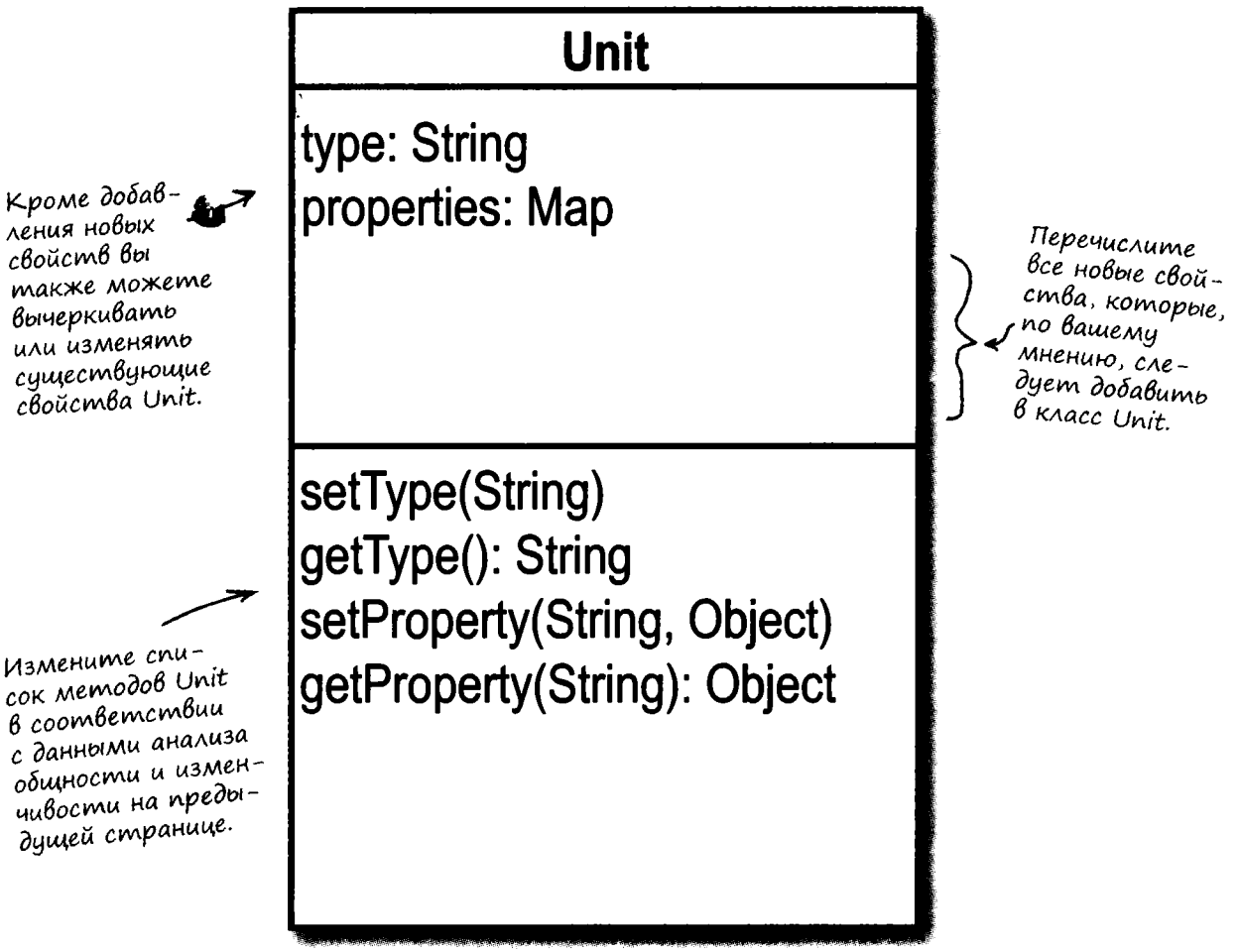
Общность

На этом листке записываются свойства, относящиеся к конкретным разновидностям юнитов.

Изменчивость

Теперь обновите диаграмму классов Unit

После уточненного анализа общности на с. 462 вам (возможно) захочется внести изменения в диаграмму Unit, изображенную ниже. Внесите любые изменения, улучшающие структуру класса Unit, и добавьте примечания, которые напомнят вам, для чего было внесено то или иное дополнение.



Возьми в руку карандаш
Решение

Свойства юнитов

Уточнение анализа общности.

На с. 462 был приведен список свойств, которыми могут обладать разные юниты. Вам было предложено записать на листке «Общность» те свойства, которыми, по вашему мнению, должны обладать все юниты независимо от типа. На листке «Изменчивость» должны быть перечислены свойства, присущие только некоторым, отдельным разновидностям юнитов.

Безусловно, свойство «type» должно быть записано на этом листке: как мы выяснили в главе 7, типом обладают все юниты без исключения.

Свойства «name» и «id» являются достаточно общими. Мы подумали, что оба свойства должны присутствовать у всех юнитов.

Общность

- type
- name
- id
- weapons

Свойств, присущих всем разновидностям юнитов, не так уж много, поэтому список общих свойств получился коротким.

С включением в список свойства «weapons» дело обстоит не столь однозначно. Мы посчитали, что в военных играх каждый юнит должна обладать хотя бы одним видом оружия, но некоторые могут оснащаться несколькими видами. Соответственно обобщенное свойство «weapons» подойдет для всех типов юнитов.

Также было решено, что одиночное свойство «weapon» нам не понадобится — юниты с одним видом оружия просто будут содержать один экземпляр оружия в свойстве «weapons». И мы отказались от этого свойства.

~~weapon~~

Расслабьтесь

Возможно, у вас будет свое мнение по поводу набора свойств.

Возможно, вы играли в другие игры и представили свой набор общих свойств. Ничего страшного, но обязательно подумайте над тем, как и почему вы принимали решение. Вам необходимо понять, почему нами были приняты такие решения, потому что они будут использоваться в оставшейся части этой главы.

Большинство свойств было включено в список «Изменчивость», потому что эти свойства применимы только к некоторым разновидностям юнитов.

В принципе свойство hitPoints могло бы стать общим для всех юнитов и попасть в список «Общность». Мы оставили его в списке «Изменчивость», потому что некоторые виды техники (скажем, танки или самолеты) не имеют такого однозначного представления «здоровья», как юниты, представляющие людей.

Изменчивость

strength
speed
stamina
hitPoints
weight
intelligence
firstName
allegiance
wingspan
landSpeed
experience
gear
lastName
groundSpeed
hunger

Эти свойства применимы либо к людям, либо к технике, но не к обоим видам юнитов одновременно.

Часто задаваемые вопросы

В: У меня в список «Общность» не попало ни одно свойство, кроме «type». Где я ошибся?

О: А вы не ошиблись. Суть анализа и проектирования заключается в принятии решений; иногда ваши решения будут отличаться от решений других программистов. В этом нет ничего плохого, при условии, что принятое решение имеет хорошо продуманную основу.

В: Но разве разные решения не приведут к разному коду и разным реализациям структуры?

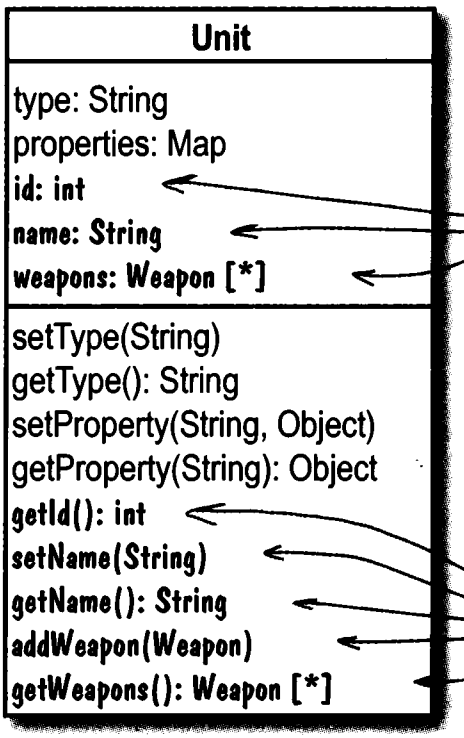
О: Конечно, приведут. Но целью ООАП и разработки ПО вообще не является поиск единственно правильного решения, во многих случаях однозначно «правильного» или «неправильного» решения вообще не существует. Цель — создание хорошо спроектированного решения, а добиться этого можно разными способами. Более того, даже если два программиста в этом упражнении примут одинаковые решения по поводу общности и изменчивости, это может привести к совершенно разным решениям, когда дело дойдет до фактического написания классов. Давайте на секунду предположим, что два разработчика выдали одинаковые списки свойств в двух категориях, а потом попытались пересмотреть структуру класса Unit в соответствии со своими результатами...

Решение № 1: прежде всего общность

Свойства юнитов

Я выделил свойства, общие для всех юнитов, в отдельные переменные и методы, а переменные свойства оставил в контейнере Map.

Мы уже встречались с Сэмом в главе 4, когда он работал над собачьей дверью для Марии и Рэнди.



Все свойства, общие для всех юнитов, представляются специальными переменными, не входящими в контейнер Map.

Сэм решил, что идентификатор будет задаваться в конструкторе Unit, поэтому метод setId() не нужен.

Для каждого из новых свойств создается набор методов.

В этом решении все разработчики игр могут напрямую обращаться к свойствам **id**, **name** и **weapons** вместо использования **getProperty()** и работы с более общим контейнером **Map**.

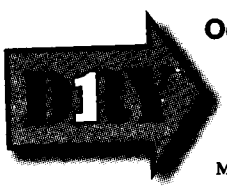
Здесь принципиально то, что свойства, ***общие*** для всех разновидностей **Unit**, хранятся вне контейнера **Map**. В контейнере объединяются только свойства, состав которых ***изменяется***.

Бесспорно, вы выявили сходство между разными видами юнитов, но как насчет инкапсуляции? На мой взгляд, класс Unit не кажется таким уж устойчивым к изменениям.

Со времени нашей последней встречи в главе 4 Рэнди узнал много нового об ОО-проектировании.

Решения из области проектирования всегда подразумевают компромисс

Сэм решил выделить аспекты, общие для всех разновидностей **Unit**. Однако у решения Сэма есть свои недостатки:

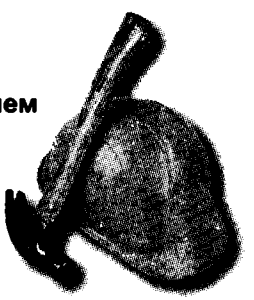


Осторожно: дублирование!

Теперь у нас появилось два разных способа обращения к свойствам: с использованием **getId()**, **getName()** и других методов конкретных свойств и с использованием метода **getProperty()**. Существование двух механизмов почти наверняка приведет к дублированию кода.

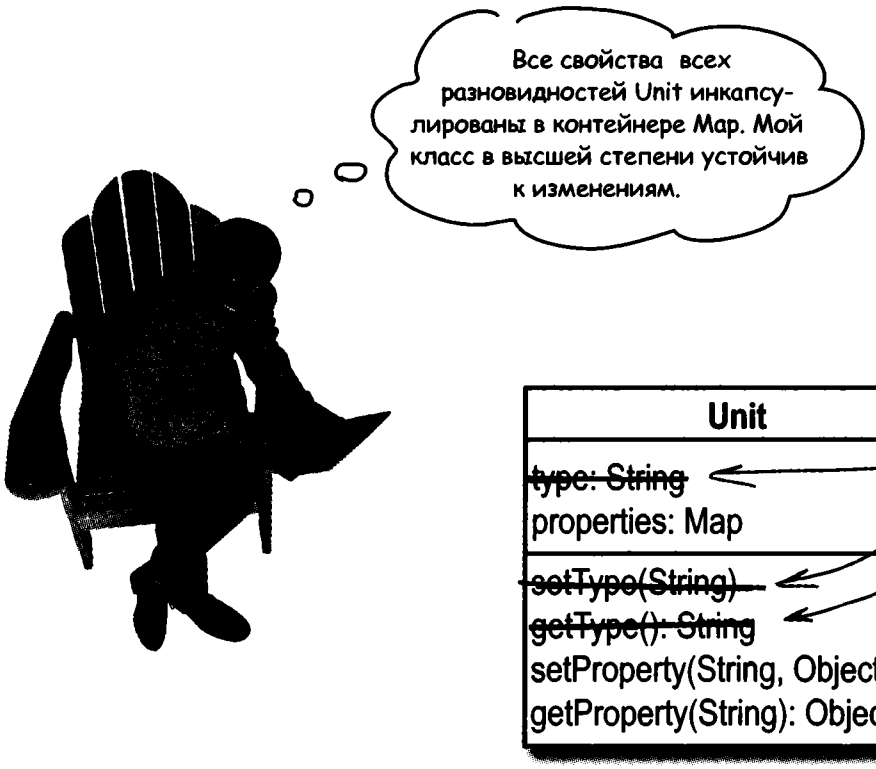
Потенциальная опасность дублирования кода почти всегда указывает на проблемы с гибкостью и сопровождением кода.

Трудности с сопровождением



Теперь имена некоторых свойств (такие, как **id** и **name**) жестко фиксированы в классе **Unit**. Если разработчик игры захочет изменить их, то это создаст изрядные проблемы и потребует внесения изменений в класс **Unit**. Обычно в таких ситуациях помогает инкапсуляция, и на этом мы подходим к решению Рэнди...

Решение № 2: прежде всего инкапсуляция



Рэнди не стал добавлять новые свойства и методы. А еще он избавился от переменной type и метода getType(), переместив это свойство в контейнер Map.

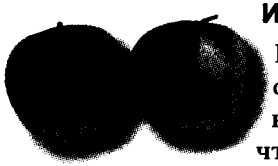
Это решение ориентировано на инкапсуляцию всех свойств **`Unit`** в контейнере **`Map`** и предоставление стандартного интерфейса (метод **`getProperty()`**) для обращения ко всем свойствам. Даже к свойствам, присутствующим во всех юнитах (таким, как **`type`** и **`id`**), в этом решении обращение осуществляется через контейнер **`Map`**.

В этом решении центральное место занимает инкапсуляция и гибкость структуры. Даже в случае изменения имен общих свойств класс **`Unit`** останется неизменным, так как имена свойств не фиксируются в самом классе.

Но вы полностью игнорируете общность разных видов Unit. И как разработчик игры узнает, что свойства name, type, id и weapons были задуманы как стандартные?

И это решение не идеально...

Решение Рэнди более устойчиво к изменениям и обеспечивает более высокий уровень инкапсуляции, но и у него есть свои «минусы». Вот лишь некоторые из них:



Игнорирование общности

Рэнди инкапсулировал все свойства в контейнере **Map**, но теперь ничто не указывает, что свойства **type**, **name**, **id** и **weapons** задуманы как общие для всех разновидностей **Unit**.

Лишняя работа во время выполнения

Метод **getProperty()** возвращает экземпляр **Object**, который необходимо будет преобразовать к правильному типу для каждого свойства, и все это во время выполнения. Преобразование типов создает большое количество дополнительной работы на стадии выполнения даже для свойств, общих для всех типов **Unit**.

МОЗГОВОЙ ШТУРМ

Как вы думаете, какое из решений лучше? А может, в каких-то ситуациях лучше подойдет одно решение, а в других — другое?

Воспользуемся решением, ориентированным на общность

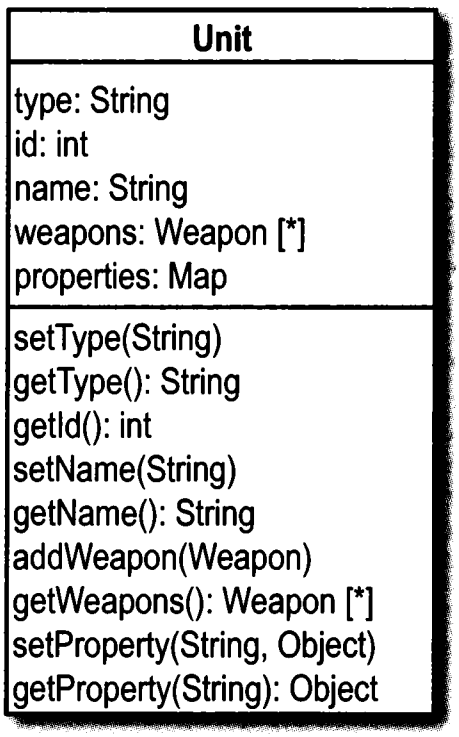
Свойства юнитов

Все еще остается частью более общей задачи реализации свойств юнитов.

Давайте используем для игровой библиотеки решение Сэма, которое выделяет общие свойства **Unit** в отдельные свойства и методы, а все остальные свойства хранятся в контейнере **Map**.

Свойства, общие для всех юнитов.

Остальные свойства, специфические для конкретных юнитов или игры, объединяются в этом контейнере Map.



Часто задаваемые вопросы

В: Мне показалось, что другое решение (ориентированное на инкапсуляцию) лучше. Это нормально?

О: Абсолютно нормально. Оба решения имели свои положительные стороны, и каждое могло хорошо сработать. Единственное, чего нельзя делать, так это отказываться от изменения структуры (какое бы из решений ни было выбрано), если она в конечном итоге будет плохо работать. На каждой стадии работы над приложением следует заново оценить принятые решения и убедиться в том, что они все еще прочны.

В: Как определить, когда мне необходимо изменить свое решение? Мой код работать не перестанет, тогда на что следует обратить внимание?

О: Ключевой момент — итеративный характер разработки. Многие решения прекрасно смотрятся на одной стадии, а потом начинают создавать проблемы, когда вы начинаете углубленно работать над другой частью вашего приложения. Если решение работает, а вы можете использовать проверенные ОО-принципы и применять паттерны проектирования, значит, все идет хорошо. Но если у вас начинаются проблемы, не бойтесь изменять и перерабатывать свои решения.

В: А если я не могу выбрать между парой хороших решений?

О: Выбор приходится делать всегда, даже если вы не уверены на 100% в его правильности. Всегда лучше выдать наилучшее предположение, чем проводить бесконечные часы за обсуждением того или иного варианта; аналитический паралич — верный путь к провалу. Гораздо лучше начать двигаться по какому-то пути, даже если вы не уверены в его правильности, и выполнить часть работы, чем не принимать никакого решения.

Хорошие программы строятся итеративным методом. Анализируйте, проектируйте, а затем повторяйте итерации, работая над все более мелкими частями приложения.

При каждой итерации заново оценивайте свои решения и не бойтесь ИЗМЕНИТЬ что-либо, если это оправданно для вашего решения.

Сопоставьте тесты с разработанной структурой

У нас имеются тестовые сценарии, которые нужно показать Гэри, и структура класса **Unit**. Перед написанием кода осталось сделать еще один шаг: убедиться в том, что разработанная нами структура **Unit** позволит написать код, который пройдет все тесты.

Unit

type: String
id: int
name: String
weapons: Weapon [*]
properties: Map

setType(String)
getType(): String
getId(): int
setName(String)
getName(): String
addWeapon(Weapon)
getWeapons(): Weapon [*]
setProperty(String, Object)
getProperty(String): Object

Этот класс должен содержать методы, позволяющие выполнить все операции из тестов.

Создание экземпляра сводится к простому вызову «new Unit()».

Мы используем setType()и getType(), так как это общее свойство для всех разновидностей Unit.

```
File Edit Window Help ItWorks
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set type to "infantry"
...Set hitPoints to 25
...Getting unit type: "infantry"
...Getting unit hitPoints: 25

Test
```

Проверка изменения значения свойства; это означает всего лишь дополнительный вызов setProperty(), так что с этим сценарием все в порядке.

```
File Edit Window Help Scenario2
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set hitPoints to 25
...Set hitPoints to 15
...Getting unit hitPoints: 15

Test compl
```

Не общие свойства (например, hitPoints), читаются и записываются с помощью методов setProperty() и getProperty().

```
File Edit Window Help Scenario3
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set hitPoints to 25
...Getting unit strength: [no value]
...Getting unit hitPoints: 25

Test complete.
```

Для этого теста необходимо вызвать getProperty(«strength») без предварительного задания свойства «strength» и посмотреть, что получится.

Пишем класс Unit

Unit.java

Мы шли к этому целых две главы! Но теперь у нас есть все необходимое для написания кода класса **Unit**. Вот как это сделали мы:

```
package headfirst.gsf.unit;

public class Unit {
  private String type;
  private int id;
  private String name;
  private List weapons;
  private Map properties;

  public Unit(int id) {
    this.id = id;
  }

  public int getId() {
    return id;
  }

  // Методы getName() и setName()
  // Методы getType() и setType()

  public void addWeapon(Weapon weapon) {
    if (weapons == null) {
      weapons = new LinkedList();
    }
    weapons.add(weapon);
  }

  public List getWeapons() {
    return weapons;
  }

  public void setProperty(String property, Object value) {
    if (properties == null) {
      properties = new HashMap();
    }
    properties.put(property, value);
  }

  public Object getProperty(String property) {
    if (properties == null) {
      return null;
    }
    return properties.get(property);
  }
}
```

Идентификатор передается в аргументе конструктора...

...поэтому необходим только метод getId() без парного метода setId().

Код простых методов доступа не приводится для экономии места.

Чтобы этот код компилировался, необходим простейший класс Weapon. Мы создали пустой класс, не содержащий методов.

Создание контейнера List откладывается до тех пор, пока в него не будет добавлено первое оружие. Тем самым экономится немного памяти, особенно при тысячах юнитов.

Как и в случае со списком оружия, создание контейнера HashMap для свойств откладывается до момента первого использования.

Так как переменная properties может быть неинициализированной, мы проверяем ее перед обращением к значению свойства.

Изучаем тестовые примеры...

Мы довольно много говорили о тестах, но до сих пор не видели ни одного из них. Давайте рассмотрим тестовый пример повнимательнее и выясним, как же создаются хорошие тесты.

Старайтесь не использовать имена вида test1, test2 и т. д. По возможности выбирайте содержательные имена.

1 **Каждый тест должен иметь идентификатор и имя.**

Имена тестовых примеров должны описывать то, что они тестируют. Имена вида «test+номер» далеко не так удобны, как имена вида `testProperty()` или `testCreation()`. Также используйте числовые идентификаторы, чтобы вы могли легко представить списки тестов (как на следующей странице).

В каждом блоке функциональности можно задействовать один метод, два метода, даже несколько классов... Но для начала сосредоточьтесь на простых блоках, по одному на тест.

2 **Каждый тест должен проверять один конкретный блок функциональности.**

Тесты должны быть атомарными: каждый тест должен проверять ровно один блок функциональности. Это позволит вам точно идентифицировать функциональность, которая создает проблемы в вашем приложении.

Если переменной hitPoints присваивается значение 15, то значение "15" должно стать входным параметром, передаваемым тестовому сценарию.

3 **У каждого теста должны быть входные данные.**

Тестовому примеру должно передаваться значение или набор значений, которые образуют входные данные. Обычно переданные данные используются для выполнения конкретного блока функциональности или аспекта поведения.

То, что должна выдавать программа. Скажем, если переменной type присваивается значение «infantry», а потом вызывается метод getType(), то ожидаемым результатом будет строка «infantry».

4 **У каждого теста должен быть предсказуемый результат.**

Что должна делать программа, класс или метод для заданных входных данных? Вы сравниваете фактический результат с ожидаемым, и если они совпали — значит, тест прошел успешно.

У класса Unit исходное состояние практически отсутствует. Мы должны создать новый экземпляр Unit, но это всё.

5 **У большинства тестов имеется исходное состояние.**

Перед проведением теста необходимо открыть подключение к базе данных, создать объект или задать значение. Все эти действия — часть исходного состояния тестового примера и должны быть выполнены до проведения фактического тестирования.

Возьми в руку карандаш

Разработайте тестовые примеры.

Ниже приведена таблица с пятью столбцами, по одному для каждой из пяти важных частей тестового примера. Заполните таблицу информацией о тестовых примерах, приводившихся ранее в этой главе. Чтобы упростить вашу задачу, мы заполнили первую строку, а также несколько пустых мест в остальных тестах.

Тест задает свойству «type» значение «infantry».

ID	Что тестируем	Входные данные	Ожидаемый результат	Исходное состояние
1	Запись/чтение свойства type	"type", "infantry"	"type", "infantry"	Существующий объект Unit
		"hitPoints", 25		
				Существующий объект Unit, hitPoints=25
4				

Помните о различиях между общими свойствами (такими, как type) и свойствами конкретных юнитов.

В этих трех сценариях тестируются четыре разных аспекта.

```
File Edit Window Help ItWorks
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set type to "infantry"
...Set hitPoints to 25
...Getting unit type: "infantry"
...Getting unit hitPoints: 25

Test complete.
```

Первый сценарий тестирует чтение и запись свойства «type».

```
File Edit Window Help Scenario2
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set hitPoints to 25
...Set hitPoints to 15
...Getting unit hitPoints: 15
```

```
File Edit Window Help Scenario3
%java UnitTester
Testing the Unit class...
...Created a new unit
...Set hitPoints to 25
...Getting unit strength: [no value]
...Getting unit hitPoints: 25

Test complete.
```

Возьми в руку карандаш
Решение

Разработайте тестовые примеры.

Ниже приведена таблица с пятью столбцами, по одному для каждой из пяти важных частей тестового сценария. Заполните таблицу информацией о тестовых примерах, приводившихся ранее в этой главе.

В большинстве тестов в качестве результата выдаются входные данные.

ID	Что тестируем	Входные данные	Ожидаемый результат	Исходное состояние
1	Чтение/запись общего свойства	"type", "infantry"	"type", "infantry"	Существующий объект Unit
2	Чтение/запись свойства конкретного юнита	"hitPoints", 25	"hitPoints", 25	Существующий объект Unit
3	Изменение значения существующего свойства	"hitPoints", 15	"hitPoints", 15	Существующий объект Unit, hitPoints=25
4	Чтение значения несуществующего свойства	N/A	"strength", нет	Существующий объект Unit без значения strength

Один тестовый сценарий должен работать с общими свойствами, а другой — со свойствами конкретных юнитов.

Смысл этого теста — не задавая значение свойства, попытаться прочитать результат.

А вы не забыли проверить в этом тестовом сценарии, что свойство не сохранило предыдущее значение?

Часто задаваемые вопросы

В: Почему три тестовых сценария превратились в четыре тестовых примера?

О: Потому что первый тестовый пример на самом деле тестирует два аспекта: чтение и запись общего свойства, имеющего собственную переменную и метод доступа (такой, как `getType()`), и чтение и запись свойства, специфичного для конкретной игры или юнита, а для обращения используется метод `getProperty()` (как в случае с `hitPoints`). Это два разных блока функциональности, для которых нужны разные тестовые примеры.

В: И все эти тесты докажут, что наша программа работает так, как ей положено?

О: Во всяком случае это хорошее начало, но помните: мы начали писать тесты, чтобы показать заказчику, что разрабатываемая программа действительно работает. Наши тестовые примеры продемонстрируют работоспособность кода, а также помогут найти в коде ошибки, пока мы еще не слишком далеко зашли в своем цикле разработки.

Головоломка

Итак, теперь вы знаете, что такое «тестовые примеры», и даже написали несколько таких примеров в табличной форме. Теперь можно написать тестовый класс, демонстрирующий заказчику, как работает программа, и подтверждающий, что в коде нет ошибок.

Проблема:

Гэри хочет убедиться в том, что поддержка юнитов для его игровой библиотеки не стоит на месте, а вы — что код, написанный вами для **`Unit.java`**, работает правильно.

Ваша задача:

1. Создайте новый класс с именем **`UnitTester.java`** и импортируйте в него класс **`Unit`** с сопутствующими классами.
2. Добавьте новый метод для каждого теста, представленного в таблице на с. 476. Не забудьте дать тестовым методам содержательные имена.
3. Каждый тестовый метод должен получать экземпляр **`Unit`** с настроенным исходным состоянием, а также параметры, которые необходимы для тестирования и сравнения фактического результата с ожидаемым.
4. Тестовый метод должен присваивать выбранное значение свойству с заданным именем переданного объекта **`Unit`**, а затем читать значение свойства с данным именем.
5. Если входное значение совпадает с выходным, метод выводит сообщение «Test passed»; в случае несовпадения выводится строка «Test failed» вместе с несовпадающими значениями.
6. Напишите метод **`main()`**, который устанавливает начальное состояние для каждого теста и запускает тесты на выполнение.

Дополнительное задание:

1. Некоторые аспекты класса **`Unit.java`** не тестируются сценариями на с. 476. Определите, о чем идет речь, создайте для каждого аспекта тестовый метод.
2. Выполните эти тесты из метода **`main()`**.

Головоломка. Решение

Класс, написанный нами для тестирования класса Unit.

```
public class UnitTester {

  public void testType(Unit unit, String type, String expectedOutputType) {
    System.out.println("\nTesting setting/getting the type property.");
    unit.setType(type);
    String outputType = unit.getType();
    if (expectedOutputType.equals(outputType)) {
      System.out.println("Test passed");
    } else {
      System.out.println("Test failed: " + outputType + " didn't match " +
        expectedOutputType);
    }
  }
```

Тестовые методы получают разные параметры, поскольку они тестируют разные аспекты класса Unit.

Большинство тестов завершается сравнением ожидаемого результата с фактическим.

Свойства, хранящиеся в Map, используют Object в качестве входных и выходных значений.

```
  public void testUnitSpecificProperty(Unit unit, String propertyName,
                    Object inputValue, Object expectedOutputValue) {
    System.out.println("\nTesting setting/getting a unit-specific
property.");
    unit.setProperty(propertyName, inputValue);
    Object outputValue = unit.getProperty(propertyName);
    if (expectedOutputValue.equals(outputValue)) {
      System.out.println("Test passed");
    } else {
      System.out.println("Test failed: " + outputValue + " didn't match " +
        expectedOutputValue);
    }
  }
```

Этот тест почти идентичен предыдущему, так как исходное состояние обеспечивает инициализацию свойства другим значением.

```
  public void testChangeProperty(Unit unit, String propertyName,
                    Object inputValue, Object expectedOutputValue) {
    System.out.println("\nTesting changing an existing property's value.");
    unit.setProperty(propertyName, inputValue);
    Object outputValue = unit.getProperty(propertyName);
    if (expectedOutputValue.equals(outputValue)) {
      System.out.println("Test passed");
    } else {
      System.out.println("Test failed: " + outputValue + " didn't match " +
        expectedOutputValue);
    }
  }
```

Предполагается, что исходное состояние было задано правильно... в противном случае тест ВСЕГДА будет завершаться неудачей.

UnitTester.java

Последнему тесту входное значение не требуется, потому что он тестирует обращение к свойству, не имеющему входного значения.

```
    public void testNonExistentProperty(Unit unit, String propertyName) {
      System.out.println("\nTesting getting a non-existent property's
value.");
      Object outputValue = unit.getProperty(propertyName);
      if (outputValue == null) {
        System.out.println("Test passed");
      } else {
        System.out.println("Test failed with value of " + outputValue);
      }
    }

    public static void main(String args[]) {
      UnitTester tester = new UnitTester();
      Unit unit = new Unit(1000);
      tester.testType(unit, "infantry", "infantry");
      tester.testUnitSpecificProperty(unit, "hitPoints",
                                      new Integer(25), new Integer(25));
      tester.testChangeProperty(unit, "hitPoints",
                                      new Integer(15), new Integer(15));
      tester.testNonExistentProperty(unit, "strength");
    }
  }
```

Свойству hitPoints в testUnitSpecificProperty() присваивается значение 25, поэтому мы можем вызвать testChangeProperty(), зная, что значение будет сброшено в этом тестовом примере.

Метод main() должен лишь создать новый экземпляр Unit, а затем выполнить тесты.

А теперь дополнительные задания...

Мы добавили в таблицу три новых тестовых примера для трех свойств, общих для всех юнитов, которые не были протестированы в UnitTester. Напишите три дополнительных тестовых метода по данным из таблицы. Удалось ли вам определить их самостоятельно?

ID	Что тестируем	Входные данные	Ожидаемый результат	Исходное состояние
1	Чтение/запись свойства type	"type", "infantry"	"type", "infantry"	Существующий объект Unit
2	Чтение/запись свойства конкретного юнита	"hitPoints", 25	"hitPoints", 25	Существующий объект Unit
3	Изменение значения существующего свойства	"hitPoints", 15	"hitPoints", 15	Существующий объект Unit, hitPoints = 25
4	Чтение значения несуществующего свойства	N/A	"strength", нет	Существующий объект Unit без значения strength
5	Чтение свойства id	N/A	1000	Существующий объект Unit, id = 1000
6	Чтение/запись свойства name	"name", "Damon"	"name", "Damon"	Существующий объект Unit
7	Добавление/чтение оружия	Объект Axe	Объект Axe	Существующий объект Unit

Этот тестовый пример не проверяет все общие свойства; он ограничивается свойством type.

Мы должны протестировать все три общих свойства, так как в каждом случае используются свои методы.

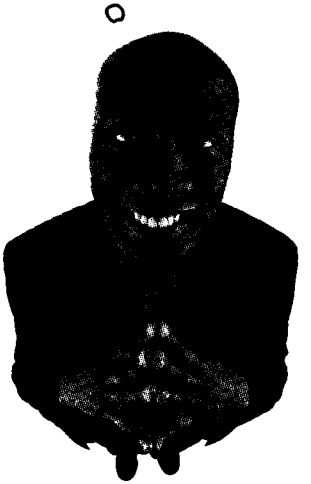
Представьте результат заказчику

Располагая классом **Unit** и набором тестовых примеров, вы можете продемонстрировать Гэри работоспособный код и доказать, что вы находитесь на верном пути к построению именно той системы, которая ему нужна. Давайте покажем ему результат выполнения тестового класса:

Тестовые классы не обязаны быть красивыми и интересными... Они просто доказывают, что ваша программа делает то, что ей положено делать.

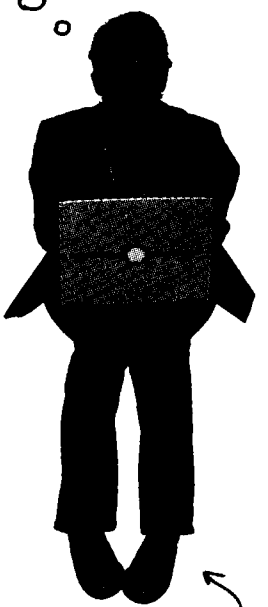
```
File Edit Window Help ProveToMe
%java UnitTester

Testing setting/getting the type property.
Test passed

Testing setting/getting a unit-specific
property.
Test passed

Testing changing an existing property's
value.
Test passed

Testing getting a non-existent property's
value.
Test passed
```

Потрясающе! Да, вы определенно знаете, что делаете. Я вышлю чек, а вы продолжайте работать над классом Unit. Юниты по доске еще не перемещаются?



Заказчик, который видит работающий код, обычно находится в хорошем настроении и продолжает платить. Заказчик, который видит только диаграммы, теряет терпение и начинает раздражаться: на поддержку и щедрую оплату рассчитывать не стоит.

А вот у меня не все идеально, мне не нравится, что ваша библиотека возвращает null и моим парням приходится проверять полученное значение. Мы пишем свой код правильно, поэтому при запросе несуществующего свойства просто инициируйте исключение, договорились?



Знакомьтесь: это Сью. Она руководит группой опытных разработчиков, заинтересованных в использовании игровой библиотеки Гэри.

Давайте изменим контракт на разработку игровой системы

Занимаясь созданием программного продукта, вы также «заключаете» контракт между продуктом и теми людьми, которые будут его использовать. Контракт подробно определяет поведение программы при выполнении некоторых действий, например обращении к несуществующему свойству юнита.

Если заказчик хочет, чтобы реакция на действие была другой, контракт придется изменить. Таким образом, требование о том, что игровая библиотека должна выдавать исключение при обращении к несуществующему свойству, означает всего лишь изменение контракта между разработчиками игр и библиотекой.

При контрактном программировании вы и пользователи вашей программы заключаете соглашение о том, что программа должна работать определенным образом.

Хотите узнать об этом больше? Переверните страницу...

До настоящего момента мы занимались контрактным программированием

Возможно, вы не заметили, но до настоящего момента при реализации класса **Unit** мы занимались *контрактным программированием*. Если пользователь класса **Unit** запрашивает несуществующее свойство, мы просто возвращаем null. То же самое происходит в **getWeapons()**; если список **weapons** еще не инициализирован, мы возвращаем null:

Принимаемые решения влияют на класс Unit и на правила обработки обращений к несуществующим свойствам, так что мы все еще работаем с первой частью функциональности Unit.

Может оказаться, что список еще не инициализирован. Если юнит еще не вооружен, эта команда вернет null.

```
public List getWeapons() {
  return weapons;
}

// другие методы

public Object getProperty(String property) {
  if (properties == null) {
    return null;
  }
  return properties.get(property);
}
```

При отсутствии свойств возвращается null...

...и при отсутствии значения запрашиваемого свойства эта команда вернет null.

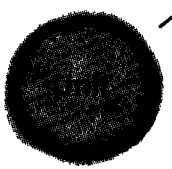
Хотя вы и не знали об этом, данный код определяет контракт, описывающий, что происходит при обращении к несуществующим свойствам.

Unit.java

Это контракт на класс Unit

Класс **Unit** предполагает, что использующие его люди являются квалифицированными программистами и умеют обрабатывать возвращаемые значения null. Таким образом, наш контракт означает следующее:

Ты выглядишь довольно сообразительным. При обращениях к несуществующим свойствам или запросу оружия, которое еще не было определено, я буду возвращать null. Ты ведь умеешь обрабатывать null, верно?

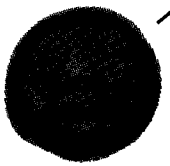


Это наш контракт... Он указывает, что делает программа в определенной ситуации.

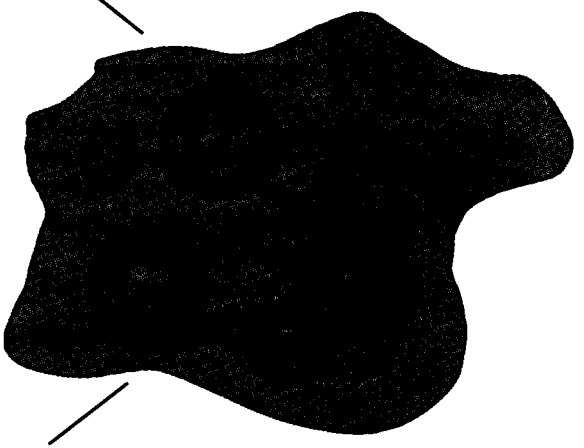
Контрактное программирование основано на доверии

Возвращая null, вы **доверяете** программистам и их умению обрабатывать возвращаемые значения null. По сути, предполагается, что программисты достаточно хорошо владеют своим ремеслом, чтобы не запрашивать несуществующие свойства или оружие, поэтому их код может не беспокоиться о возможном получении null от класса `Unit`:

Ты выглядишь довольно сообразительным. При обращениях к несуществующим свойствам или запросу оружия, которое еще не было определено, я буду возвращать null. Ты ведь умеешь обрабатывать null, верно?



Послушай, мы знаем, что делаем. Наш код будет обращаться только к существующим свойствам. Так что возвращай null... а мы сами сделаем что надо.

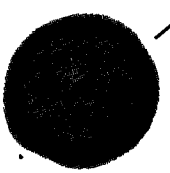


И при необходимости контракт всегда можно изменить...

На с. 481 нас попросили отказаться от возвращения null, вместо этого программа должна инициировать исключение. В действительности изменение контракта не столь уж значительно; оно означает лишь то, что теперь у разработчиков игр при обращении к несуществующим свойствам или оружию возникнут более серьезные проблемы.

Пожалуй, мы твердо уверены в том, что не будем обращаться к несуществующим свойствам. А если это все же произойдет, просто выдай исключение — программа аварийно завершится, и мы начнем искать ошибку. Доверься нам... Исключение не создаст проблем.

Договорились. Если ты знаешь, что я буду выдавать исключение, меня это устраивает. Я немного изменю свой код, и мы начнем использовать новый контракт.



Но если вы не доверяете пользователям...

А если вы не уверены в том, что ваш код будет использоваться правильно? Или вы считаете, что некоторые действия настолько опасны, что их обработку просто нельзя доверить пользователям? В таких случаях стоит использовать **защитное программирование**.

Допустим, вас сильно беспокоят опасения, что при использовании класса Unit разработчики будут обращаться к несуществующим свойствам и получать значения null без их корректной обработки. Перепишите метод **getProperty()** в следующем виде:

```
public Object getProperty(String property)
  throws IllegalAccessException {

  if (properties == null) {
    return null;
    throw new IllegalAccessException(
      "What are you doing? No properties!");
  }
  return properties.get(property);
  Object value = properties.get(property);
  if (value == null) {
    throw new IllegalAccessException(
      "You're screwing up! No property value.");
  } else {
    return value;
  }
}
```

Null больше не возвращается... Мы СЕРЬЕЗНО относимся к запросам несуществующих свойств.

Эта версия getProperty() может инициировать ПРОВЕРЯЕМОЕ исключение, которое должно быть обработано кодом, использующим Unit.

Защитная версия Unit.java.

Unit.java

Я понимаю, что ты пишешь отличный код, и все такое, но я тебе не доверяю. Я могу передать null, и твой код попросту рухнет. Так что давай для надежности я отправлю проверяемое исключение, которое тебе придется перехватить. Так я буду уверен в том, что ты не получишь значение null, с которым произойдет что-нибудь плохое.

Защищенное программирование всегда предполагает худшее и старается защитить себя (и вас) от порчи или некорректного использования данных.

...или они не доверяют вам...

Конечно, программисты, которые используют ваш код, тоже могут не доверять вам... И они тоже могут применить защитное программирование. Допустим, они сомневаются, что **getProperty()** возвращает только отличные от null значения? Тогда они тоже снижают риск для своего кода посредством защитного программирования:

Пример кода, использующего класс Unit.

```
// Метод создает экземпляр Unit
Unit unit = getUnit();

// Теперь использует созданный экземпляр...
String name = unit.getName();
if ((name != null) && (name.length() > 0)) {
  System.out.println("Unit name: " + name);
}
Object value = unit.getProperty("hitPoints");
if (value != null) {
  try {
    Integer hitPoints = (Integer)value;
  } catch (ClassCastException e) {
    // Обработка возможной ошибки
  }
}
// И т.д...
```

Этот код выполняет ОСНОВАТЕЛЬНУЮ проверку ошибок... Он никогда не полагается на то, что Unit вернет допустимые данные.

Данный код чрезвычайно защищен

Часто задаваемые вопросы

В: На с. 483 вы сказали, что контракт можно изменить для выдачи исключения, а теперь говорите, что генерация исключения — это защитное программирование...

О: Неважно, какое исключение будет выдаваться. Важно то, как ваши заказчики и пользователи приняли это решение. При контрактном программировании вы работаете с клиентом и согласовываете с ним способ решения проблемы; в защитном программировании вы принимаете решение таким образом, чтобы полностью исключить аварийные ситуации в вашем коде, независимо от того, чего хочет клиент.

Когда мы решили переключиться с возвращения null на выдачу исключения, мы побеседовали с клиентом и договорились о том, что при запросе несуществующих свойств должно выполняться именно это действие. И мы использовали исключение RuntimeException, потому что клиент не хотел включать в свой код многочисленные блоки try/catch. С таким же успехом он мог попросить реализовать проверяемое исключение, мы бы согласились, и все равно это было бы контрактным программированием.

С другой стороны, при защитном программировании мнение клиента нас вообще не интересует. Мы должны гарантировать, что наш код не станет причиной сбоя в программе, и по возможности лишить клиента возможности создания аварийной ситуации в его коде.

Беседа у камина

Сегодня в программе: **Контрактное программирование и Защитное программирование обсуждают проблемы доверия**

Контрактное программирование

Приятно вот так посидеть, пообщаться лицом к лицу... Ведь мы нечасто встречаемся.

Что ты имеешь в виду?

Ну да, конечно... Но нельзя жить, постоянно опасаясь плохого кода. В какой-то момент нужно просто взять на себя обязательства и поверить в то, что другие программисты будут правильно использовать тебя.

Кажется, у кого-то проблемы с доверием.

Нет. Я верю в то, что программисты понимают тот контракт, который я им предоставляю.

Защитное программирование

Да, я редко куда-нибудь выбираюсь. Вокруг столько проблем...

Ну, я могу переходить улицу и поскользнуться на объекте Banana, который не был уничтожен уборщиком мусора, или на перекресток может выскочить на бешеной скорости цикл for с неправильным условием завершения.... Вокруг так много плохого кода.

Шутишь? Ты когда-нибудь говорил с программистами, которым предлагаешь доверять? Да они тратят больше времени на просмотр «Остаться в живых», чем на поиски ошибок в своем коде. Если бы они перестали пялиться на Кейт с ее веснушками, возможно, мне бы не пришлось постоянно перепроверять их работу.

А у тебя их нет?

Контракт? Ты все еще полагаешь, что если все подробно объяснить, «хорошие» программисты будут правильно использовать тебя? Какая наивность!

Контрактное программирование

В моем контракте четко определено, что должен делать я, а что должны делать люди, которые используют мой код.

Послушай, если программисты и пользователи не соблюдают свою часть сделки, я за это не отвечаю. Если они нарушают контракт, то вся вина лежит на них. Я здесь ни при чем..

А цена? Тонна лишнего кода с постоянными проверками if (value == null)? Мне кажется, что ты сам замедляешь работу программ.

Супермена? Надеюсь, мне послышалось...

Но в этом-то все и дело! Может, я плохо подхожу ленивым программистам, но зато идеален для профессионалов, которые думают о своей работе. Они меня используют, и код становится эффективным и компактным.

И именно это я предоставляю, без всех проблем с доверием и балласта, который ты тащишь с собой....

Получай безопасный код, ах ты...

Защитное программирование

А ты не в курсе? В наши дни свыше 50% контрактов заканчиваются разводом... Или это о другом? Не знаю... Ты действительно думаешь, что программисты обратят внимание на твой жалкий контракт?

По-моему, это жестоко. Я стараюсь помогать своим пользователям и даже по возможности защищаю их от самих себя.

Конечно, не всегда я работаю с максимальной эффективностью, но, безусловно, защищаю пользователей от катастроф. Иногда я кажусь себе чем-то вроде... Супермена.

Нет, правда! И потом, сколько раз твой код горел синим пламенем из-за ленивого программиста, который не обратил внимания на контракт?

Компактный код — пустяки. Я предпочитаю хороший код. Код, обеспечивающий безопасность программистов и пользователей..

Я тебе покажу проблемы с доверием...

...передача прервана по техническим причинам... 4 8 15 ...

Кто я?

Функционально-ориентированная разработка, **Сценарно-ориентированная разработка**, **Контрактное программирование** и **Защитное программирование** явились на маскарад, но не надели таблички с именами. Послушайте, что они говорят, и определите, кто скрывается за той или иной маской. Будьте внимательны... Некоторые высказывания могут принадлежать сразу нескольким персонажам в масках.

Я предельно упорядочен. Я делаю все последовательно, по одному шагу, пока не сделаю от начала до конца.

_____

Ну конечно, она обещала, но можно ли сегодня кому-нибудь доверять?

_____

О, требования вдохновляют меня на свершения.

_____

Я разбираюсь со всем своим поведением, прежде чем переходить к чему-то еще.

_____

Всё ради заказчика. В конце концов, я всего лишь стараюсь выполнить его пожелания.

_____

Послушай, ты уже большой и можешь разобраться с этим сам... Теперь это не моя проблема, договорились?

_____

Если тебя это устраивает, то и я не против. С какой стати я буду говорить тебе, что делать, если ты знаешь, чего можно ожидать от меня?

_____

→ Ответы на с. 499.

Возьми в руку карандаш

Измените программный контракт Unit.

Клиенты Гэри хотят, чтобы класс Unit был рассчитан на правильное использование. Таким образом, обращение к несуществующему свойству свидетельствует о каких-то фатальных проблемах в игре, поэтому программа должна инициировать исключение. Ваша задача:

1. Обновите Unit.java так, чтобы обращения к несуществующим свойствам приводили к выдаче исключения. Выберите тип исключения на свое усмотрение.
2. Обновите UnitTester и тестовые примеры, на которые влияет изменение контракта.
3. Снова запустите UnitTester и убедитесь в том, что Unit.java проходит все тесты.

Наши ответы приведены на следующей странице.

Возьми в руку карандаш
Решение

Измените программный контракт Unit

Клиенты Гэри хотят, чтобы класс Unit был рассчитан на правильное использование. Таким образом, обращение к несуществующему свойству свидетельствует о каких-то фатальных проблемах в игре, поэтому программа должна инициировать исключение.

В Unit.java достаточно изменить всего один метод getProperty().

Обращение к свойству при переменной properties, равной null, так и остается обращением к несуществующему свойству.

```
public Object getProperty(String property) {
  if (properties == null) {
    ~~return null;~~
    throw new RuntimeException(
      "No properties for this Unit.");
  }
  ~~return properties.get(property);~~
  Object value = properties.get(property);
  if (value == null) {
    throw new RuntimeException(
      "Request for non-existent property.");
  } else {
    return value;
  }
}
```

При любых обращениях к несуществующим свойствам инициируется исключение RuntimeException.

Код, использующий Unit, не обязан перехватывать RuntimeException... Таким образом, мы по-прежнему используем контрактное программирование, а не защитное.

Сейчас мы не можем просто вернуть значение из Map. Нужно убедиться в том, что свойство существует и отлично от null.

Unit.java

Часто задаваемые вопросы

В: Почему мы инициируем RuntimeException, а не проверяемое исключение, например IllegalAccessException?

О: При использовании проверяемых исключений код, вызывающий `getProperty()`, должен будет проверять это исключение в блоках `try/catch`. Это не соответствует пожеланиям заказчика. Мы согласились на контракт, который разрешает пользователю программировать без перехвата исключений. С `RuntimeException` от клиентского кода не потребуется никакой дополнительной работы для использования класса `Unit`.

В: Как насчет get-методов для других свойств, таких как weapons, name и id?

О: `id` и `name` хранятся в `int` и `String` соответственно, так что с ними проблем не будет (значение `id` необходимо для создания нового экземпляра `Unit`, а `name` содержит либо null, либо строку значения). Свойство `weapons` представляет собой контейнер `List`, так что при вызове `getWeapons()` для пустого списка будет возвращаться значение null. Метод можно изменить, чтобы при пустом списке вооружений он инициировал исключение, хотя наши клиенты этого явно не просили.

При контрактном программировании проблемные ситуации совместно разрешаются вашим и клиентским кодом.

При защитном программировании вы следите за тем, чтобы клиент получал «безопасный» ответ независимо от желания самого клиента.

test4() тестирует обращения к несуществующим свойствам.

Мы ожидаем, что при обращении к несуществующему значению будет инициироваться исключение RuntimeException, поэтому мы перехватываем его...

...и используем как признак успешного прохождения теста.

```
public void test4(Unit unit, String propertyName) {
  System.out.println(
    "\nTesting getting a non-existent property's value.");
  try {
    Object outputValue = unit.getProperty(propertyName);
  } catch (RuntimeException e) {
    System.out.println("Test passed");
    return;
  }
  System.out.println("Test failed.");
}
```

UnitTester.java

Не забудьте, что после инициирования исключения и выдачи сообщения о прохождении теста следует выйти из тестового метода.

Если исключение не инициируется, значит, возникли проблемы... тест не прошел.

В: Вы сказали, что контрактное программирование приводит к более компактному коду, но, похоже, в нашем случае в Unit.java появляется много нового кода.

О: Дело в том, что вместо прямого возвращения null мы инициируем исключение `RuntimeException`. Скорее это особый, а не типичный случай. Обычно дело ограничивается простым возвращением объектов и значений без проверки их на null или принадлежность к определенному диапазону данных.

В: Я все равно не понимаю, зачем мы переключаемся на контрактное программирование. Чем оно лучше?

О: Вопрос не в том, лучше оно или хуже, а в том, чего хочет заказчик. Собственно, вам вряд ли придется часто решать, какое программирование вы будете использовать — контрактное или защитное. Обычно выбор определяется пожеланиями заказчика и квалификацией пользователей, которые будут работать с написанным вами кодом.

Перемещение юнитов

Свойства юнитов | Группы юнитов

Путь оказался довольно долгим, но мы наконец-то подошли к следующему блоку функциональности класса Unit.

Мы наконец-то завершили работу над реализацией свойств юнитов и теперь можем перейти к следующему пункту списка.

1. **Каждый юнит должен обладать свойствами, причем разработчики игры должны иметь возможность включать новые свойства в типы юнитов в их играх.**

Переходим к перемещению.

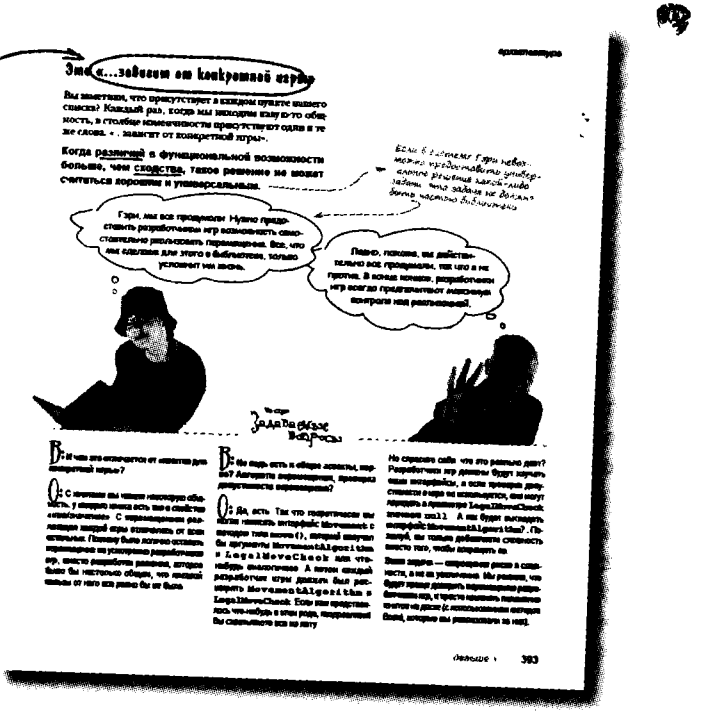
2. **Юниты должны уметь перемещаться с одного поля доски на другое.**

3. **Юниты должны уметь объединяться в армии.**

Кажется, что-то такое уже было?

Выглядит вроде бы знакомо... Вспомните главу 7:

В главе 7 мы решили, что перемещения будут по-разному реализовываться в разных играх.


Это «...зависит от конкретной игры»

Вы заметили, что присутствует в каждом пункте нашего списка? Каждый раз, когда мы находим какую-то общность, в столбце комментарии присутствуют одни и те же слова. «...зависит от конкретной игры».

Когда различий в функциональной возможности больше, чем сходства, такое решение не может считаться хорошим и универсальным.

Если бы в системе Гэри невозможно предложить универсальное решение какой-либо задачи, эта задача не должна быть частью библиотеки

Гэри, мы все прекрасно! Нужно предоставить разработчикам игр возможность самостоятельно реализовать перемещения. Все, что мы сделаем для этого в библиотеке, только усложнит им жизнь.

Ладно, похоже, мы действительно все преувеличили, так что я не против. В конце концов, разработчики игр всегда предпочитают максимум контроля над реализацией.

Задайте вашие вопросы

В: И что это отличается от объектов для конкретной игры?

О: С точками мы имеем некоторую общность, у каждого юнита есть три свойства координаты. С перемещениями реализация каждой игры отличается от всех остальных. Поэтому было логично оставить перемещения на усмотрение разработчиков игр, вместо разработки решения, которое было бы настолько общим, что никакой толк от него вся равно бы не был.

В: Но ведь есть и общие элементы, например? Допустим перемещения, проверка допустимости перемещения?

О: Да, есть. Так что теоретически мы могли ввести интерфейс IMovement с методом типа move(), который получал бы аргументы IMovementAlgorithm и LogicMoveCheck или что-нибудь аналогичное. А потом каждый разработчик игры должен был реализовать IMovementAlgorithm и LogicMoveCheck. Если мы представили что-нибудь в этом роде, получилось бы слишком всё на лету.

Но спросим себя: что это реально дает? Разработчики игр должны будут изучить наши интерфейсы, а если проверки допустимости в игре не используются, они могут передать в проверяющие LogicMoveCheck значения null. А как будет выглядеть интерфейс IMovementAlgorithm? Подумай, мы только добавляем сложность вместо того, чтобы упрощать ее.

Наша задача — упрощение решения в сложности, а не ее увеличение. Мы решили, что будет проще доверить перемещения разработчикам игр, и просто написать небольшие комментарии к классам (с использованием интерфейсов), которые мы рекомендуем за них.

дальше ▸ 383

Разбивайте свои приложения на меньшие функциональные блоки

Мы много говорили об углублении итераций и выполнении на каждой стадии более подробного анализа и проектирования. Иначе говоря, мы берем каждую задачу и разбиваем ее (по вариантам использования или функциональным возможностям), решаем часть задачи... и так снова и снова. Именно так мы действовали в этой главе — брали одну функциональную возможность и работали над ней, пока она не будет полностью реализована.

Задачей здесь является игровая библиотека Гэри.

Задача

Функц. возможность

Функц. возможность

Функц. возможность

Но разбиение можно продолжить...

После того как выбраны функциональная возможность или вариант использования, их обычно удается разбить на более мелкие аспекты поведения. Например, юниты должны обладать свойствами и поддерживать перемещение, а еще необходимо обеспечить поддержку группировки юнитов. Таким образом, нам предстоит последовательно работать с каждым из функциональных блоков.

Мы выбираем одну функциональную возможность и работаем над ней: в данном случае это игровые юниты.

Юниты

Свойства

Перемещение

Группировка

При разбиении приложения вам придется выполнять новый анализ и проектирование на каждом шаге. Принятые ранее решения должны оставаться обоснованными, в противном случае их следует изменить или переработать.

К решениям также может применяться итеративный метод

Вы часто будете обнаруживать, что принятые ранее решения избавляют вас от лишней работы. В системе Гэри мы решили, что разработчики игр будут реализовывать перемещения самостоятельно. Теперь, когда мы добрались до перемещения юнитов, можно взять принятое ранее решение и применить его. Так как решение выглядит вполне разумно и изменять его незачем, мы предоставим разработчикам игр заниматься перемещениями, а сами перейдем к следующему аспекту поведения.

Решения, принимаемые относительно задачи в целом...

...часто также применимы к уровням функциональных возможностей и поведения.

Просто включите в документацию для разработчиков игр упоминание о том, что перемещения они должны реализовать сами.

Юниты должны уметь перемещаться с одного поля доски на другое.

Мы можем закрыть следующие функциональности в классе Unit.

Головоломка

Осталось реализовать эту часть, и функциональность Unit будет завершена.

К настоящему моменту вы уже достаточно хорошо разбираетесь в функционально-ориентированной разработке, итеративном методе, анализе и проектировании. Мы оставим вам реализацию последнего блока функциональности, завершающую работу над классом Unit в игровой библиотеке Гэри.

Проблема:

Библиотека должна поддерживать группировку юнитов.

Ваша задача:

1. Создайте новый класс, который может группировать юниты, включать и удалять юниты из группы.
2. Заполните таблицу тестовыми примерами, которые проверят правильность работы кода и докажут Гэри, что группировка юнитов действительно функционирует.
3. Включите в **`UnitTester`** методы для реализации тестовых примеров из таблицы. Убедитесь в том, что все тесты проходят успешно.

Возможно, некоторые строки останутся неиспользованными... А может, и наоборот — строк не хватит.

ID	Что тестируем	Входные данные	Ожидаемый результат	Исходное состояние

Наше решение приведено на с. 496.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Чтобы написать хороший программный продукт, сначала нужно убедиться, что ваше приложение соответствует пожеланиям заказчика.
- Заказчиков не интересуют диаграммы и списки, они хотят видеть программу, которая что-то делает.
- При сценарно-ориентированной разработке последовательно прорабатываются сценарии варианта использования приложения.
- При сценарно-ориентированной разработке вы обычно работаете над отдельными сценариями, но при этом реализуете все сценарии одного варианта использования, прежде чем переходить к другим вариантам использования.
- При функционально-ориентированной разработке вы полностью реализуете отдельную функциональную возможность, прежде чем двигаться дальше.
- При функционально-ориентированной разработке можно работать как над большими, так и над мелкими функциональными возможностями, но важно работать над ними по отдельности.
- Разработка всегда выполняется итерационно. Разработчик анализирует общую картину, а затем переходит к деталям.
- Анализ и проектирование должны выполняться на каждом шаге цикла разработки, в том числе и при переходе к новой функциональной возможности или варианту использования.
- Тесты помогают убедиться в отсутствии ошибок и демонстрируют заказчику, что его программа действительно работает.
- Хороший тестовый пример работает с одним блоком функциональности.
- В тестовом примере может быть задействован один или несколько методов одного класса, а могут быть задействованы несколько классов.
- При разработке через тестирование вы сначала пишете тесты, а затем разрабатываете программы, которые проходят эти тесты. В результате появляется полностью функционирующая, работоспособная программа.
- Контрактное программирование предполагает, что обе стороны понимают, какое поведение генерируется по тем или иным действиям, и соблюдают этот контракт.
- При возникновении ошибок в условиях контрактного программирования методы обычно возвращают null или непроверяемые исключения.
- Защитное программирование выявляет потенциальные проблемы и осуществляет многочисленные проверки для предотвращения проблемных ситуаций.
- В условиях защитного программирования методы при некорректных действиях обычно возвращают «пустые» объекты или инициируют проверяемые исключения.

Головоломка. Решение

Библиотека Гэри должна поддерживать объединение юнитов в группы, а также группировку полученных групп (на столько уровней, на сколько захочет разработчик).

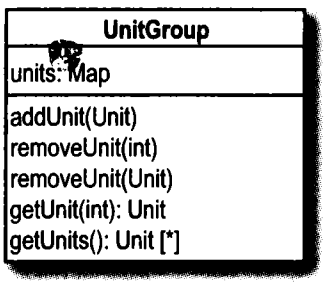
Мы решили использовать контейнер Map, в котором ключом является идентификатор юнита, а значением — объект Unit.

Новый экземпляр UnitGroup создается на основе контейнера List с юнитами, которые изначально включаются в группу.

```
public class UnitGroup {
  private Map units;

  public UnitGroup(List unitList) {
    units = new HashMap();
    for (Iterator i = unitList.iterator(); i.hasNext(); ) {
      Unit unit = (Unit)i.next();
      units.put(unit.getId(), unit);
    }
  }

  public UnitGroup() {
    this(new LinkedList());
  }

  public void addUnit(Unit unit) {
    units.put(unit.getId(), unit);
  }

  public void removeUnit(int id) {
    units.remove(id);
  }

  public void removeUnit(Unit unit) {
    removeUnit(unit.getId());
  }

  public Unit getUnit(int id) {
    return (Unit)units.get(id);
  }

  public List getUnits() {
    List unitList = new LinkedList();
    for (Iterator i = units.entrySet().iterator(); i.hasNext(); )
{
      Unit unit = (Unit)i.next();
      unitList.add(unit);
    }
    return unitList;
  }
}
```

Конструктор просто добавляет все юниты в контейнер, используя в качестве ключа каждого элемента идентификатор юнита.

Диаграмма классов UnitGroup дает представление о нашей реализации.



Благодаря использованию контейнера Map мы можем получать и удалять юниты по идентификатору, что довольно удобно.

Для получения списка всех юнитов, хранящихся в Map, придется немного потрудиться, но мы решили, что удобство обращения к юнитам по идентификатору важнее.

А теперь к тестовым примерам:

Мы увеличили идентификаторы, чтобы они не конфликтовали с идентификаторами уже написанных тестовых примеров.

ID	Что тестируем	Входные данные	Ожидаемый результат	Исходное состояние
10	Создание UnitGroup по списку юнитов	List с юнитами	Тот же список	Без существующего экземпляра UnitGroup
11	Включение юнита в группу	Unit с ID = 100	Unit с ID = 100	UnitGroup без элементов
12	Обращение к юниту по идентификатору	100	Unit с ID = 100	UnitGroup без элементов
13	Получение всех юнитов в группе	N/A	Список юнитов, совпадающий с исходным	UnitGroup с известным списком юнитов
14	Удаление юнита по идентификатору	100	Список юнитов (не содержащий юнит с ID = 100)	UnitGroup без элементов
15	Удаление юнита по экземпляру Unit	Unit с ID = 100	Список юнитов (не содержащий юнит с ID = 100)	UnitGroup без элементов

Тестовые примеры для UnitGroup, которые придумали мы. А что получилось у вас?

Мы начинаем с пустого экземпляра UnitGroup, чтобы юниты, с которыми мы работаем, не отражались на результатах тестов.

В сети

Вы и сами сможете написать код UnitTester по приведенной таблице тестовых примеров. Если вы хотите сравнить свой код с нашим, посетите сайт http://www.headfirstlabs.com, откройте раздел этой книги и найдите ссылку «UnitGroup Test Cases».

Инструментарий ООАП

В этой главе мы рассмотрели несколько методов итеративной разработки проектов и две распространенные методологии программирования. Добавьте их в ваш инструментарий:

Требования

Хорошие т
гают убеди
ваша систе
как хочет

Требования
страня
антов

Из вари
можно
казчик

Анализ
ния мож
или от

Анализ и проектирование

Хорошо спроектированный
легко изменяется и расшир
ется.

Решение больших задач

Слушайте заказчика и пойми-
те, какая система ему нужна.
Постройте список функци-
ональных возможностей на
языке, понятном заказчику.

р, что функциональ-

ОО-принципы

Инкапсулируйте то, что изменя

Программируйте для интерфейс
реализации.

Каждый класс приложения долже
только одну причину для изменен

Методологии программирования

Контрактное программирование определяет правила поведения кода, которые должны соблюдаться и вами и пользователями.

Защитное программирование не доверяет внешнему коду и осуществляет обширную проверку ошибок и данных, чтобы предотвратить возможную передачу некорректной или ненадежной информации.

Методы разработки

При сценарно-ориентированной разработке вы берете один вариант использования и пишете код реализации всего варианта использования, вк ная все сценарии. Только после завершения можно переходить к следующему аспекту приложения.

При функционально-ориентированной разработке вы работаете над одной функциональной возможностью и полностью программируете ее поведение. Только после завершения можно переходить к следующему аспекту приложения.

При разработке через тестирование сначала пишутся тесты для функционального блока и только потом программируется сама функциональность.

Хорошо организованный процесс разработки обычно включает <u>все</u> эти модели на разных стадиях цикла разработки.

Кто я? Решение

Функционально-ориентированная разработка, **Сценарно-ориентированная разработка**, **Контрактное программирование** и **Защитное программирование** явились на маскарад, но не надели таблички с именами. Послушайте, что они говорят, и определите, кто скрывается за той или иной маской. Будьте внимательны... Некоторые высказывания могут принадлежать сразу нескольким персонажам в масках.

Я предельно упорядочен. Я делаю все последовательно, по одному шагу, пока не сделаю от начала до конца.

Сценарно-ориентированная разработка

Ну конечно, она обещала, но можно ли сегодня кому-нибудь доверять?

Защитное программирование

О, требования вдохновляют меня на свершения.

Функционально-ориентированная разработка, сценарно-ориентированная разработка

Здесь также можно было бы добавить «контрактное программирование», так как контракт в действительности является разновидностью требований.

Я разбираюсь со всем своим поведением, прежде чем переходить к чему-то еще.

Функционально-ориентированная разработка

Всё ради заказчика. В конце концов, я всего лишь стараюсь выполнить его пожелания.

Всё!

Все эти методы и методологии в действительности преследуют одну цель: предоставить заказчику ту программу, которая ему нужна.

Послушай, ты уже большой. Ты можешь разобраться с этим сам... Теперь это не моя проблема, договорились?

Контрактное программирование

Если тебя это устраивает, то и я не против. С какой стати я буду говорить тебе, что делать, если ты знаешь, чего можно ожидать от меня?

Контрактное программирование

Все вместе

Далеко ли до цели? Мы изучили множество способов совершенствования программных продуктов, но пришло время собрать все воедино. Да, это именно тот момент, которого вы так долго ждали: мы возьмем всё, о чем вы узнали из книги, и покажем, что в действительности это были лишь части общего процесса, который можно использовать вновь и вновь для написания хороших программ.

Разработка в стиле ООАП

К этому моменту у вас появилось множество новых инструментов, приемов и идей относительно того, как разрабатывать хорошие программы... Но мы все еще не собрали все новые знания воедино.

Именно об этом написана эта глава — мы возьмем все отдельные навыки и приемы, такие как определение требований, написание вариантов использования и применение паттернов, и преобразуем их в универсальный процесс, который позволит вам стабильно добиваться успеха даже с самыми хитроумными задачами.

Итак, как же выглядит этот процесс?

Список функциональных возможностей

Определите, что должно делать ваше приложение, на высоком уровне.

Диаграммы вариантов использования

Выделите масштабные процессы, выполняемые приложением, и все задействованные внешние факторы.

Разбиение задачи

Разбейте приложение на функциональные модули и решите, в каком порядке вы будете их рассматривать.

Требования

Сформулируйте отдельные требования для каждого модуля и убедитесь в том, что они соответствуют общей картине.

1. Убедитесь в том, что поведение программы соответствует требованиям заказчика.

Вам не кажется, что ОЧЕНЬ много времени тратится на функциональность? И правильно... Если заказчик не будет доволен программным продуктом, ваша работа не может быть успешной.

Эта часть применяется и ко всем дробным задачам. Вы используете эти фазы при решении задачи, применяете итеративный переход к следующей задаче и снова используете их.

Итеративная разработка

Анализ предметной области

Определите соответствия между вариантами использования и объектами приложения. Убедитесь в том, что представления заказчика совпадают с вашими.

Предварительное проектирование

Заполните подробную информацию об объектах, определите отношения между объектами, примените принципы и паттерны.

Реализация

Напишите код, протестируйте его и убедитесь в том, что он работает. Проделайте это для каждого аспекта поведения, каждой функциональной возможности, каждого варианта использования и каждой задачи, пока реализация приложения не будет завершена.

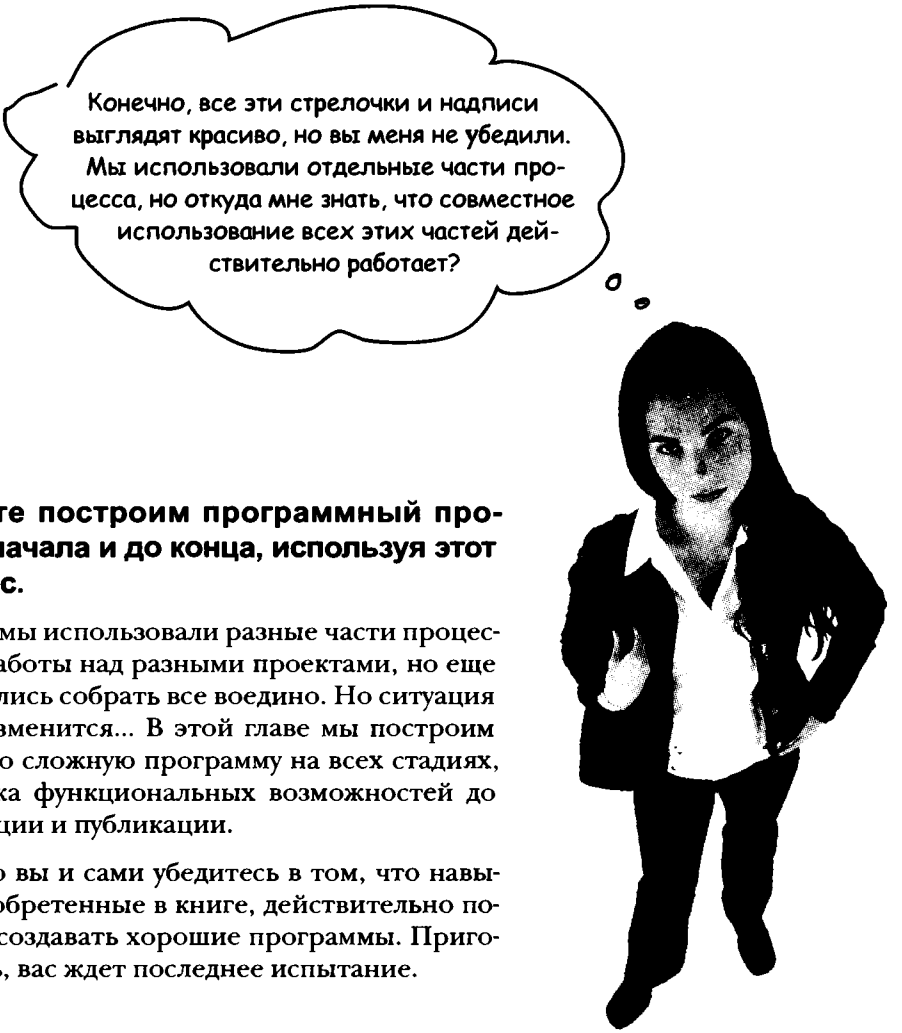
Публикация

Готово! Выпустите в свет свой продукт, представьте счета к оплате и получите деньги.

2. Применяйте базовые ОО-принципы для повышения гибкости.

3. Постарайтесь создать структуру кода, упрощающую сопровождение и повторное использование.

Эти части процесса только выглядят маленькими, большая часть времени разработки проводится в фазах проектирования и реализации.

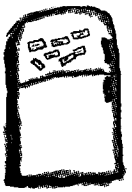


Давайте построим программный проект от начала и до конца, используя этот процесс.

В книге мы использовали разные части процесса для работы над разными проектами, но еще не пытались собрать все воедино. Но ситуация скоро изменится... В этой главе мы построим довольно сложную программу на всех стадиях, от списка функциональных возможностей до реализации и публикации.

Попутно вы и сами убедитесь в том, что навыки, приобретенные в книге, действительно помогают создавать хорошие программы. Приготовьтесь, вас ждет последнее испытание.

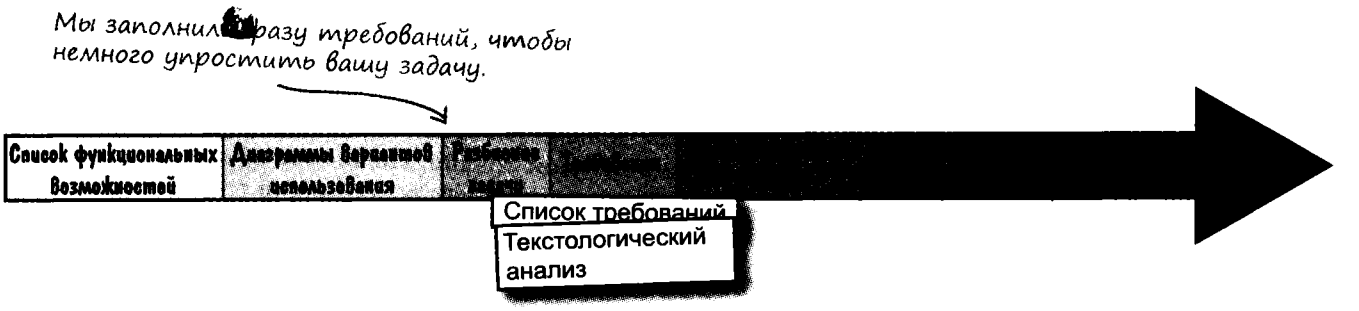
ПРЕДУПРЕЖДЕНИЕ: Ответы на это упражнение приводятся НЕ на следующей странице. Продолжайте читать и выполнять задания. Все ответы вы найдете в конце этой главы.



Развлечения с магнитами

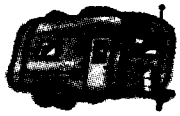
Прежде чем переходить к задачам, которыми мы будем заниматься в этой главе, необходимо понять, какое место занимают изученные нами методы в Большом Жизненном Цикле ООАП-проекта. В нижней части страницы находятся магниты с названиями инструментов, которые мы уже изучили; ваша задача — разместить их в правильных фазах диаграммы жизненного цикла ООАП, также приведенной ниже.

В каждой фазе может находиться более одного магнита, а некоторые магниты могут использоваться многократно. Не торопитесь и хорошенько подумайте!



Задача

Перед вами проект, которым мы будем заниматься в этой главе, от начала и до конца:



Objectville Travel, Inc.
210 Tourist Ave.
Objectville, HF 90210

Постановка задачи

Поздравляем! Узнав о вашей выдающейся работе для магазина Рика и фирмы Дуга, мы решили доверить вам программирование нашей новой системы поиска маршрутов.

Поскольку Объектвиль стал популярным местом для поездок, мы хотим предоставить туристам простые средства, упрощающие посещение городских достопримечательностей. Приложение RouteFinder предназначено для хранения полных описаний линий метро Объектвиля, включая информацию о станциях на каждой линии. Поезда могут перемещаться в обоих направлениях, поэтому хранить информацию о направлении не нужно.

Приложение RouteFinder также должно уметь прокладывать маршруты по указанным начальной и конечной точкам. Наши агенты должны иметь возможность напечатать маршрут между станциями с указанием линий, станций и пересадок.

В Объектвиле всегда уделялось особое значение гибкости и расширяемости. Поэтому мы рассчитываем, что класс RouteFinder будет легко расширяться по мере того, как мы будем предоставлять туристам новые возможности для получения впечатлений в нашем объектно-ориентированном городе.

Надеемся скоро увидеть вашу проектную документацию, а затем и работающее приложение RouteFinder. Мы рассчитываем на вас!

Искренне ваш,

Orbin Traveloctic, исполнительный директор

P. S. Чтобы упростить вашу работу, прилагаем карту метро Объектвиля, а также перечень всех станций и линий.

Эта страница намеренно оставлена пустой, чтобы вы могли вырезать карту Объектвиля и повесить ее на стену.

[от отдела маркетинга: Что вы натворили?!! Почему мы не продаем карту за отдельную плату?]

Карта метро Объектвиля

Условные обозначения

- Станция
- Станция пересадки

Линия Booch

Линия Gamma

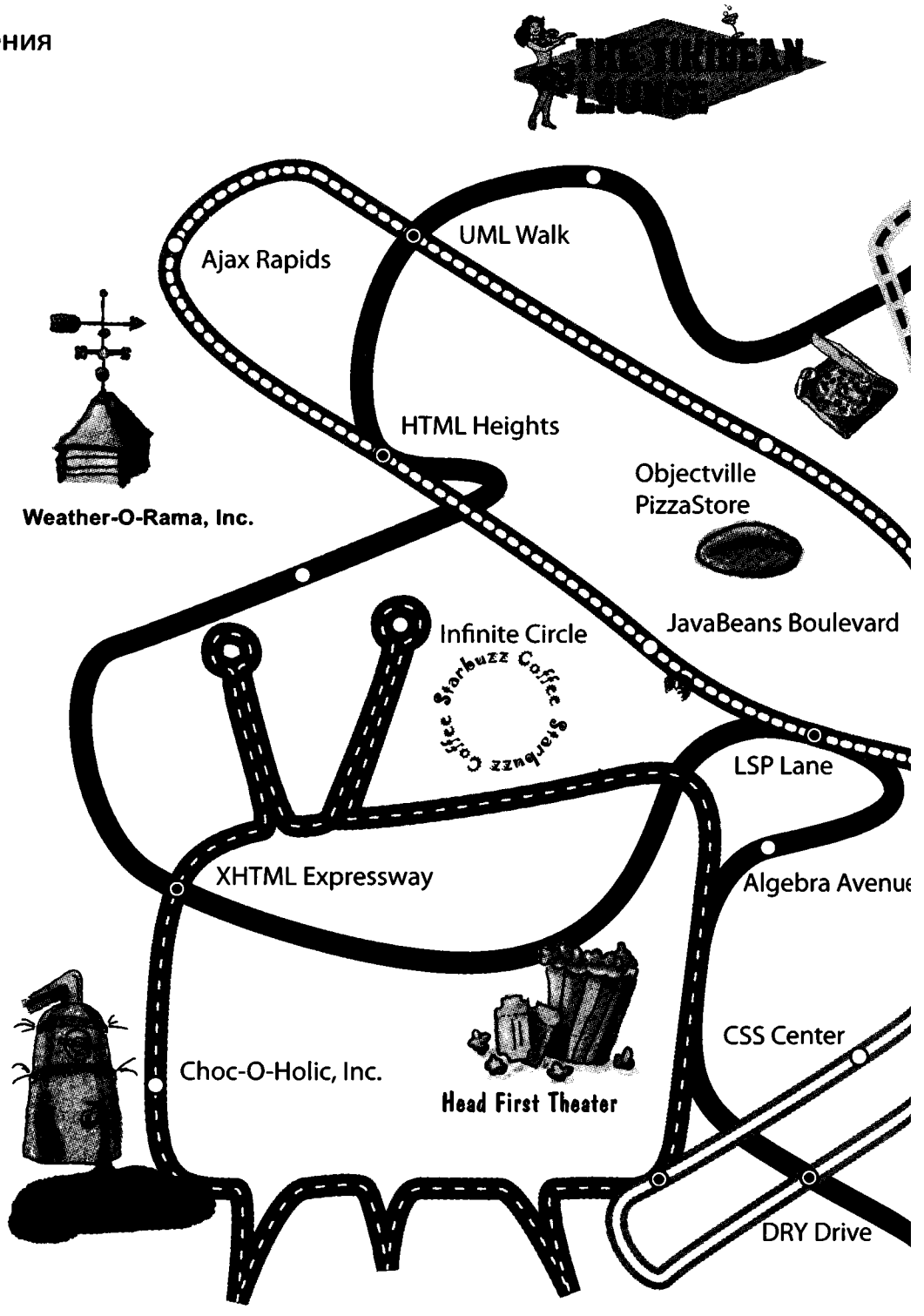
Линия Jacobson

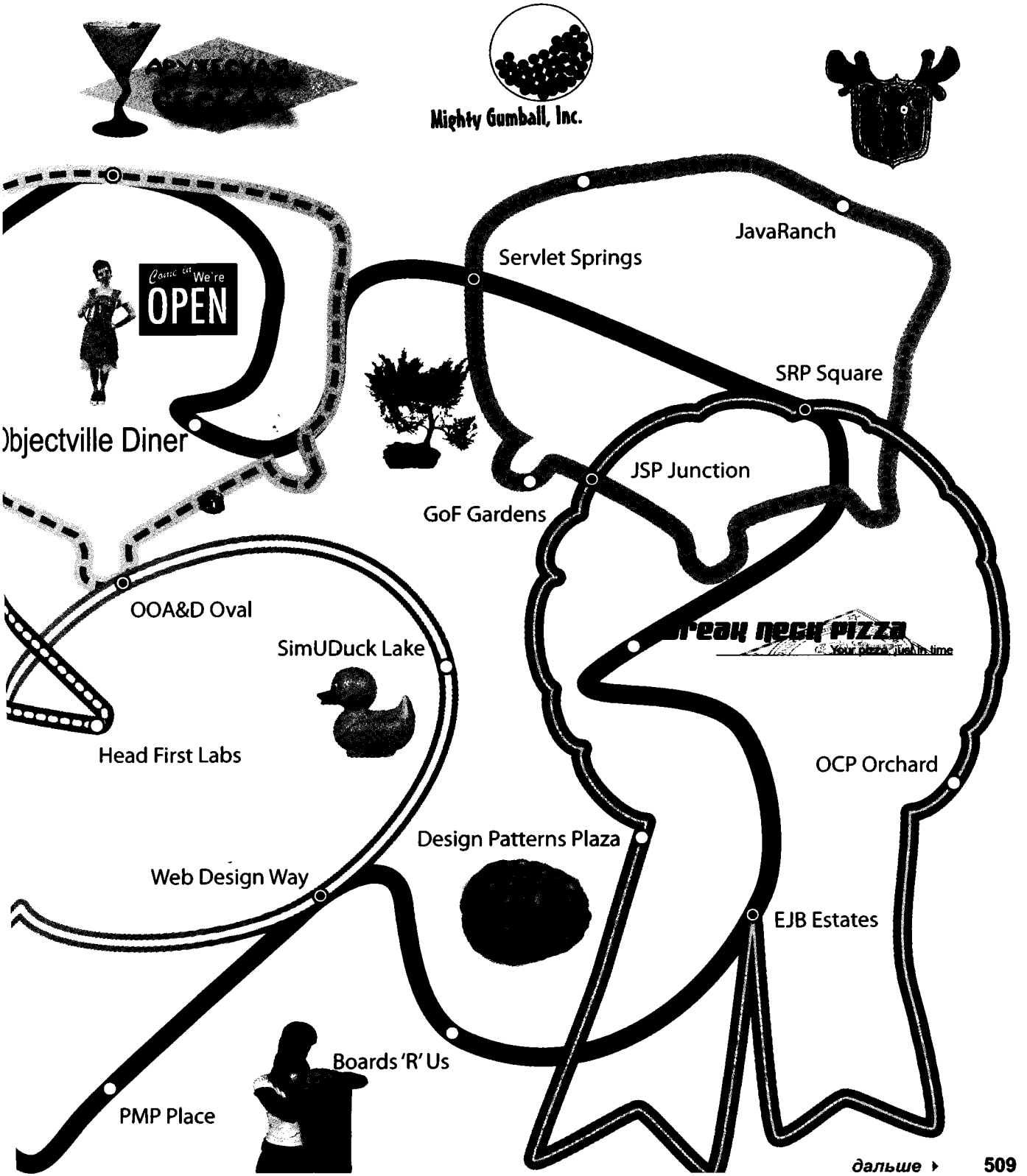
Линия Liskov

Линия Meyer

Линия Rumbaugh

Линия Wirfs-Brock




Mighty Gumball, Inc.
JavaRanch
Servlet Springs
Come in We're OPEN
SRP Square
Objectville Diner
JSP Junction
GoF Gardens
OOA&D Oval
SimUDuck Lake
Head First Labs
OCP Orchard
Design Patterns Plaza
Web Design Way
EJB Estates
Boards 'R' Us
PMP Place

Эта страница намеренно оставлена пустой, чтобы вы могли вырезать карту Объектвиля и повесить ее на стену.

Возьми в руку карандаш

Составьте список функциональных возможностей для приложения RouteFinder.

Обойдемся без долгих вступлений. Возьмите постановку задачи на с. 506 и используйте ее для разработки списка функциональных возможностей. Если вам потребуется вспомнить, что такое функциональные возможности и как выглядит типичный список функциональных возможностей, обращайтесь к главе 6.

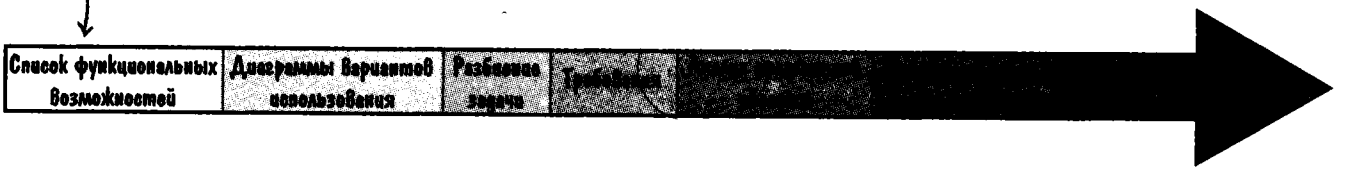
Приложение RouteFinder

Список функциональных возможностей

1. _____
2. _____
3. _____
4. _____
5. _____
6. _____
7. _____

Заполнять все поля не обязательно (если, конечно, вы считаете, что они не нужны).

Вы сейчас находитесь ЗДЕСЬ.



Возьми в руку карандаш
Решение

Составьте список функциональных возможностей для приложения RouteFinder.

Вам было предложено взять постановку задачи на с. 506 и использовать ее для разработки списка функциональных возможностей приложения RouteFinder, которое мы собираемся построить.

Вот что получилось у нас. Вероятно, ваш ответ не совпадет с нашим полностью, но он должен быть достаточно близок к нему и в него должны быть включены те же четыре базовые функции.

Приложение RouteFinder

Список функциональных возможностей

1. Представление линий метро и станций на каждой линии.
2. Загрузка данных нескольких линий метро, в том числе и пересекающихся.
3. Определение действительного пути между любыми двумя станциями на любых линиях.
4. Распечатка маршрута между двумя станциями в виде набора указаний.

Линии пересекаются, если они содержат одну и ту же станцию.

«Действительный маршрут» может проходить по одной линии, а может включать несколько разных линий.

Распечатка — отдельная функциональная возможность... Не забудьте об этом при построении своего списка.

Часто задаваемые вопросы

В: Почему мы не занимаемся сбором требований? Кстати говоря, я до сих пор плохо понимаю, чем функциональная возможность отличается от требования.

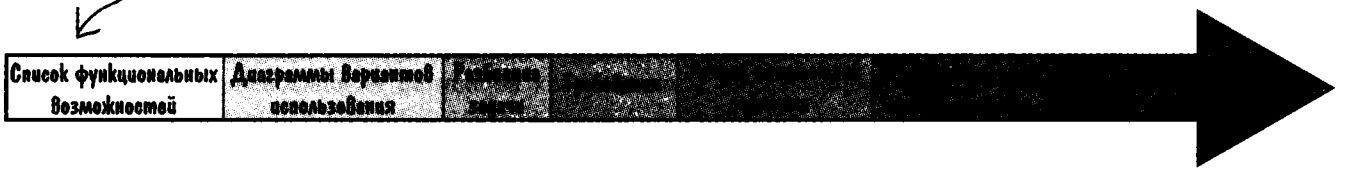
О: Понятия «функциональная возможность» и «требование» часто используются как синонимы. Однако в большинстве случаев под «функциональными возможностями» понимаются БОЛЬШИЕ задачи, решаемые приложением. Таким образом, в реализации одной функциональной возможности может быть задействовано несколько требований. И так как функциональные возможности обычно относятся к более высокому уровню, любой новый проект рекомендуется начинать со списка функциональных требований, как мы это делаем в приложении RouteFinder.

В: Почему вывод маршрута указан отдельным пунктом в списке? Ведь если маршрут между двумя станциями известен, вывести его несложно?

О: Да, пожалуй. Но список функциональных возможностей — это не список трудных задач, которые должно решать приложение; это список всего, что должно делать приложение. Таким образом, даже если функциональная возможность выглядит простой или тривиальной, она все равно должна войти в список.

Вы уже знаете, что делать дальше.

К настоящему моменту мы завершили первую фазу.



При наличии списка функциональных возможностей вы хорошо представляете, что должно делать приложение. Вероятно, вы даже начали продумывать его структуру, хотя вскоре мы выделим на это существенно больше времени.

Разобравшись со списком функциональных возможностей, переходите к диаграммам вариантов использования. Диаграммы помогут связать то, что *делает* приложение, с тем, как оно будет *использоваться*, то есть с тем, что интересует заказчика.

Списки функциональных возможностей помогают понять, что должна делать программа.

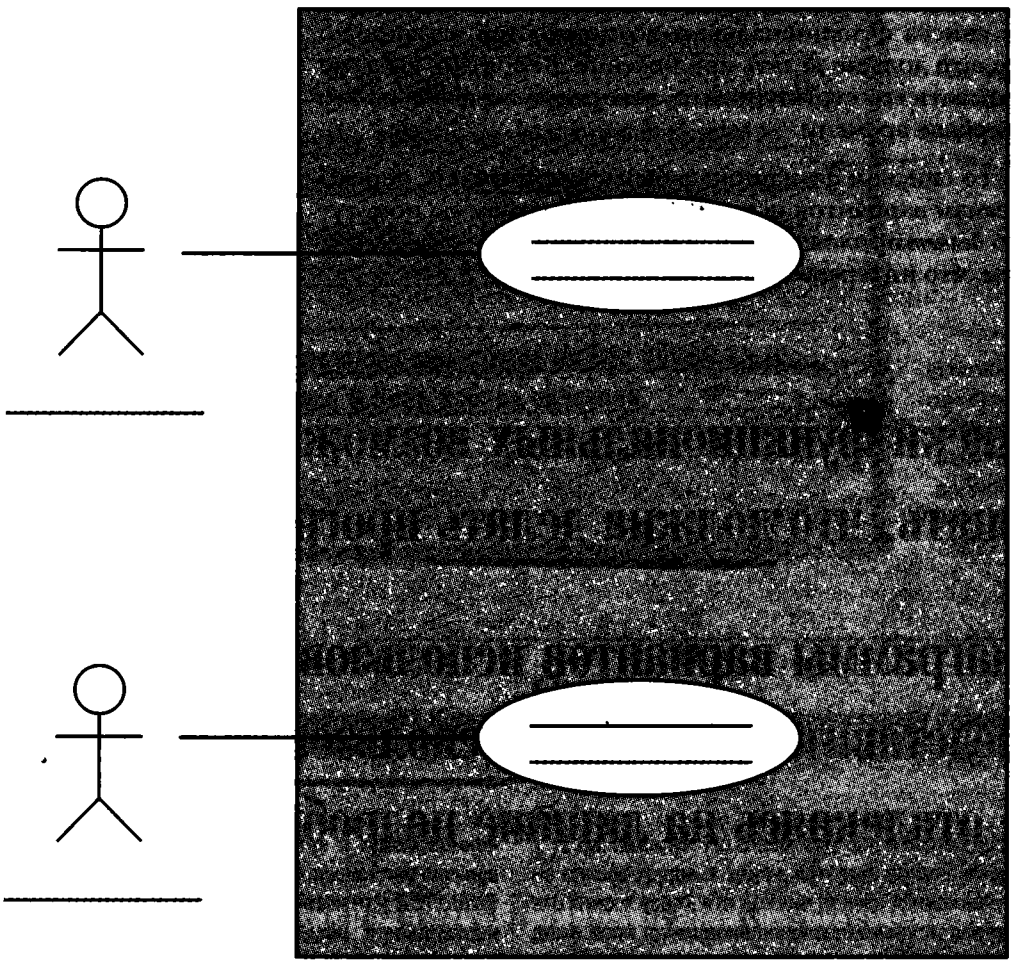
Диаграммы вариантов использования помогают представить, как будет использоваться программа, не отвлекаясь на лишние подробности.

Возьми в руку карандаш

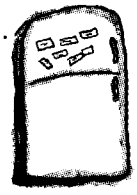
Определите структуру кода RouteFinder.

Располагая списком функциональных возможностей, давайте посмотрим, как будет использоваться наше приложение. Ниже представлена диаграмма вариантов использования для приложения RouteFinder. В нашем проекте задействованы два субъекта и всего два варианта использования (звучит просто, не правда ли?).

Определите, кто (или что) является субъектом, и опишите вариант использования. За дополнительной информацией о диаграммах вариантов использования обращайтесь к главе 6.



Ответы на с. 516.



Развлечения с магнитами

Когда диаграммы вариантов использования будут готовы, необходимо проверить, что варианты соответствуют функциональным возможностям, которые необходимо реализовать для заказчика. Ниже приведены магниты с текстом всех функциональных возможностей RouteFinder. Разместите каждый магнит рядом с одним из вариантов использования, которые были определены на с. 514. Прежде чем переворачивать страницу, убедитесь в том, что варианты использования покрывают все функциональные возможности.

Приложение RouteFinder

Список функциональных возможностей

1. **Представление линий метро и станций на каждой линии.**
2. **Загрузка данных нескольких линий метро, в числе и пересекающихся.**
3. **Определение действительного пути между любыми двумя станциями на любых линиях.**
4. **Вывод маршрута между двумя станциями в виде набора инструкций.**

Представление линий метро и станций на каждой линии.

Загрузка данных нескольких линий метро.

Определение действительного пути между любыми двумя станциями на любых линиях.

Вывод маршрута между двумя станциями.

Каждый из магнитов должен находиться на одном из этих вариантов использования.

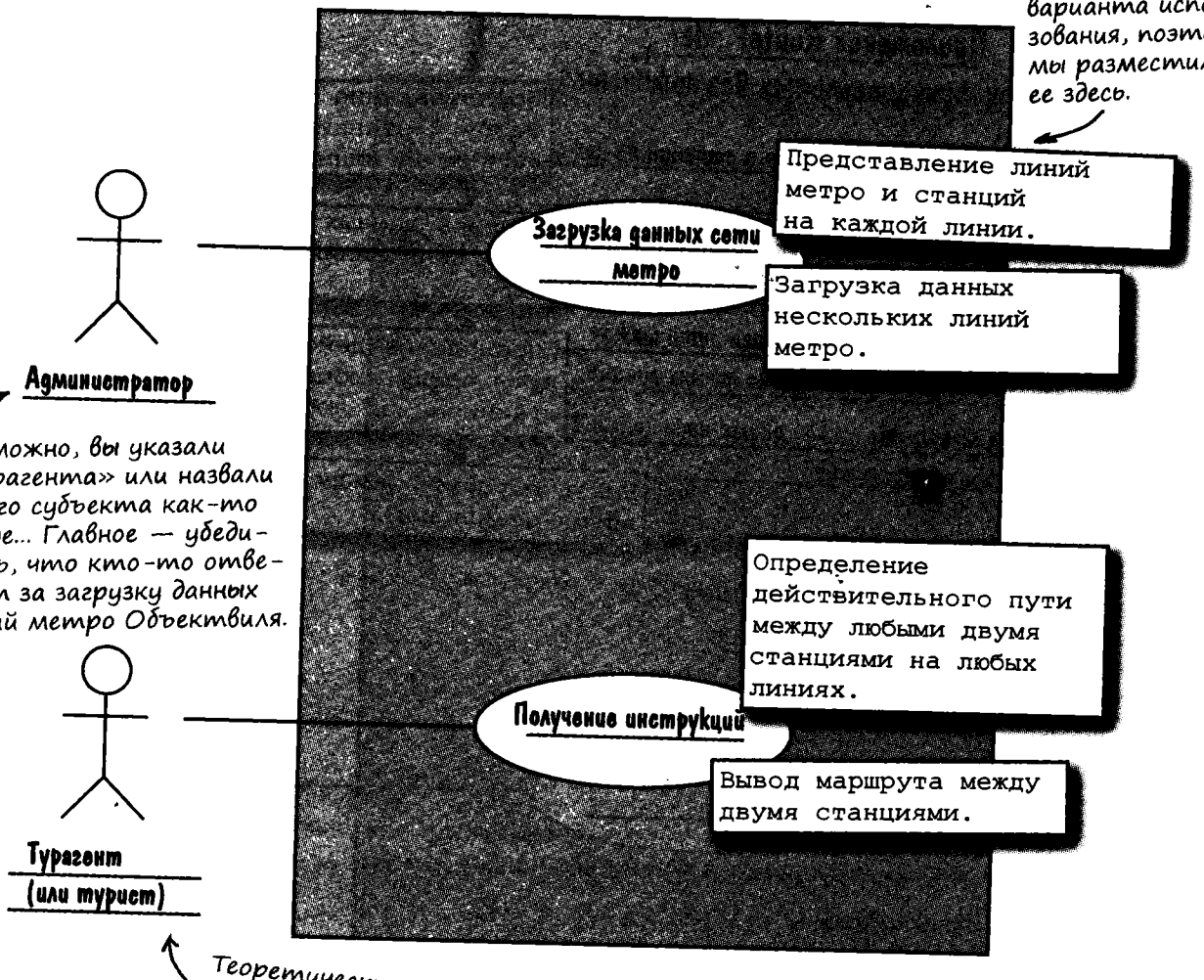
Вы находитесь ЗДЕСЬ.

Возьми в руку карандаш
Решение

Определите структуру кода RouteFinder и свяжите варианты использования со списком функциональности.

Вам было предложено определить субъектов и варианты использования, а также убедиться в том, что варианты использования охватывают все функциональные возможности приложения RouteFinder.

Функциональная возможность необходима для работы данного варианта использования, поэтому мы разместили ее здесь.

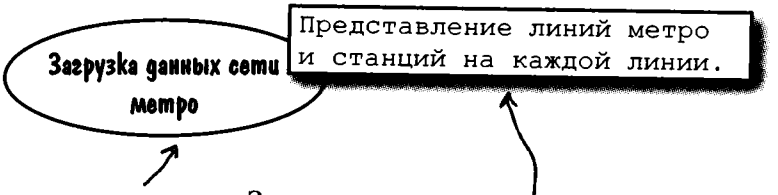


Возможно, вы указали «Турагента» или назвали этого субъекта как-то иначе... Главное — убедитесь, что кто-то отвечает за загрузку данных линий метро Объектвиля.

Теоретически этот блок функциональности могут использовать как турагенты, так и туристы.

Варианты использования отражают типичное использование, а функциональные возможности — функциональность приложения

Давайте повнимательнее рассмотрим одну из ассоциаций «функциональная возможность — вариант использования» на предыдущей странице:



Вариант использования «Загрузка данных сети метро» не использует напрямую функциональную возможность представления сети. Разумеется, представление сети необходимо для работы варианта использования, но подобное связывание выглядит немного искусственным.

Тем не менее оно не является ошибкой: при написании вариантов использования мы имеем дело только с **взаимодействиями** между субъектами и системой. Речь идет о вариантах **использования** системы (отсюда и взялся сам термин).

Способы использования нашего приложения RouteFinder.

Функция должна поддерживаться приложением RouteFinder, чтобы вариант использования работал, но НЕ ЯВЛЯЕТСЯ самостоятельным взаимодействием или частью какого-либо варианта использования.

Вариант использования может зависеть от функциональной возможности, но последняя может и не являться частью самого варианта. «Загрузка данных сети метро» зависит от представления сети, но использует эту функциональную возможность косвенно.

Возможности системы отражают ее функциональность. Система должна делать все перечисленное, чтобы варианты использования работали, даже когда эта функциональность неявно присутствует в конкретных вариантах использования.

Вы (по-прежнему) находитесь ЗДЕСЬ.

Часто задаваемые вопросы

В: Разве вы не говорили, что каждая функциональная возможность должна связываться с некоторым вариантом использования в моей системе?

О: Да, и это действительно так. Каждая функциональная возможность в вашей системе должна хотя бы частично быть задействована в описании одного или нескольких вариантов использования на диаграмме. Однако это не означает, что вариант использования должен напрямую задействовать эту функциональную возможность, достаточно часто функциональная возможность обеспечивает работу варианта использования без прямого участия в нем.

В приложении RouteFinder невозможно загрузить данные сети метро без способа представления линий (одна из функциональных возможностей). Но вариант использования «Загрузка данных сети» не содержит шагов, напрямую соответствующих этой функциональной возможности... Просто предполагается, что представление сети существует. Таким образом, вариант использует эту функциональную возможность косвенно.

В: А все-таки вариант использования — это требование... Или функциональная возможность — это требование?

О: И то и другое! Варианты использования определяют требования к взаимодействию людей и других сущностей (субъектов) с системой, а функциональные возможности определяют требования к тому, что должна делать ваша система. Они связаны между собой, но это не одно и то же. Тем не менее для реализации вариантов использования системы вам понадобится функциональность, заложенная в ее функциональных возможностях. Вот почему всегда следует связывать функциональные возможности с вариантами использования, работу которых они обеспечивают.

В: А если какую-то функциональную возможность не удается связать с вариантом даже косвенно?

О: Хорошенько присмотритесь к ней и убедитесь, что она действительно является необходимой частью вашей системы. Ваш заказчик (и его клиенты) взаимодействует с системой только через вариант использования. Таким образом, если функциональная возможность не задействована в варианте использования хотя бы косвенно, она не принесет никакой пользы заказчику. Если вы полагаете, что конкретная функциональная возможность никак не влияет на поведение системы, обсудите это с заказчиком. Не бойтесь исключать из системы то, что не улучшает ее.

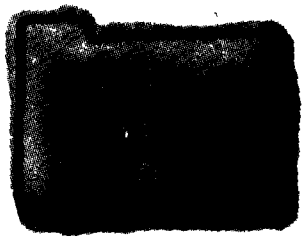
Функциональные возможности — это описание того, что делает система. Они не всегда напрямую отражаются в вариантах использования.

Функциональные возможности и варианты использования работают вместе, но это не одно и то же.

Как разбить систему на модули

Работа постепенно набирает темп. При наличии диаграмм вариантов использования можно переходить к разбиению задачи на меньшие функциональные блоки. В приложении это можно сделать несколькими способами, но мы постараемся сделать RouteFinder модульной системой. А для этого следует четко разделить разные функциональные блоки — каждый модуль должен иметь единственную обязанность.

Разбейте систему RouteFinder на четыре «модуля». Тщательно продумайте оптимальный способ разбиения системы... Он необязательно должен совпадать с четырьмя функциональными возможностями системы (хотя может оказаться и так!)



Запишите в этом поле краткое обозначение функциональности модуля.

ПОДСКАЗКА: Не забудьте, что наряду с организацией функциональности системы вам также необходимо продемонстрировать заказчику, что система работает.

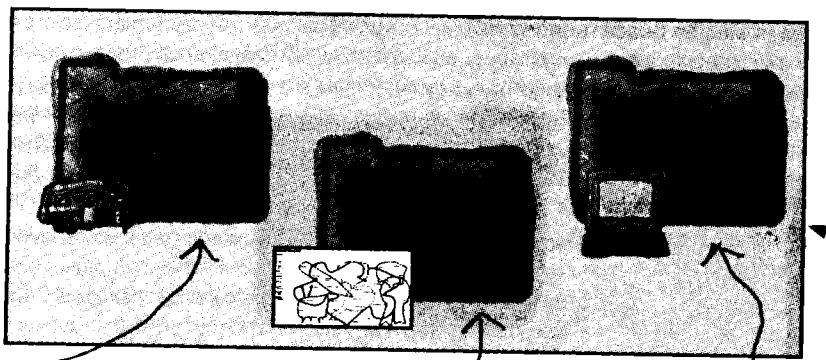
Мы двигаемся вперед.

Как разбить систему на модули. Решение

Работа постепенно набирает темп. Разбейте систему RouteFinder на четыре «модуля».

Эти три модуля образуют своего рода «черный ящик», который используется туристами и турагентами для получения инструкций.

Мы добавили модуль Testing, потому что нам понадобится код, который будет демонстрировать заказчику работоспособность системы... Не только в «идеальных условиях», но и в реальном мире.



Модуль Subway содержит весь код представления станций, связей между станциями и всей системы метро. Он также умеет выдавать инструкции для перемещения от одной станции к другой.

Загрузка данных может осуществляться разными способами: из файла, непосредственно пользователем и даже из базы данных. Загрузка разделена с представлением сети метро, поэтому она выделяется в отдельный модуль.

Вывод, как и загрузка, отделен от представления данных сети. Этот модуль выводит информацию на любое устройство и в любом формате, который нам может понадобиться.

Модуль тестирования находится за пределами системы. Он взаимодействует с системой, но не является ее непосредственной частью.

Возьми в руку карандаш

Какие ОО-принципы мы используем?

Установите флажок рядом с ОО-принципами, которые были использованы при разбиении функциональности RouteFinder.

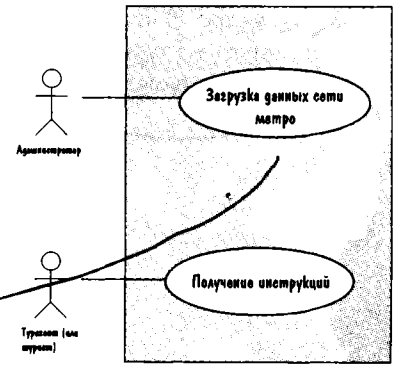
- ☐ Принцип единственной обязанности
- ☐ Принцип отсутствия дублирования
- ☐ Инкапсуляция
- ☐ Делегирование
- ☐ Полиморфизм
- ☐ Принцип подстановки Лисков

Ответы на с. 522.

Начало итераций

Когда система будет разбита на несколько отдельных блоков функциональности, можно переходить к итеративной обработке каждого блока, пока приложение не будет готово.

На этой стадии следует взять общую картину системы, полученную на основании диаграммы вариантов использования, и преобразовать ее в требования, которые мы можем последовательно реализовать. Для первой итерации мы возьмем вариант использования «Загрузка данных сети метро» и преобразуем его в набор требований, но более конкретных. Разобравшись с этим вариантом использования, мы перейдем к следующему и повторим итерацию.



Начинаем с первого варианта использования на диаграмме.

Возьми в руку карандаш

Какой метод разработки мы используем в этой главе?

Возьми в руку карандаш

Напишите вариант использования для загрузки линий метро.

Превратите маленький овал на диаграмме в полноценный вариант использования. Напишите те действия, которые должна реализовать система RouteFinder для реализации «Загрузки данных сети метро».

Загрузка данных сети метро

Вариант использования

1. _____
2. _____
3. _____
4. _____
5. _____
6. _____

При необходимости используйте больше места или увеличьте количество шагов.

Сомневаетесь? Ничего страшного... Дополнительная информация на следующей странице.

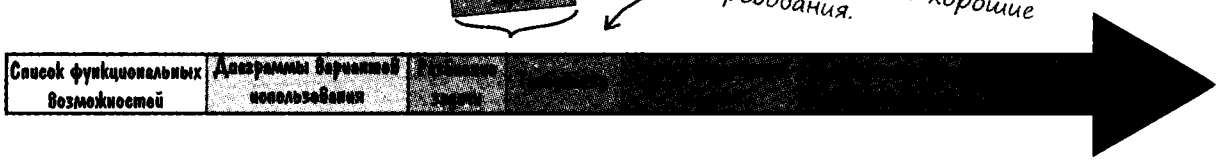
И как я буду писать этот вариант? Ведь я еще толком не знаю, что собой представляет линия метро. И что с тем файлом, который обещали прислать из туристического бюро Объективиля? Разве он не повлияет на то, как будет написан наш вариант использования?

Часто между разбиением задачи и написанием вариантов использования приходится дополнительно потрудиться.

Вы только что обнаружили «скрытый шаг» нашего процесса написания программ по правилам ООАП:

«Дополнительный шаг» иногда бывает необходим.

Мы просто недостаточно хорошо понимаем задачу, чтобы написать хорошие требования.



Если у вас возникнут трудности с написанием варианта использования, попробуйте сделать шаг назад и проанализировать задачу, которую пытаетесь решить, затем вернитесь к варианту использования — теперь вам будет проще правильно сформулировать его.

Разбиение вывода, загрузки и представления на три модуля позволяет добиться того, чтобы у каждого модуля была только одна причина для изменения.

Возьми в руку карандаш

Решение

Какие ОО-принципы мы используем?

Установите флажок рядом с ОО-принципами, которые, по вашему мнению, были использованы при разбиении функциональности RouteFinder.

- [x] Принцип единственной обязанности
- [ ] Принцип отсутствия дублирования
- [x] Инкапсуляция
- [x] Делегирование
- [ ] Полиморфизм
- [ ] Принцип подстановки Лисков

Операции вывода и загрузки могут изменяться и поэтому отделены от неизменяющегося представления сети.

Пока непонятно, будет ли использоваться делегирование в нашей системе, хотя вероятно, что с SRP и инкапсуляцией будет.

Подробнее о представлении линий метро

Прежде чем разбираться, как загрузить данные линий метро, необходимо хорошо понять две вещи:

1. Что собой представляет система линий метро.
2. Какой информацией будет располагать администратор при загрузке набора станций и линий метро.

Мы находимся на итерации 1 и занимаемся первым вариантом использования: загрузкой данных сети метро.

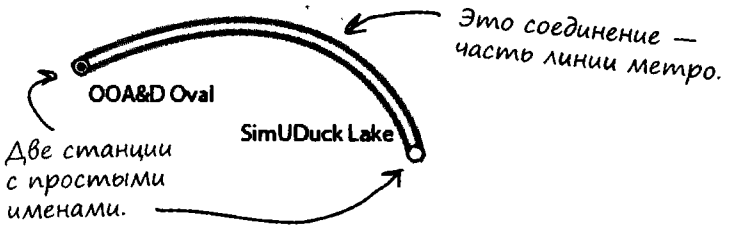
Что такое станция?

Система метро состоит из станций, соединений между станциями и линий, которые представляют собой совокупности соединений. Так что давайте для начала определим, что же собой представляет станция.

Станция — точка на карте, которой присвоено имя.

А соединение между станциями?

С появлением нескольких станций необходимо как-то представить соединения между этими станциями:

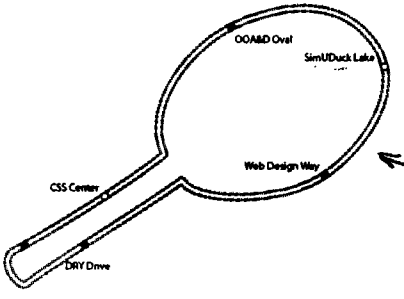


Это соединение — часть линии метро.

Две станции с простыми именами.

Тогда линия — это просто совокупность соединений...

Объединив несколько соединений, мы получаем линию метро.



Линия метро представляет собой множество станций, каждая из которых соединена с другой станцией.

Возьми в руку карандаш

Решение

Какой метод разработки мы используем в этой главе?

Сценарно-ориентированная разработка

Заглянем в файл с описанием метро

Итак, теперь мы примерно представляем, как устроена сеть метро. Давайте посмотрим, с какими данными нам придется работать. Помните, в турагентстве обещали прислать файл с перечнем всех станций и линий, чтобы мы поняли, какая информация будет использоваться администратором для загрузки данных в систему.

ObjectvilleSubway.txt — файл, который нам прислали из турагентства Объектвиля. Файл содержит информацию, которая будет использоваться для загрузки данных сети.

```
Ajax Rapids
HTML Heights
JavaBeans Boulevard
LSP Lane
Head First Labs
Objectville PizzaStore
UML Walk
XHTML Expressway
Choc-O-Holic, Inc.
Head First Theater
Infinite Circle
CSS Center
OOA&D Oval
SimUDuck Lake
Web Design Way
... другие названия станций...

Booch Line
Ajax Rapids
HTML Heights
JavaBeans Boulevard
LSP Lane
Head First Labs
Objectville PizzaStore
UML Walk
Ajax Rapids

Gamma Line
OOA&D Oval
Head First Lounge
OOA&D Oval

Jacobson Line
Servlet Springs
... другие станции на этой линии...

... другие линии...
```

Файл начинается со списка всех названий станций.

Название линии...

...и последовательное перечисление станций на линии.

На кольцевых линиях первая станция также указывается как последняя.

Линии разделяются пустыми строками.

Вы сейчас находитесь ЗДЕСЬ.

Список функциональных возможностей	Диаграммы вариантов использования	Разбиение задачи	Требования

Возьми в руку карандаш

Напишите вариант использования для загрузки линий метро.

Сейчас вы уже достаточно хорошо понимаете структуру сети метро и формат входных данных, чтобы написать вариант использования.

Загрузка данных сети метро

Вариант использования

1. _____
2. _____
3. _____
4. _____
5. _____
6. _____
7. _____
8. _____
9. _____

При необходимости используйте больше места или увеличьте количество шагов.

Возьми в руку карандаш
Решение

Напишите вариант использования для загрузки линий метро.

Сейчас вы уже достаточно хорошо понимаете структуру сети метро и формат входных данных, чтобы написать вариант использования.

Начинаем с исходного условия: загрузчик получает файл, из которого должны загружаться данные.

А вы не забыли об этом шаге? Станции не должны дублироваться... Позднее дубликаты создадут слишком много проблем.

Еще один шаг проверки входных данных: убедиться, что соединение не противоречит структуре существующей сети.

Шаги с описанием условия повторения других шагов делают варианты использования более точными и понятными.

Загрузка данных сети метро

Вариант использования

1. Администратор передает файл с информацией о станциях и линиях.
2. Система читает название станции.
3. Система проверяет, существует ли такая станция.
4. Система добавляет новую станцию в сеть.
5. Система повторяет шаги 2-4, пока не будут загружены все станции.
6. Система читает название линии.
7. Система читает названия двух соединенных станций.
8. Система проверяет, что эти станции существуют.
9. Система создает новое двустороннее соединение между двумя станциями на текущей линии.
10. Система повторяет шаги 7-9, пока линия не будет завершена.
11. Система повторяет шаги 6-10, пока не будут загружены данные всех линий.

Это всего лишь один из возможных способов написания варианта использования. Вместо создания объекта SubwayLine мы решили собрать информацию обо всех соединениях и связать каждое соединение с некоторой линией метро.

Отмечать это не обязательно, но мы подумали, что будет важно напомнить, что поезда в метро Объектвиля двигаются в ОБОИХ направлениях.

Часто задаваемые вопросы

В: Мой вариант использования выглядит совсем не так. Ваш вариант — это единственное решение?

О: Вовсе нет. К настоящему моменту вы уже должны знать, что при решении любой задачи приходится принимать множество решений, и наш вариант использования просто отражает решения, принятые нами. Мы будем работать с этим конкретным вариантом до конца главы, чтобы вы поняли логику действий, но если вы создали собственный вариант, решающий ту же задачу загрузки сети станций и линий, — это абсолютно нормально.

В: В моем варианте нет шагов проверки данных. Это нормально?

О: Проверку данных обязательно следует добавить, если вы забыли о ней. Проверка существования станций, участвующих в соединении, — примерно то же, что проверка препятствий для закрытия собачьей двери: вроде бы несущественно — до тех пор, пока реальный мир не нарушит ваши планы и кто-нибудь не перепутает название станции. И внезапно в системе появится соединение, ведущее к несуществующей станции. Если вы забыли добавить проверку данных, непременно включите ее в свой вариант.

Проверяем, работает ли наш вариант использования

Вариант использования загрузки данных сети получился довольно запутанным, и он содержит несколько повторяющихся групп шагов. Давайте проверим последовательность действий по текстовому файлу, прежде чем браться за анализ варианта использования и проектирование классов нашей системы.

1. Администратор передает загрузчику файл следующего вида.

Ajax Rapids
HTML Heights
JavaBeans Boulevard
LSP Lane
Head First Labs
Objectville PizzaStore
UML Walk
XHTML Expressway
Choc-O-Holic, Inc.
Head First Theater
Infinite Circle
CSS Center
OOA&D Oval
SimUDuck Lake
Web Design Way
... другие названия станций...

Booch Line
Ajax Rapids
HTML Heights
JavaBeans Boulevard
LSP Lane
Head First Labs
Objectville PizzaStore
UML Walk
Ajax Rapids

Gamma Line
OOA&D Oval
Head First Lounge
OOA&D Oval

Jacobson Line
Servlet Springs
... другие станции на линии ...

... другие линии...

2. Система читает название станции.

3. Система проверяет, не была ли эта станция уже загружена.

4. Система добавляет станцию в сеть.

5. Система повторяет эти шаги, пока не будут введены все станции.

6. Система читает название линии.

7. Система читает названия двух соединенных станций.

8. Система проверяет, что обе станции существуют в сети.

9. Система создает новое соединение между станциями.

10. Эти шаги повторяются для всех последующих пар станций на линии.

11. Процесс добавления линии повторяется для каждой линии в файле.

Головоломка

На этот раз вам предстоит одолеть сразу две фазы процесса ООАП. Сначала выполните текстологический анализ приведенного варианта использования и найдите существительные, которые являются кандидатами для классов, затем глаголы — кандидаты для операций. Запишите эти существительные и глаголы в пустых полях под вариантом использования.

Загрузка данных сети метро

Вариант использования

1. Администратор передает файл с информацией о станциях и линиях.
2. Система читает название станции.
3. Система проверяет, существует ли такая станция.
4. Система добавляет новую станцию в сеть.
5. Система повторяет шаги 2-4, пока не будут загружены все станции.
6. Система читает название линии.
7. Система читает названия двух соединенных станций.
8. Система проверяет, что эти станции существуют.
9. Система создает новое двустороннее соединение между двумя станциями на текущей линии.
10. Система повторяет шаги 7-9, пока линия не будет завершена.
11. Система повторяет шаги 6-10, пока не будут загружены данные всех линий.

Существительные (кандидаты для классов):

_____ _____ _____

_____ _____ _____

Глаголы (кандидаты для операций):

_____ _____ _____

_____ _____ _____

Переходим к фазе предварительного проектирования. Используя существительные и глаголы, полученные из варианта использования, нарисуйте внизу диаграмму классов, которая показывает, как, по вашему мнению, должна выглядеть модель системы метро в программном коде. Используйте ассоциации и все остальные конструкции UML, с которыми ваша структура станет четкой и понятной.

Эта часть головоломки относится к анализу предметной области...

...а эта — к предварительному проектированию.

Головоломка. Решение

Загрузка данных сети метро

Вариант использования

1. Администратор передает файл с информацией о станциях и линиях.
2. Система читает название станции.
3. Система проверяет, существует ли такая станция.
4. Система добавляет новую станцию в сеть.
5. Система повторяет шаги 2-4, пока не будут загружены все станции.
6. Система читает название линии.
7. Система читает названия двух соединенных станций.
8. Система проверяет, что эти станции существуют.
9. Система создает новое двустороннее соединение между двумя станциями на текущей линии.
10. Система повторяет шаги 7-9, пока линия не будет завершена.
11. Система повторяет шаги 6-10, пока не будут загружены данные всех линий.

Здесь все просто... Все кандидаты присутствуют на диаграмме классов.

Существительные (кандидаты для классов):

~~администратор~~ ~~система~~ сеть линия

~~файл~~ станция соединение

Мы знаем, что субъекты находятся вне системы, поэтому класс не понадобится.

Это входные данные для нашей системы, а не что-то такое, что мы моделируем в своей системе.

Мы решили не создавать класс Line... Подробности на следующей странице.

Глаголы (кандидаты для операций)

передает файл ~~читает~~

Для этого можно использовать операции ввода/вывода Java.

проверяет станцию добавляет станцию ~~повторяет~~ добавляет подключение

Большинство операций напрямую соответствует методам наших классов.

Класс загрузчика содержит мало функциональности... Он просто получает файл и генерирует по нему экземпляр Subway.

SubwayLoader

loadFromFile(File): Subway

Этот класс использует всю функциональность Subway для создания сети метро по текстовому файлу с описаниями станций и линий.

Основной объект, с которым мы будем работать. Должен содержать метод для добавления станций и соединений.

Subway

addStation(String)
hasStation(String): boolean
addConnection(String, String, String)

Вы должны были определить по варианту использования, что нам понадобится операция для проверки существования станции. Она станет частью проверки правильности данных станций на шагах 2 и 5.

Объект Subway содержит коллекции станций и соединений между станциями.

stations *

connections *

Соединение состоит из двух станций и названия линии, в которую входит это соединение.

Класс Station очень прост... Он содержит название станции, и все. Вскоре мы объясним, для чего в него были добавлены методы equals() и hashCode().

Station

name: String

getName(): String
equals(Object): boolean
hashCode(): int

station1

station2

Connection

lineName: String

getStation1(): Station
getStation2(): Station
getLineName(): String

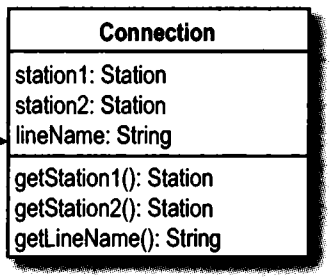
Эти атрибуты не следует называть start и stop (или beginning и end), потому что линии метро в Объектвиле являются двусторонними. Мы выбрали имена station1 и station2.

Мы занимаемся проектированием и двигаемся по направлению к реализации текущей итерации.

Использовать класс Line или не использовать класс Line... Вот в чем вопрос

По варианту использования мы без особого труда определили, что нам понадобятся классы **Station**, **Connection** и **Subway** – они играют основополагающую роль в нашей системе. Но затем мы решили отказаться от создания класса **Line**. Вместо этого с каждым соединением просто связывается название линии:

Линии существуют только в виде названий (String), связываемых с конкретными соединениями.



Это решение было принято по единственной причине: мы знаем, как будет использоваться система. В исходной постановке задачи (см. с. 506) нам было сказано, что нам придется ввести представление сети и получать инструкции по перемещениям между станциями. При наличии инструкций мы можем просто запросить название линии у каждого соединения; класс **Line** для этого не нужен.

Ваши структурные решения должны быть основаны на предполагаемом использовании системы и на хороших ОО-принципах.

Часто задаваемые вопросы

В: Я вижу глагол-действие «проверяет существование станции», но такой операции нет ни в одном из классов предложенной вами структуры?

О: Мы смоделировали эту операцию в методе `hasStation()` класса `Subway`. Операцию также можно было назвать `validate()` или `validateStation()`, но эти названия не так информативны, как `hasStation()`. Всегда старайтесь сделать код по как можно более понятным.

В: Можно поподробнее о представлении повторяющихся шагов в варианте использования?

О: Достаточно часто вариант использования содержит группы шагов, которые должны повторяться, но стандартного представления для таких повторений не существует. И нам пришлось его выдумать! Цель варианта использования — представить набор шагов, подробно описывающих, что должна делать система. Мы решили, что самый понятный способ представления повторений — запись вида «Повторить шаги 7–9».

В: Я тут думал о представлении сети, и мне кажется, что эта структура данных очень похожа на граф. Почему мы не используем граф?

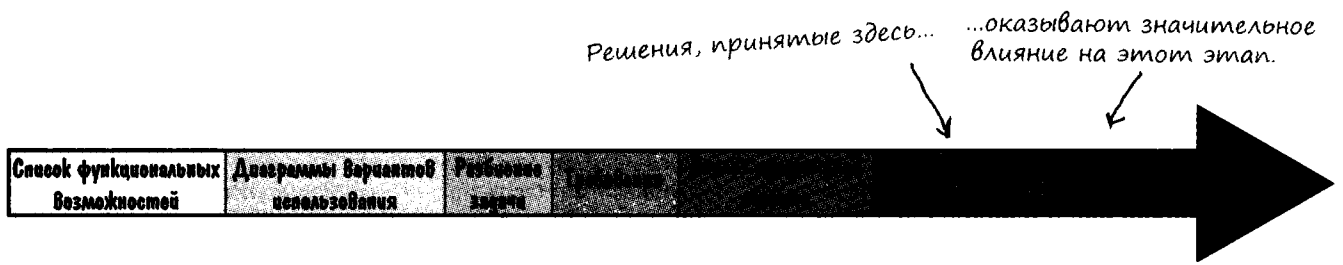
О: Ого, да вы недавно изучали структуры данных и алгоритмы! Для представления сети действительно можно использовать граф. В этом случае каждой станции будет соответствовать узел, а каждому соединению — ребро.

В: Тогда почему мы не используем граф в данном примере?

О: Нам показалось, что в данной ситуации это будет слишком сложно. Если вы знакомы с графами, узлами и ребрами и у вас имеется готовый код для таких структур данных, используйте его. Но с нашей точки зрения, адаптация графа потребует больше работы, чем написание нескольких простых классов вроде `Station` и `Connection`. Как обычно бывает на стадии проектирования, задачу можно решить несколькими способами, и вы должны выбрать решение, которое лучше всего подходит вам.

В: Я ничего не понял... программы.... какие-то ребра-узлы... Что это такое?

О: Не страшно; для понимания и решения этой конкретной задачи разбираться в графах не нужно. Вам не о чем беспокоиться, по крайней мере, пока мы не напишем книгу из серии Head First о структурах данных.



Программирование класса Station

У нас имеются требования в форме варианта использования, диаграмма классов, и мы знаем, что класс **Station** будет использоваться в нашей модели **Subway**. Можно переходить к написанию кода:

Мы определили модуль Subway на с. 520.

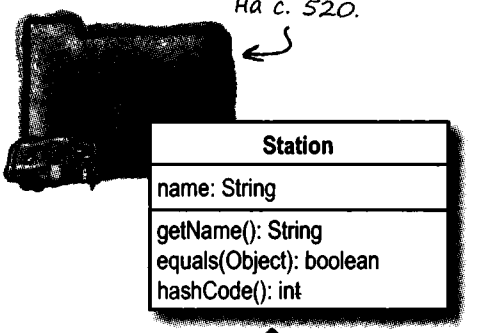


Диаграмма классов для Station, см. с. 534.

Если хотите, поместите Station в пакет вида objectville.subway. Разбиение модулей по пакетам — ваше личное дело.

```
public class Station {
  private String name;

  public Station(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public boolean equals(Object obj) {
    if (obj instanceof Station) {
      Station otherStation = (Station)obj;
      if (otherStation.getName().equalsIgnoreCase(name)) {
        return true;
      }
    }
    return false;
  }

  public int hashCode() {
    return name.toLowerCase().hashCode();
  }
}
```

В сущности, объект Station состоит только из имени.

Таким образом мы гарантируем, что станция с названием «AJAX RAPIDS» будет считаться той же станцией, что и «Ajax Rapids».

Поскольку станции будут часто сравниваться друг с другом, мы определили метод equals(). В данной версии два объекта Station считаются равными, если им присвоено одно название.

Хэш-код Station вычисляется на основании того же свойства, по которому выполняются сравнения: названия станции.

Будьте осторожны!

При переопределении equals() в Java следует переопределить hashCode(), чтобы сравнения работали правильно.

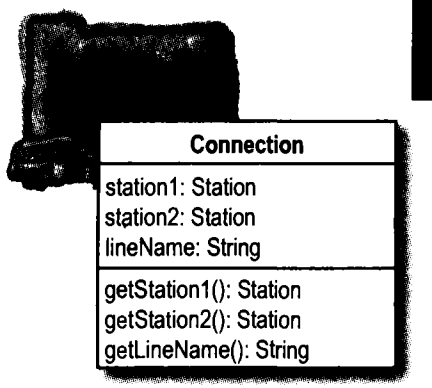
Спецификация Java говорит, что два одинаковых объекта должны иметь одинаковые хэш-коды. Соответственно если проверка равенства осуществляется по значению свойства, желательно переопределить hashCode() и вернуть хэш-код, основанный на том же свойстве. Это важно при использовании объекта в Hashtable или HashMap, которые интенсивно используют метод hashCode().

За более подробной информацией о equals() и hashCode() обращайтесь к главе 16 книги «Head First Java».

Возьми в руку карандаш

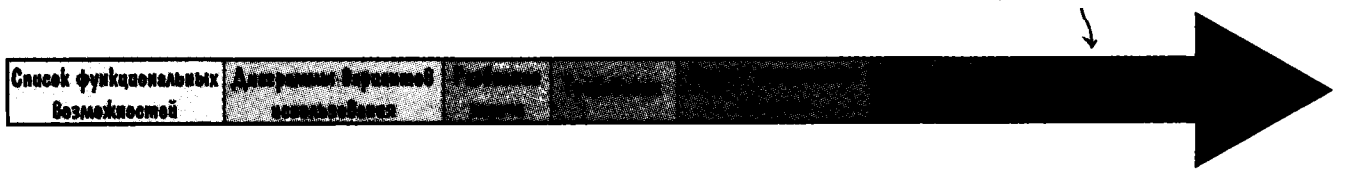
Напишите класс Connection.

Используя приведенную справа диаграмму, допишите класс Connection — заполните пропуски правильным кодом. Прежде чем переворачивать страницу и просматривать наш ответ, убедитесь в том, что ваш класс успешно компилируется.




```
public class Connection {

  private ________ ___________, _____________;
  private ________ ___________;

  public Connection(_________ __________, _________ __________,
                    __________ __________) {
    this.__________ = station1;
    this.__________ = station2;
    this.__________ = lineName;
  }

  public ________ ___________() {
    ________ station1;
  }

  public ________ ___________() {
    ________ station2;
  }

  public ________ ___________() {
    ________ lineName;
  }
}
```

Сейчас мы углубились в фазу реализации первой итерации.



Возьми в руку карандаш
Решение

Напишите класс Connection.

Используя приведенную справа диаграмму, заполните пропуски и допишите класс Connection.

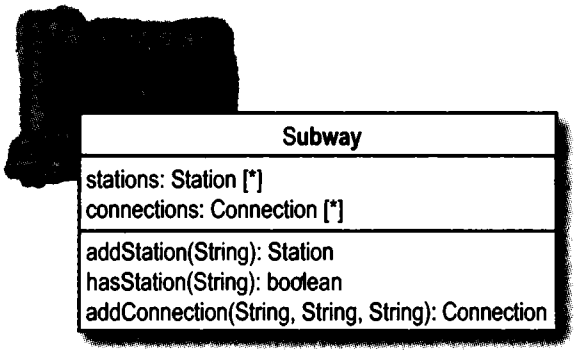
Connection
station1: Station station2: Station lineName: String
getStation1(): Station getStation2(): Station getLineName(): String

```
public class Connection {

  private  Station   station1   ,  station2   ;
  private  String    lineName   ;

  public Connection( Station   station1   ,  Station   station2   ,
                     String    lineName   ) {
    this. station1   = station1;
    this. station2   = station2;
    this. lineName   = lineName;
  }

  public  Station   getStation1   () {
    return   station1;
  }

  public  Station   getStation2   () {
    return   station2;
  }

  public  String    getLineName   () {
    return   lineName;
  }
}
```

Мы сейчас здесь.

Программирование класса Subway

Дальше идет сам класс **Subway**. С готовыми классами **Station** и **Connection**, а также хорошей диаграммой классов сюрпризов быть не должно:




```
public class Subway {

  private List stations;
  private List connections;

  public Subway() {
    this.stations = new LinkedList();
    this.connections = new LinkedList();
  }

  public void addStation(String stationName) {
    if (!this.hasStation(station)) {
      Station station = new Station(stationName);
      stations.add(station);
    }
  }

  public boolean hasStation(String stationName) {
    return stations.contains(new Station(stationName));
  }

  public void addConnection(String station1Name, String station2Name,
                            String lineName) {
    if ((this.hasStation(station1Name)) &&
        (this.hasStation(station2Name))) {
      Station station1 = new Station(station1Name);
      Station station2 = new Station(station2Name);
      Connection connection = new Connection(station1, station2, lineName);
      connections.add(connection);
      connections.add(new Connection(station2, station1,
                                     connection.getLineName()));
    } else {
      throw new RuntimeException("Invalid connection!");
    }
  }
}
```

Здесь будут храниться все станции и соединения между станциями

Сначала мы проверяем название и убеждаемся в том, что станция еще не присутствует в списке.

Если станция не найдена, создаем новый экземпляр Station и добавляем его в сеть.

Метод проверяет, присутствует ли станция в контейнере List объекта сети.

Как и с addStation(), начинаем с проверки, существуют ли в сети обе станции.

ОЧЕНЬ важный шаг: так как метро Объектвиля является двусторонним, в сеть добавляются два соединения — по одному для каждого направления.

Не лучший вариант обработки ошибок... Попробуйте придумать более удачный способ обработки ситуаций, в которых одна станция соединения не существует.

Достопримечательности (класса) метро Объектвиля

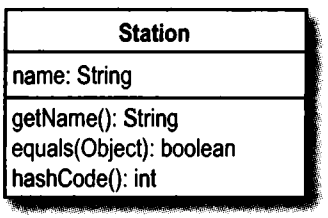
В классе **Subway** есть пара моментов, заслуживающих внимания. Во-первых, в нем довольно часто встречается строка:

```
Station station = new Station(stationName);
```

Например, при создании нового экземпляра **Connection** используется следующий код:

```
Station station1 = new Station(station1Name);
Station station2 = new Station(station2Name);
Connection connection =
  new Connection(station1, station2, lineName);
```

Чтобы найти объект **Station** с именем **station1Name**, многие программисты воспользуются перебором списка станций в классе **Subway**. Но перебор займет много времени, и существует более эффективное решение. Помните, что мы определили в классе **Station** методы **equals()** и **hashCode()**?



Эти методы указывают Java, что при сравнении двух объектов **Station** следует сравнивать их имена. Если имена равны, то *даже если объекты не расположены по одному адресу памяти*, они должны рассматриваться как равные. Таким образом, вместо того, чтобы искать конкретный объект Station в списке станций класса Subway, гораздо проще создать новый объект Station и использовать его.

Обычно метод equals() в Java просто проверяет, являются ли два объекта ОДНИМ объектом... Иначе говоря, он смотрит, расположены ли обе ссылки по одному адресу в памяти. Но это НЕ ТО, что нам нужно для сравнения двух объектов Station.

Переопределение equals() и hashCode() ускоряет поиск и снижает сложность кода. Архитектурные решения всегда должны улучшать реализацию, а не усложнять ее или делать менее понятной.

Как работает реализация equals() по умолчанию...

Эти два экземпляра Station имеют одинаковые имена, но относятся к двум разным областям памяти... Поэтому вызов equals() говорит:

Object.equals()

Эти два объекта НЕ равны!

И это основательно усложняет код реализации.

Как работает наша реализация equals()...

Наша версия equals() сообщает Java, что объекты должны сравниваться по значению атрибута name, а не по их расположению в памяти.

Station.equals()

Эти два объекта РАВНЫ!

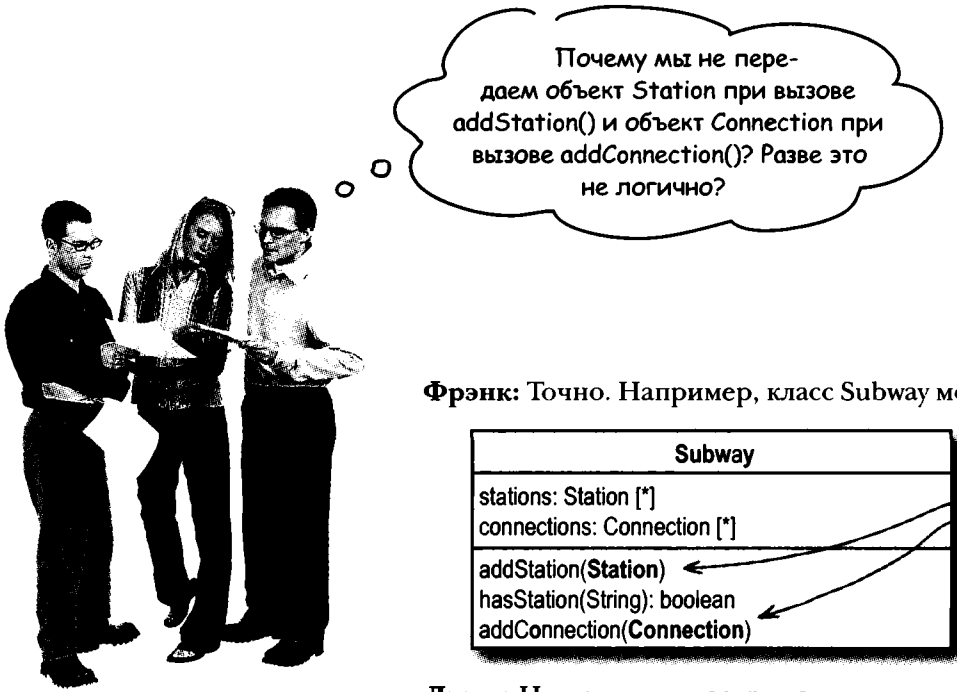
Этот результат существенно упрощает работу с объектами Station.

Часто задаваемые вопросы

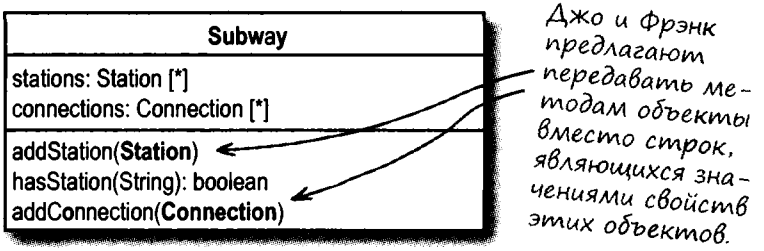
В: Какое отношение это имеет к ООАП?

О: Здесь становится понятна выгода от применения ООАП: хорошо зная свою систему, мы не сомневаемся, что две станции должны считаться идентичными, если они имеют одинаковое название. Кроме того, возможность сравнения станций по названиям, а не по их местонахождению в памяти, улучшает структуру кода.

Таким образом, время, потраченное на сбор требований и анализ системы, улучшает ее структуру, а это, в свою очередь, существенно упрощает реализацию. Знание системы оборачивается гибкостью структуры, а иногда даже более чистым кодом, и все потому, что вы не пожалели времени на то, чтобы выслушать заказчика и собрать требования, вместо того чтобы с ходу браться за IDE и писать исходный код.



Фрэнк: Точно. Например, класс Subway может выглядеть так:



Джил: Но тогда мы раскрываем внутренние механизмы приложения!

Джо: Так-так... Не знаю, о чем ты говоришь, но, похоже, делать этого не стоит. Так о чем ты?

Джил: Посмотри на текущую версию кода. Чтобы загрузить данные сети метро, пользователю не нужно работать с объектами Station или Connection. Он просто вызывает методы нового класса Subway.

Фрэнк: И чем это отличается от того, что мы предлагаем?

Джил: Если пойти по вашему пути, пользователям класса Subway также придется работать со Station и Connection. В текущей версии они работают со строками: названием станции и названием линии.

Джо: И это плохо, потому что...

Фрэнк: Погоди, я, кажется, понял. Их код привязывается к нашей реализации классов Station и Connection, потому что им приходится работать с этими классами напрямую.

Джил: Точно! А в нашей версии мы можем изменить Connection или Station, и нам придется внести изменения только в наш класс Subway. Код пользователей останется неизменным, потому что наши реализации Connection и Station в нем абстрагированы.

Защита ваших классов (и классов клиентов)

То, о чем говорят Фрэнк, Джо и Джил, в действительности является еще одной формой абстракции. Давайте присмотримся повнимательнее:

```
Subway subway = new Subway();
subway.addStation("OOA&D Oval");
subway.addStation("Head First Lounge");
subway.addConnection("OOA&D Oval", "Head First Lounge",
                     "Gamma Line");
```

Клиентский код нигде не работает с объектами Station или Connection — только со строками.

addStation(String)

Subway

addConnection(String, String, String)

Station

Connection

Вся работа со станциями и соединениями осуществляется через сам объект Subway.

Сейчас с объектами Station и Connection взаимодействует только класс Subway.

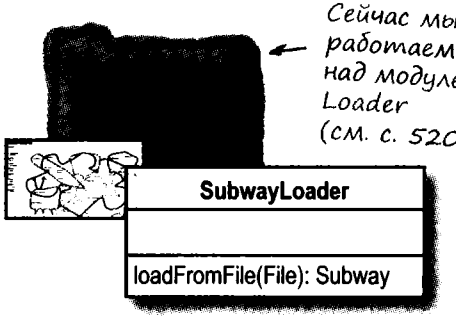
Предоставляйте клиентам доступ только к тем классам, с которыми им НЕОБХОДИМО взаимодействовать.

Классы, с которыми клиент не взаимодействует, можно изменить с минимальными последствиями для клиентского кода.

В этом приложении мы можем изменить внутреннюю реализацию Station и Connection, и это никак не отразится на коде, который работает только с объектом Subway; этот код защищен от изменений в реализации.

Класс SubwayLoader

Первая итерация почти завершена, как и первый вариант использования. Остается запрограммировать класс, загружающий данные из тестового файла, полученного от турагентства.



Сейчас мы работаем над модулем Loader (см. с. 520).

```
public class SubwayLoader {
  private Subway subway;

  public SubwayLoader() {
    this.subway = new Subway();
  }

  public Subway loadFromFile(File subwayFile) throws IOException {
    BufferedReader reader = new BufferedReader(
      new FileReader(subwayFile));

    loadStations(subway, reader);
    String lineName = reader.readLine();
    while ((lineName != null) && (lineName.length() > 0)) {
      loadLine(subway, reader, lineName);
      lineName = reader.readLine();
    }

    return subway;
  }

  private void loadStations(Subway subway, BufferedReader reader)
    throws IOException {
    String currentLine;
    currentLine = reader.readLine();
    while (currentLine.length() > 0) {
      subway.addStation(currentLine);
      currentLine = reader.readLine();
    }
  }

  private void loadLine(Subway subway, BufferedReader reader,
                        String lineName)
    throws IOException {
    String station1Name, station2Name;
    station1Name = reader.readLine();
    station2Name = reader.readLine();
    while ((station2Name != null) && (station2Name.length() > 0)) {
      subway.addConnection(station1Name, station2Name, lineName);
      station1Name = station2Name;
      station2Name = reader.readLine();
    }
  }
}
```

Начинаем с загрузки всех станций.

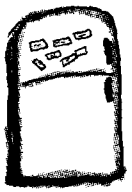
После загрузки станций мы переходим к следующей строке, содержащей название линии, и добавляем расположенные под ней станции в сеть.

Процесс загрузки станций состоит из чтения строки и включения ее в сеть как названия новой станции — этот процесс повторяется до тех пор, пока не будет прочитана пустая строка.

Читаем первую станцию, потом следующую...

...и добавляем новое соединение с использованием названия текущей линии.

Мы берем текущую вторую станцию, делаем ее первой, а затем читаем следующую линию, чтобы получить вторую станцию.



Развлечения с магнитами

Давайте посмотрим, что же происходит, когда мы вызываем метод loadFromFile() из SubwayLoader и передаем ему текстовый файл, полученный от турагентства. Расставьте магниты, находящиеся в нижней части страницы (соответствующие методам классов SubwayLoader и Subway), рядом со строками текстового файла, в которых будут вызываться эти методы.

ObjectvilleSubway.txt

```
Ajax Rapids
HTML Heights
JavaBeans Boulevard
LSP Lane
Head First Labs
Objectville PizzaStore
UML Walk
XHTML Expressway
Choc-O-Holic, Inc.
Head First Theater
Infinite Circle
CSS Center
OOA&D Oval
SimUDuck Lake
Web Design Way
... другие названия станций ...

Booch Line
Ajax Rapids
HTML Heights
JavaBeans Boulevard
LSP Lane
Head First Labs
Objectville PizzaStore
UML Walk
Ajax Rapids

Gamma Line
OOA&D Oval
Head First Lounge
OOA&D Oval

Jacobson Line
Servlet Springs
... другие станции на линии...

... другие линии...
```

В этом упражнении используются всего пять методов, но большинство магнитов должно использоваться несколько раз.

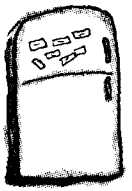
loadLine()

loadStations()

addStation()

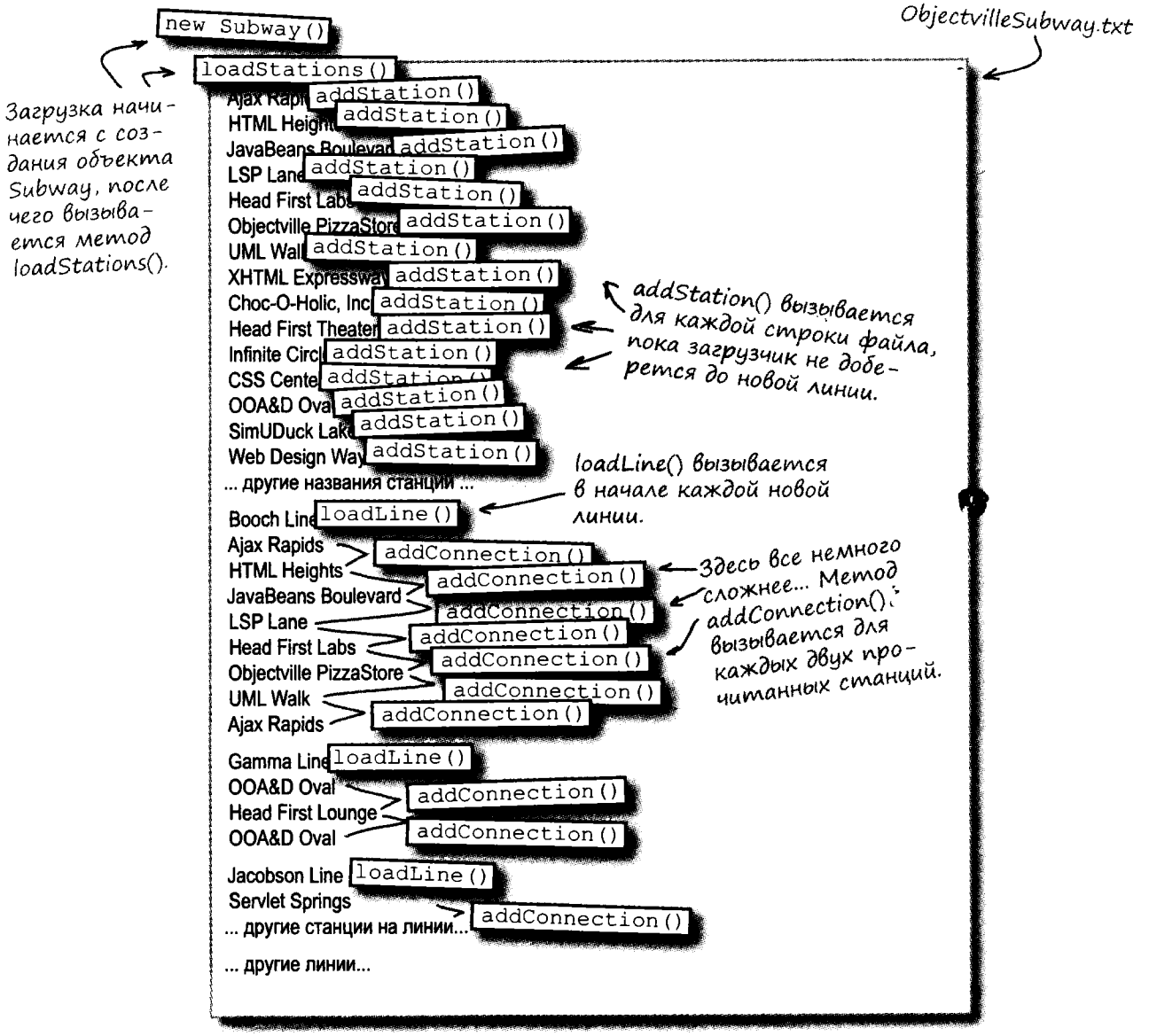
new Subway()

addConnection()



Развлечения с магнитами. Решение

Давайте посмотрим, что же происходит, когда мы вызываем метод loadFromFile() из SubwayLoader и передаем ему текстовый файл, полученный от турагентства. Вам было поручено расставить магниты, находящиеся в нижней части страницы, рядом со строками текстового файла, в которых будут вызываться соответствующие методы.



Головоломка

Мы почти завершили первый вариант использования и первую итерацию! Остается лишь протестировать наше решение и убедиться в том, что оно действительно работает.

Проблема:

Протестируйте процесс загрузки данных из файла **`ObjectvilleSubway.txt`** и убедитесь в том, что **`SubwayLoader`** правильно загружает все станции и соединения в файле.

Ваша задача:

1. Добавьте в **`Subway`** метод, проверяющий существование соединения по двум названиям станций и линии для этого соединения.
2. Напишите тестовый класс **`LoadTester`** с методом **`main()`**, который загружает данные сети метро по текстовому файлу, полученному нами от турагентства.
3. Включите в **`LoadTester`** код, который проверяет некоторые станции и соединения из текстового файла по объекту **`Subway`**, возвращенному методом **`loadFromFile()`** класса **`SubwayLoader`**. Проверьте не менее трех станций и трех соединений на трех разных линиях.
4. Запустите тестовую программу и убедитесь в том, что итерация 1 завершена.

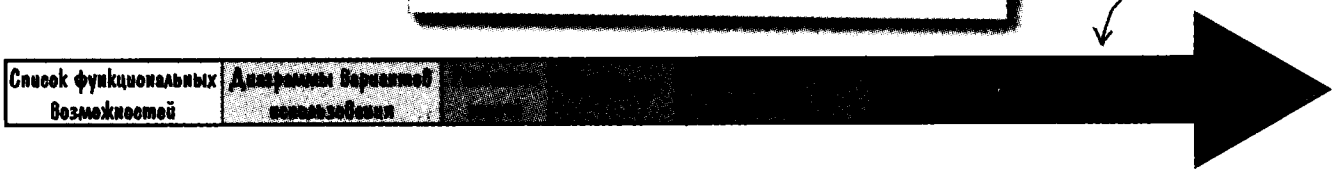
Вариант использования, который мы тестируем в этом упражнении.

Загрузка данных сети метро

Вариант использования

1. Администратор передает файл с информацией о станциях и линиях.
2. Система читает название станции.
3. Система проверяет, существует ли такая станция.
4. Система добавляет новую станцию в сеть.
5. Система повторяет шаги 2-4, пока не будут загружены все станции.
6. Система читает название линии.
7. Система читает названия двух соединенных станций.
8. Система проверяет, что эти станции существуют.
9. Система создает новое двустороннее соединение между двумя станциями на текущей линии.
10. Система повторяет шаги 7-9, пока линия не будет завершена.
11. Система повторяет шаги 6-10, пока не будут загружены данные всех линий.

Тестирование является частью фазы реализации. Код не может считаться завершенным, пока он не будет протестирован.



Головоломка. Решение

Вам было поручено протестировать класс **SubwayLoader** и наше представление сети и убедиться в том, что описание сети успешно загружается из текстового файла.

1. Добавьте в **Subway** метод, проверяющий существование соединения по двум названиям станций и линии для этого соединения.

```
public boolean hasConnection(String station1Name, String station2Name,
                             String lineName) {
  Station station1 = new Station(station1Name);
  Station station2 = new Station(station2Name);
  for (Iterator i = connections.iterator(); i.hasNext(); ) {
    Connection connection = (Connection)i.next();
    if (connection.getLineName().equalsIgnoreCase(lineName)) {
      if ((connection.getStation1().equals(station1)) &&
          (connection.getStation2().equals(station2))) {
        return true;
      }
    }
  }
  return false;
}
```

Ничего сложного... Код перебирает все соединения в сети и сравнивает названия линий и станций.

Subway

stations: Station [*]
connections: Connection [*]

addStation(String)
hasStation(String): boolean
addConnection(String, String, String)
hasConnection(String, String, String): boolean

Subway.java

Часто задаваемые вопросы

В: А не проще ли было бы написать метод hasConnection() с использованием объекта Line, как мы предлагали несколько страниц назад?

О: Проще. Если бы у нас был объект **Line**, то мы могли бы определить линию по названию, переданному **hasConnection()**, и просто перебрать объекты **Connection** для этой линии. Таким образом, в большинстве случаев при наличии объекта **Line** метод **hasConnection()** будет выполнять меньше работы и быстрее возвращать результат.

Тем не менее мы все равно решили не использовать объект **Line**, потому что метод **hasConnection()** был добавлен только для упрощения тестирования. Идея добавления объекта **Line** только для ускорения работы тестового метода выглядит сомнительно. Впрочем, если мы увидим, что метод **hasConnection()** найдет применение и в других частях приложения, то, возможно, к этой идее можно будет вернуться.

2. Напишите тестовый класс **LoadTester** с методом **main()** который загружает данные сети метро по текстовому файлу, полученному нами от турагентства.

3. Включите в **LoadTester** код, который проверяет некоторые станции и соединения из текстового файла по объекту **Subway**, возвращенному методом **loadFromFile()** класса **SubwayLoader**. Проверьте не менее трех станций и трех соединений на трех разных линиях.

Этот код просто загружает текстовый файл, а затем проверяет несколько станций и соединений.

```
public class LoadTester {
  public static void main(String[] args) {
    try {
      SubwayLoader loader = new SubwayLoader();
      Subway objectville =
        loader.loadFromFile(new File("ObjectvilleSubway.txt"));
      System.out.println("Testing stations...");
      if (objectville.hasStation("DRY Drive") &&
          objectville.hasStation("Weather-O-Rama, Inc.") &&
          objectville.hasStation("Boards 'R' Us")) {
        System.out.println("...station test passed successfully.");
      } else {
        System.out.println("...station test FAILED.");
        System.exit(-1);
      }

      System.out.println("\nTesting connections...");
      if (objectville.hasConnection("DRY Drive",
              "Head First Theater", "Meyer Line") &&
          objectville.hasConnection("Weather-O-Rama, Inc.",
              "XHTML Expressway", "Wirfs-Brock Line") &&
          objectville.hasConnection("Head First Theater",
              "Infinite Circle", "Rumbaugh Line")) {
        System.out.println("...connections test passed successfully.");
      } else {
        System.out.println("...connections test FAILED.");
        System.exit(-1);
      }
    } catch (Exception e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Здесь можно использовать любые станции и соединения.

LoadTester.java

Головоломка. Решение (окончание)

Вам было поручено протестировать класс **SubwayLoader** и наше представление сети и убедиться в том, что описание сети успешно загружается из текстового файла.

4. Запустите тестовую программу и убедитесь в том, что итерация 1 завершена.

```
File Edit Window Help NotVeryExciting
%java LoadTester
Testing stations...
...station test passed successfully.

Testing connections...
...connections test passed successfully.
```

Тесты обычно не вызывают особого интереса... Пока не поймешь один простой факт — они доказывают, что ваша программа РАБОТАЕТ!

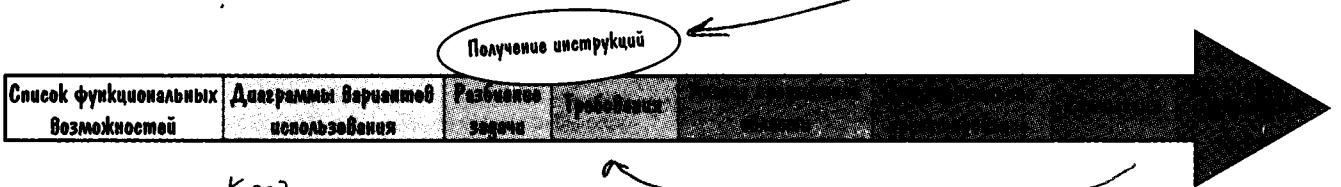
МОЗГОВОЙ ШТУРМ

Напишите тестовый вариант, который выводит все станции и соединения из объекта Subway, доказывая, что ваш код правильно загружает данные сети.

Эта часть работы завершена.

Пора переходить к следующей итерации

Наш тест доказывает, что первая итерация действительно завершена. Вариант использования «Загрузка данных сети метро» реализован, а это означает, что пришло время переходить к следующей итерации. На этот раз мы берем следующий вариант использования («Получение инструкций»), возвращаемся в фазу требований и прорабатываем этот вариант.

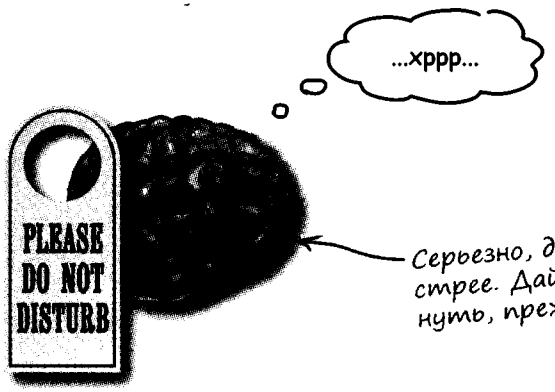


Когда итерация будет завершена, возьмите следующую функциональную возможность или вариант использования и снова начните с фазы требований.

Но прежде чем переходить к итерации 2...

Это была ДЛИННАЯ итерация, и мы основательно потрудились. ОСТАНОВИТЕСЬ, сделайте ПЕРЕРЫВ, что-нибудь съешьте или выпейте воды. Пусть ваш мозг немного ОТДОХНЕТ.

А затем, когда ваше дыхание восстановится, переверните страницу — мы приступаем к последнему варианту использования. Готовы? Тогда за дело.



Серьезно, дальше все пойдет еще быстрее. Дайте вашему мозгу отдохнуть, прежде чем двигаться дальше.

Что осталось сделать?

Мы далеко продвинулись как по вариантам использования, так и по списку функциональных возможностей. Ниже приведен список функциональных возможностей и диаграмма вариантов использования, построенная нами ранее в этой главе:

Objectville RouteFinder

Список функциональных возможностей

1. **Представление линий метро и станций на каждой линии.**
2. **Загрузка данных нескольких линий метро, в том числе и пересекающихся.**
3. **Определение действительного пути между любыми двумя станциями на любых линиях.**
4. **Вывод маршрута между двумя станциями в виде набора инструкций.**

Представление линий метро и станций на каждой линии.

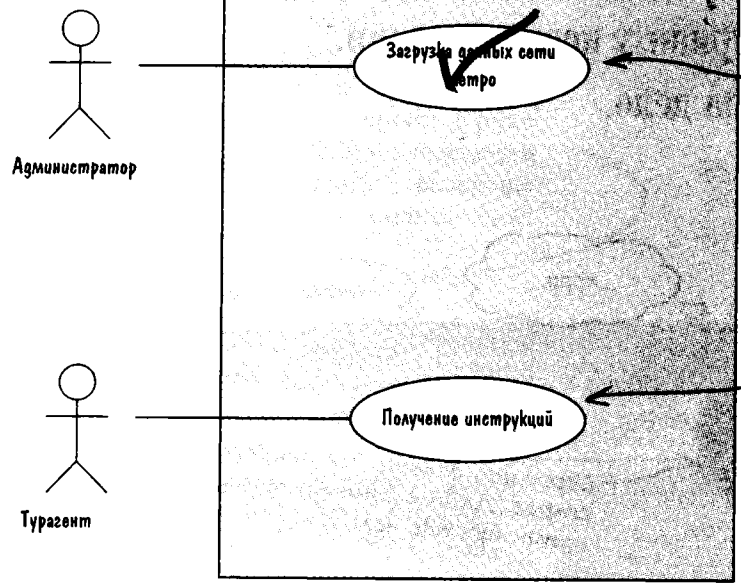
Загрузка данных нескольких линий метро.

Определение действительного пути между любыми двумя станциями на любых линиях.

Вывод маршрута между двумя станциями.

Обе эти функциональные возможности реализованы в итерации 1.

Вторая итерация должна реализовать две оставшиеся функциональные возможности.

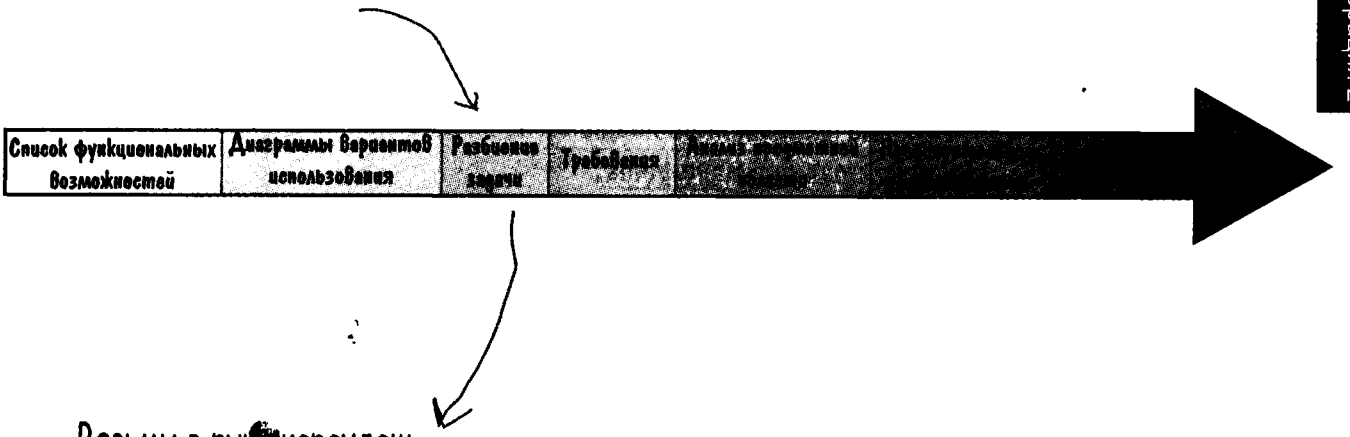


Также мы реализовали первый вариант использования.

Этому варианту использования будет посвящена вторая итерация.

Возвращаемся к фазе требований...

Итак, можно переходить к следующему варианту использования. Мы вернемся в фазу требований и проработаем новый вариант точно так же, как было сделано с первым. Начнем с названия варианта использования (диаграмма «Получение инструкций») и превратим его в полноценный вариант использования.



Возьми в руки карандаш

Напишите полный вариант использования для получения инструкций.

Мы снова возвращаемся к написанию вариантов использования. На этот раз вам предлагается написать вариант использования для получения инструкций по перемещению от одной станции к другой на метро Объектвиля.

Получение инструкций

Вариант использования

1. _____
2. _____
3. _____
4. _____

В этом варианте шагов будет меньше, чем в предыдущем.

Возьми в руку карандаш

Решение

Напишите полный вариант использования для получения инструкций.

Получение инструкций

Вариант использования

1. **Турагент передает системе начальную и конечную станции маршрута.**
2. **Система проверяет, что обе станции существуют в сети метро.**
3. **Система определяет маршрут от начальной до конечной станции.**
4. **Система выводит маршрут, если он был определен успешно.**

Как и в первом варианте, следует добавить проверку данных.

Непростая задача, которую нам предстоит решить в этой итерации.

Когда маршрут известен, вывести его будет несложно.

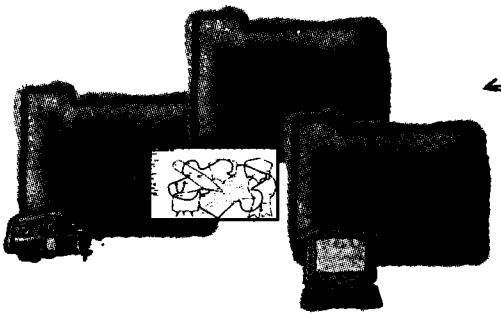
Два модуля, которые будут использоваться в реализации нашего варианта использования.

Я слегка запутался. Мы так старались разбить код на модули, но в наших вариантах использования задействовано несколько модулей. Почему мы прыгаем туда-сюда между модулями и вариантами использования?

Все внимание коду, потом все внимание заказчикам. Снова все внимание коду, потом снова заказчикам...

Когда мы начали разбивать приложение на модули на с. 520, речь шла о структуре приложения и о его логической организации. Классы **Subway** и **Station** находятся в модуле Subway, класс **SubwayLoader** — в модуле Loader, и так далее. Иначе говоря, мы сосредоточились на коде.

В действительности мы переключались между кодом и использованием системы в обеих итерациях.

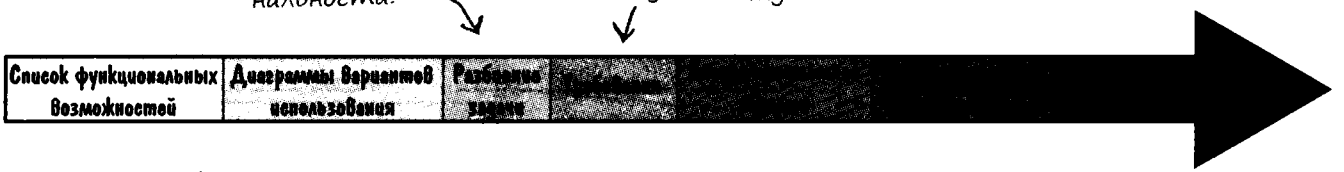


Отражает логическую организацию кода нашей системы.

Но в ходе работы над вариантами использования мы сосредоточились на том, как пользователь будет использовать систему, — мы рассмотрели формат входного файла и начали работать, ориентируясь на взаимодействия пользователя с системой. Так что в действительности мы постоянно переключались между кодом (на шаге «Разбиение задачи») и потребностями заказчика (на шаге «Требования»):

Этот шаг ориентирован на код и разбиение функциональности.

Этот шаг ориентирован на то, как заказчик будет использовать нашу систему.



Подобные переключения типичны для разработки ПО. Вы должны следить за тем, чтобы ваш продукт делал то, что положено, но в основе функциональности программы лежит написанный вами код.

Вы должны выдержать баланс между реализацией функциональности, необходимой заказчику, гибкостью и хорошей структурой кода.

Головоломка

Пришло время снова заняться анализом предметной области и проектированием системы. Возьмите следующий вариант использования, определите кандидатов для классов и операций, а затем отметьте все необходимые, по вашему мнению, изменения на диаграмме справа.

Упражнения кажутся знакомыми? Да, мы используем одни и те же приемы на каждой итерации цикла разработки.

Получение инструкций

Вариант использования

1. **Турагент передает системе начальную и конечную станции маршрута.**
2. **Система проверяет, что обе станции существуют в сети метро.**
3. **Система определяет маршрут от начальной до конечной станции.**
4. **Система выводит маршрут, если он был определен успешно.**

Существительные (кандидаты для классов):

_____ _____ _____

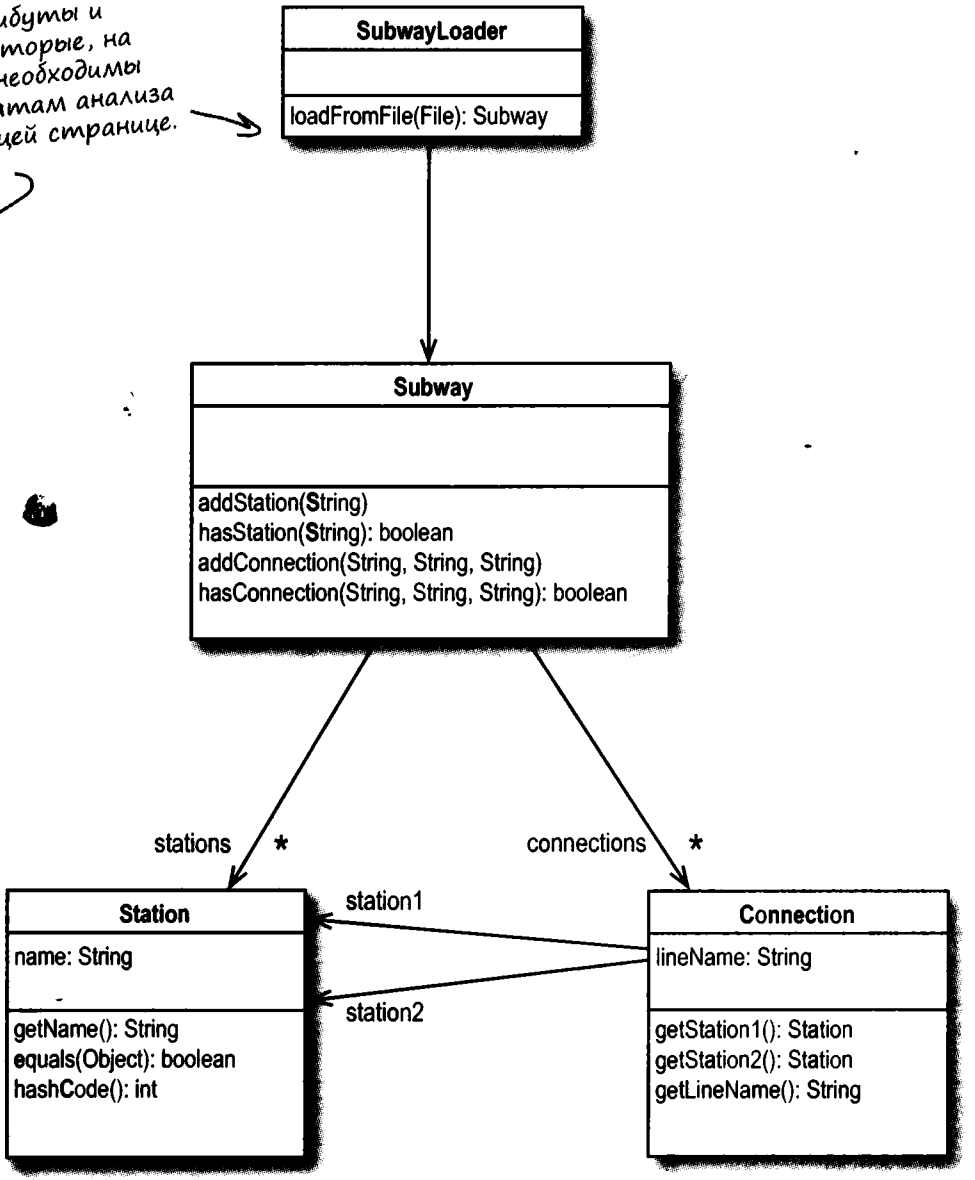
_____ _____ _____

Глаголы (кандидаты для операций):

_____ _____ _____

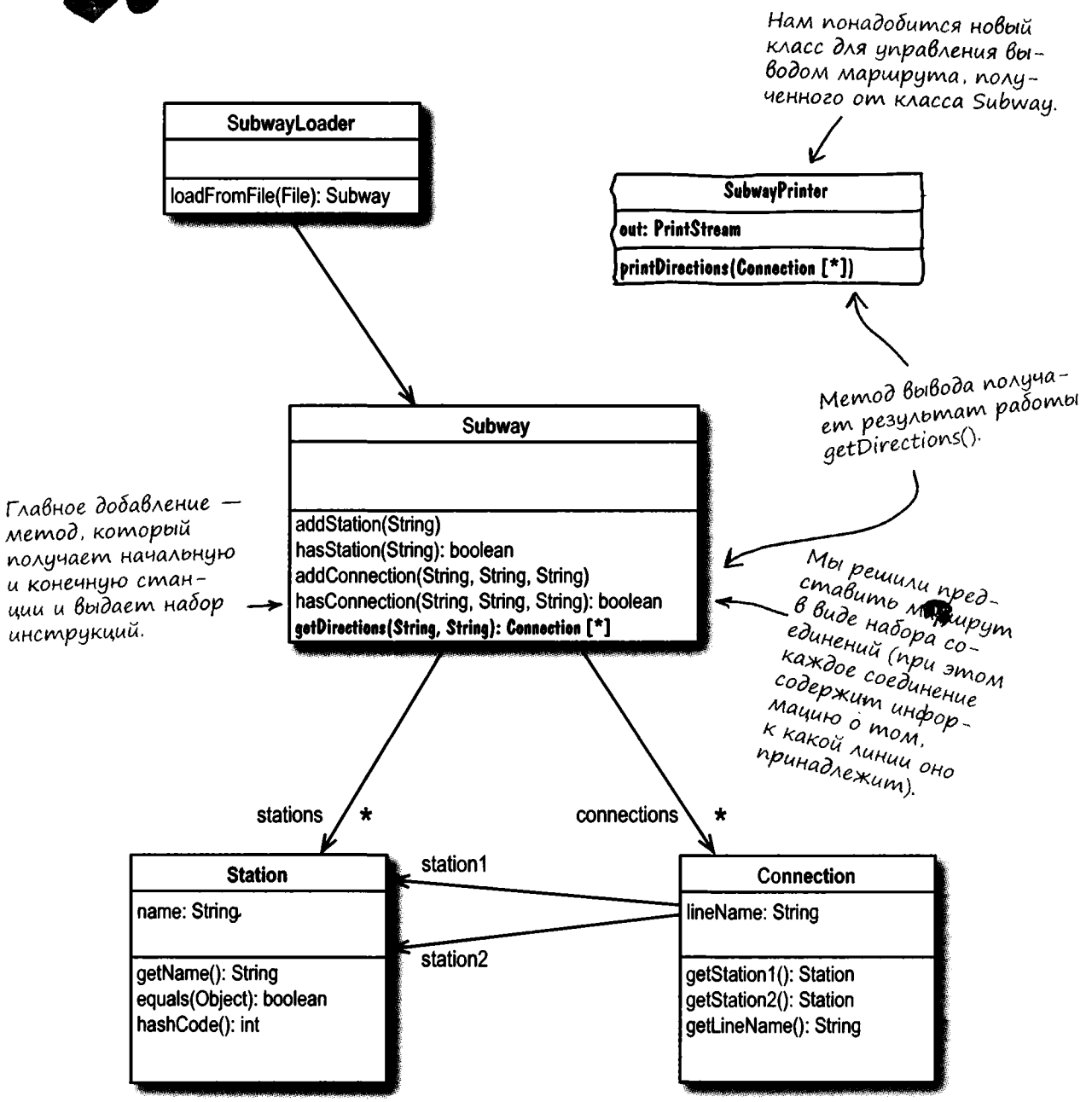
_____ _____ _____

Добавьте все новые классы, атрибуты и операции, которые, на ваш взгляд, необходимы по результатам анализа на предыдущей странице.



Мы снова одновременно занимаемся этими двумя фазами.

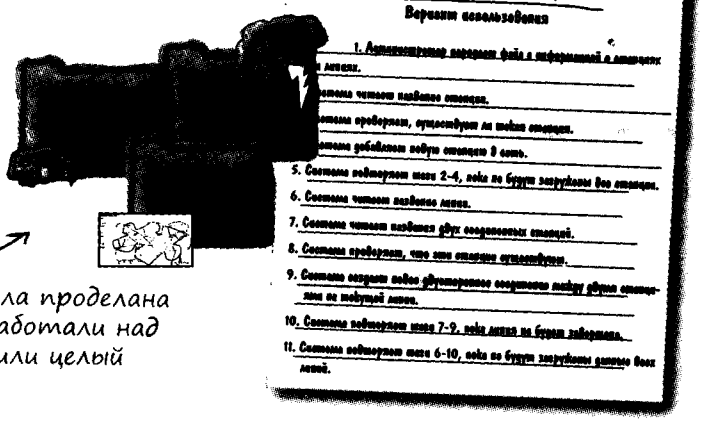
Головоломка. Решение



* Существительные и глаголы не приведены — к настоящему моменту вы уже достаточно хорошо освоили этот анализ и сможете самостоятельно преобразовать их в представленную диаграмму классов.

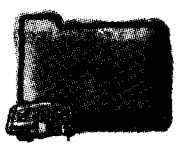
Итеративный метод упрощает решение

Диаграмма классов на предыдущей странице не так уж сильно отличается от диаграммы классов из первой итерации (см. с. 531). Это объясняется тем, что большая часть работы, проделанной в ходе первой итерации, относится ко всем итерациям.

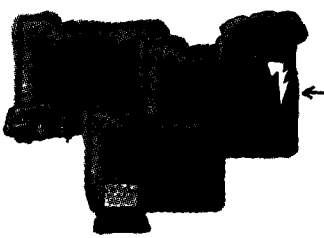


В ходе первой итерации была проделана БОЛЬШАЯ работа... Мы работали над тремя модулями и завершили целый вариант использования.

После завершения первой итерации будет намного проще, потому что большая часть выполненной работы относится и ко всем последующим итерациям.



Наша проработка модуля Subway заложила основу для второй итерации.



Нам еще многое нужно сделать, но основной путь был пройден в ходе первой итерации. Готова большая часть модуля Subway и даже написаны некоторые тесты.

Можно переходить к фазе реализации... Но эта итерация идет намного быстрее и требует меньшего объема работы, чем первая.

Реализация: Subway.java

Готово к употреблению

Задача вычисления маршрута между двумя станциями весьма нетривиальна. В частности, в ней приходится использовать теорию графов, о которых мы кратко упоминали на с. 533. Чтобы вам помочь, мы приводим готовый код определения маршрута.

Либо введите готовый код в том виде, в котором он приведен, либо загрузите полную версию Subway.java с сайта Head First Labs.

```
public class Subway {
  private List stations;
  private List connections;
  [redacted]

  public Subway() {
    this.stations = new LinkedList();
    this.connections = new LinkedList();
    [redacted]
  }

  // addStation(), hasStation() и hasConnection() остаются неизменными

  public Connection addConnection(String station1Name, String station2Name,
                                  String lineName) {
    if ((this.hasStation(station1Name)) &&
        (this.hasStation(station2Name))) {
      Station station1 = new Station(station1Name);
      Station station2 = new Station(station2Name);
      Connection connection = new Connection(station1, station2, lineName);
      connections.add(connection);
      connections.add(new Connection(station2, station1,
                                     connection.getLineName()));
      [redacted]
      return connection;
    } else {
      throw new RuntimeException("Invalid connection!");
    }
  }

  private void addToNetwork(Station station1, Station station2) {
   if (network.keySet().contains(station1)) {
      List connectingStations = (List)network.get(station1);
      if (!connectingStations.contains(station2)) {
        connectingStations.add(station2);
      }
    } else {
      List connectingStations = new LinkedList();
      connectingStations.add(station2);
```

Контейнеры для хранения станций и списков всех станций, с которыми они связаны.

При добавлении соединений необходимо обновить информацию о станциях и их соединениях с сетью метро.

С этого метода начинается новый код.

Ключами этой карты являются станции, а значениями — списки всех станций, с которыми они соединены (независимо от того, по какой линии проходит соединение).

```
    network.put(station1, connectingStations);
  }
}

public List getDirections(String startStationName,
                          String endStationName) {
  if (!this.hasStation(startStationName) ||
      !this.hasStation(endStationName)) {
    throw new RuntimeException(
      "Stations entered do not exist on this subway.");
  }

  Station start = new Station(startStationName);
  Station end = new Station(endStationName);
  List route = new LinkedList();
  List reachableStations = new LinkedList();
  Map previousStations = new HashMap();

  List neighbors = (List)network.get(start);
  for (Iterator i = neighbors.iterator(); i.hasNext(); ) {
    Station station = (Station)i.next();
    if (station.equals(end)) {
      route.add(getConnection(start, end));
      return route;
    } else {
      reachableStations.add(station);
      previousStations.put(station, start);
    }
  }

  List nextStations = new LinkedList();
  nextStations.addAll(neighbors);
  Station currentStation = start;

  searchLoop:
  for (int i=1; i<stations.size(); i++) {
    List tmpNextStations = new LinkedList();
    for (Iterator j = nextStations.iterator(); j.hasNext(); ) {
      Station station = (Station)j.next();
      reachableStations.add(station);
      currentStation = station;
      List currentNeighbors = (List)network.get(currentStation);
      for (Iterator k = currentNeighbors.iterator(); k.hasNext(); ) {
        Station neighbor = (Station)k.next();
        if (neighbor.equals(end)) {
          reachableStations.add(neighbor);
          previousStations.put(neighbor, currentStation);
          break searchLoop;
```

Проверка начальной и конечной станций, упоминаемых в вариантах использования на с. 552.

Метод основан на хорошо известном алгоритме Дейкстры, вычисляющем кратчайший путь между узлами графа.

Первая часть кода обрабатывает ситуацию, при которой конечная станция находится на расстоянии всего одного соединения от начальной.

Циклы перебирают все наборы станций, достижимых из начальной станции, пытаясь найти маршрут от начальной до конечной точки с минимальным количеством промежуточных станций.

```
        } else if (!reachableStations.contains(neighbor)) {
          reachableStations.add(neighbor);
          tmpNextStations.add(neighbor);
          previousStations.put(neighbor, currentStation);
        }
      }
    }
    nextStations = tmpNextStations;
  }

  // We've found the path by now
  boolean keepLooping = true;
  Station keyStation = end;
  Station station;

  while (keepLooping) {
    station = (Station)previousStations.get(keyStation);
    route.add(0, getConnection(station, keyStation));
    if (start.equals(station)) {
      keepLooping = false;
    }
    keyStation = station;
  }

  return route;
}

private Connection getConnection(Station station1, Station station2) {
  for (Iterator i = connections.iterator(); i.hasNext(); ) {
    Connection connection = (Connection)i.next();
    Station one = connection.getStation1();
    Station two = connection.getStation2();
    if ((station1.equals(one)) && (station2.equals(two))) {
      return connection;
    }
  }
  return null;
}
}
```

Готово к употреблению

Получив путь, мы просто «раскручиваем» его и создаем список соединений для перехода от начальной станции к конечной.

Вспомогательный метод получает две станции и ищет соединение между ними (на любой линии).

Да это просто смешно. 500 страниц я читаю о том, какая замечательная штука ООАП, и вы мне просто выдаете самый сложный код в книге? Я думала, что к этому моменту уже смогу писать хорошие программы самостоятельно.

Иногда лучший способ решения задачи — найти кого-то, кто уже выполнил эту работу за вас.

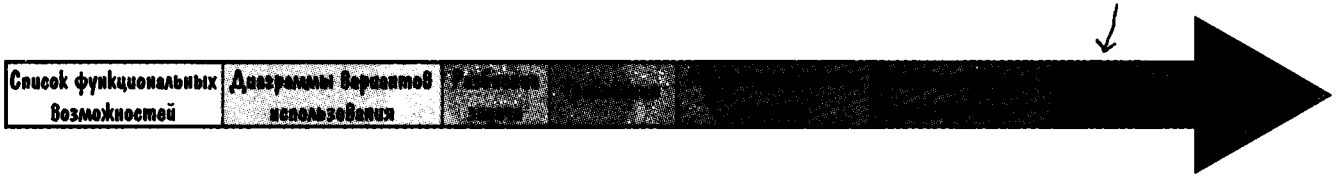
Действительно выглядит странно — мы даем вам готовый код определения маршрута между двумя станциями. Но это один из аспектов мастерства разработчика: готовность воспользоваться существующими решениями сложных задач.

Знакомый студент помог нам реализовать версию алгоритма Дейкстры, которая работает с сетью метро (серьезно!) Конечно, вы можете разработать собственное, полностью оригинальное решение для каждой задачи, но зачем, если кто-то уже выполнил всю работу за вас?

Спасибо Феликсу Геллеру — его код нас сильно выручил.

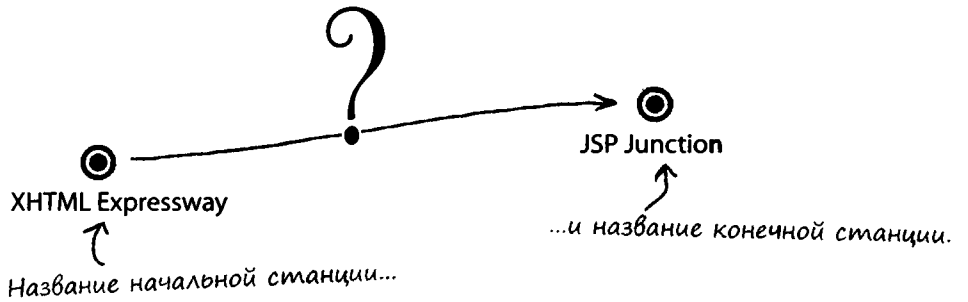
Иногда оказывается, что лучший код для решения конкретной задачи уже был написан. Не старайтесь сделать все самостоятельно, если у кого-то уже есть рабочее решение.

Вы находитесь ЗДЕСЬ.

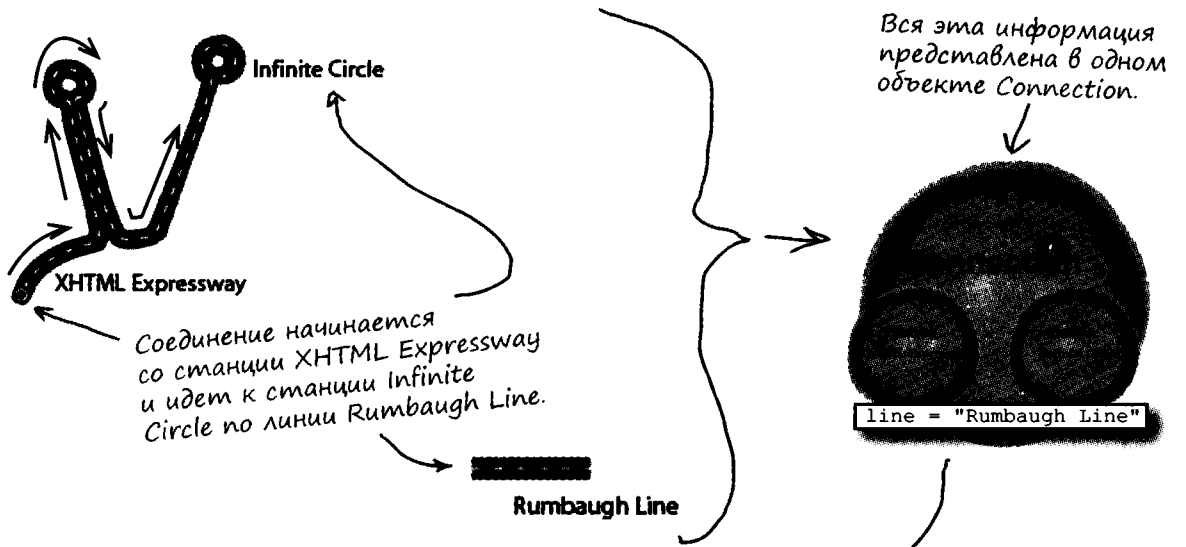


Как выглядит маршрут?

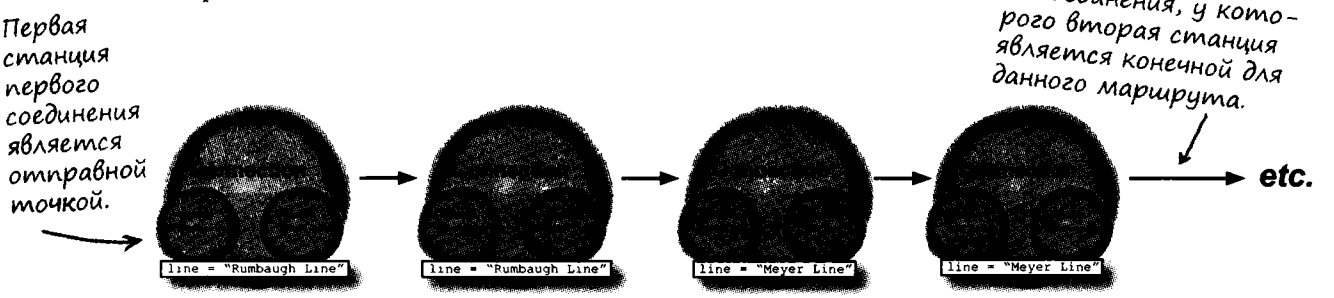
Метод **getDirections()**, добавленный нами в класс **Subway**, получает два объекта **String**: название начальной станции и название конечной станции, на которую хочет попасть турист:



Метод **getDirections()** возвращает контейнер **List**, заполненный объектами **Connection**. Каждый объект **Connection** является частью пути между двумя станциями:



Таким образом, весь маршрут, возвращаемый **getDirections()**, выглядит как серия объектов **Connection**:



Головоломка

Работа почти закончена! С работоспособным методом **getDirections()** класса **Subway** осталось только реализовать вывод инструкций. Напишите класс с именем **SubwayPrinter**, который получает структуру данных, возвращаемую **getDirections()** (список объектов **Connection**), и выводит инструкции. Вывод должен осуществляться в поток **OutputStream**, переданный конструктору класса **SubwayPrinter** при создании экземпляра SubwayPrinter.

Используйте следующую диаграмму классов **SubwayPrinter**:

SubwayPrinter
out:PrintStream
printDirections(Connection [*])

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: Также напишите класс SubwayTester, который тестирует загрузку сети метро Объективиля, а затем выводит инструкции.

Сперва выводится начальная станция.

Вывод должен выглядеть примерно так:

Линия определяется вызовом getConnection. Выведите название линии и станцию, к которой следует двигаться.

Выведите все станции на проходимой линии.

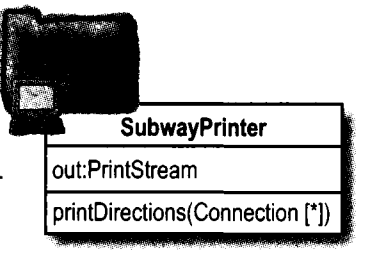
При переходе на новую линию выводите название линии и станцию, на которой следует выйти...

...а также новую линию, на которую нужно пересесть.

```
File Edit Window Help ShowMeTheWay
%java SubwayTester "XHTML Expressway" "JSP Junction"
Start out at XHTML Expressway.
Get on the Rumbaugh Line heading towards Infinite Circle.
  Continue past Infinite Circle...
When you get to Head First Theater, get off the Rumbaugh Line.
Switch over to the Meyer Line, heading towards DRY Drive.
  Continue past DRY Drive...
When you get to Web Design Way, get off the Meyer Line.
Switch over to the Wirfs-Brock Line, heading towards Boards 'R' Us.
  Continue past Boards 'R' Us...
When you get to EJB Estates, get off the Wirfs-Brock Line.
Switch over to the Liskov Line, heading towards Design Patterns Plaza.
  Continue past Design Patterns Plaza...
Get off at JSP Junction and enjoy yourself!
```

Последним должен выводиться объект Connection, ведущий к конечной станции маршрута.

Головоломка. Решение



Ниже приведена наша реализация класса **SubwayPrinter**. Ваша реализация перебора контейнера маршрутов может несколько отличаться, но для получения высшей оценки результат должен полностью совпадать с приведенным.

```
public class SubwayPrinter {
  private PrintStream out;

  public SubwayPrinter(OutputStream out) {
    this.out = new PrintStream(out);
  }

  public void printDirections(List route) {
    Connection connection = (Connection)route.get(0);
    String currentLine = connection.getLineName();
    String previousLine = currentLine;
    out.println("Start out at " +
      connection.getStation1().getName() + ".");
    out.println("Get on the " + currentLine + " heading towards " +
      connection.getStation2().getName() + ".");
    for (int i=1; i<route.size(); i++) {
      connection = (Connection)route.get(i);
      currentLine = connection.getLineName();
      if (currentLine.equals(previousLine)) {
        out.println("  Continue past " +
          connection.getStation1().getName() + "...");
      } else {
        out.println("When you get to " +
          connection.getStation1().getName() + ", get off the " +
          previousLine + ".");
        out.println("Switch over to the " + currentLine +
          ", heading towards " + connection.getStation2().getName() + ".");
        previousLine = currentLine;
      }
    }
    out.println("Get off at " + connection.getStation2().getName() +
      " and enjoy yourself!");
  }
}
```

Вместо прямого вывода в System.out наш класс при конструировании получает объект OutputStream. Это позволяет направить инструкции в любой выходной поток, а не только в консольное окно на экране пользователя.

Начинаем с вывода начальной станции...

...и первой линии, а также следующей станции в маршруте.

Проверяем текущее соединение и определяем, требуется ли пересадка на другую линию.

Если линия осталась прежней, просто выводим название станции.

Если линия изменилась, выводим описание пересадки.

Весь маршрут пройден... Можно выходить из метро.

И последний тестовый класс...

Осталось лишь свести все воедино. Приведенный ниже класс **SubwayTester** загружает данные сети метро Объективиля, получает две станции из командной строки и выводит инструкции по перемещению между этими двумя станциями с использованием нового метода **getDirections()** и класса SubwayPrinter.

Чтобы доказать, что программа работает, нам понадобятся тестовые примеры и тестовые классы.

```
public class SubwayTester {
  public static void main(String[] args) {
    if (args.length != 2) {
      System.err.println("Usage: SubwayTester [startStation] [endStation]");
      System.exit(-1);
    }
    try {
      SubwayLoader loader = new SubwayLoader();
      Subway objectville =
        loader.loadFromFile(new File("ObjectvilleSubway.txt"));

      if (!objectville.hasStation(args[0])) {
        System.err.println(args[0] + " is not a station in Objectville.");
        System.exit(-1);
      } else if (!objectville.hasStation(args[1])) {
        System.err.println(args[1] + " is not a station in Objectville.");
        System.exit(-1);
      }

      List route = objectville.getDirections(args[0], args[1]);
      SubwayPrinter printer = new SubwayPrinter(System.out);
      printer.printDirections(route);
    } catch (Exception e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Для этого теста в командной строке передаются две станции.

После предшествующего тестирования мы уже знаем, что загрузка сети метро работает нормально.

Мы также проверяем, что обе указанные станции существуют в сети метро.

Имея две проверенные станции, мы можем проложить между ними маршрут...

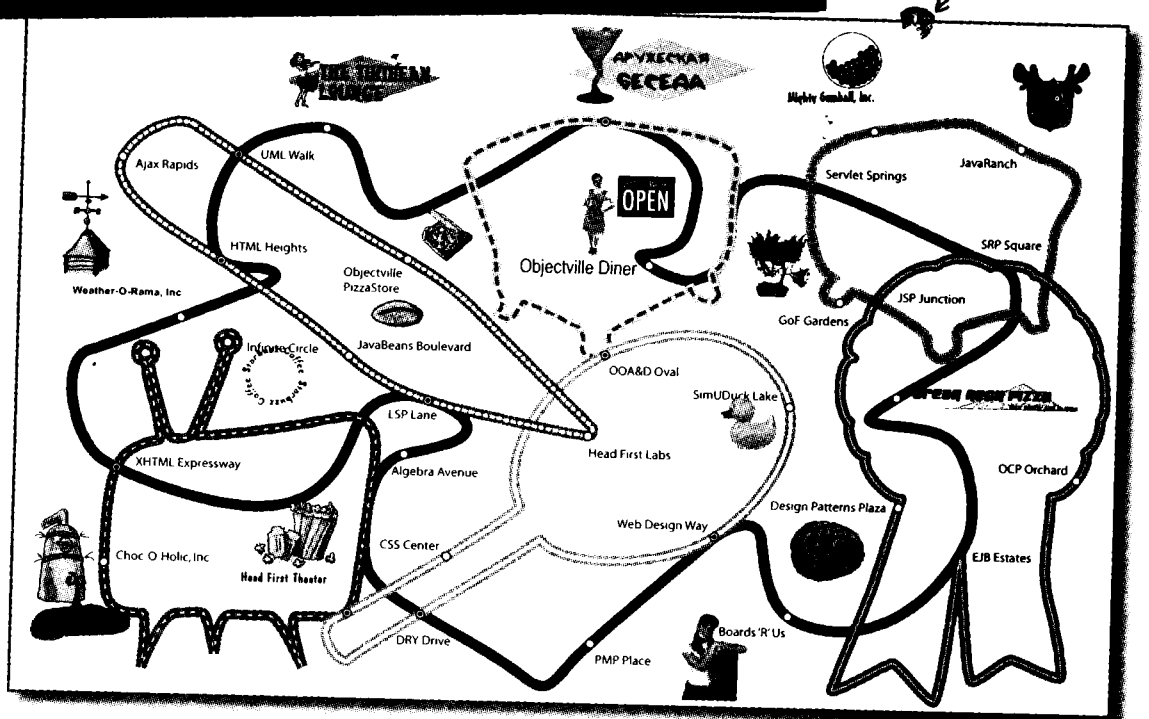
...и использовать новый класс SubwayPrinter для вывода маршрута.

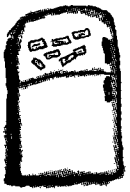
Увидеть Объектвиль собственными глазами

Пришло время откинуться на спинку кресла и насладиться плодами своих трудов. Откомпилируйте все классы приложения и протестируйте SubwayTester с несколькими разными вариантами начальной и конечной станции. Один из наших любимых маршрутов:

```
File Edit Window Help CandyIsKing
%java SubwayTester "Mighty Gumball, Inc." "Choc-O-Holic, Inc."
Start out at Mighty Gumball, Inc..
Get on the Jacobson Line heading towards Servlet Springs.
When you get to Servlet Springs, get off the Jacobson Line.
Switch over to the Wirfs-Brock Line, heading towards Objectville Diner.
  Continue past Objectville Diner...
When you get to Head First Lounge, get off the Wirfs-Brock Line.
Switch over to the Gamma Line, heading towards OOA&D Oval. .
When you get to OOA&D Oval, get off the Gamma Line.
Switch over to the Meyer Line, heading towards CSS Center.
  Continue past CSS Center...
When you get to Head First Theater, get off the Meyer Line.
Switch over to the Rumbaugh Line, heading towards Choc-O-Holic, Inc..
Get off at Choc-O-Holic, Inc. and enjoy yourself!
```

А куда Вы хотите отправиться сегодня?

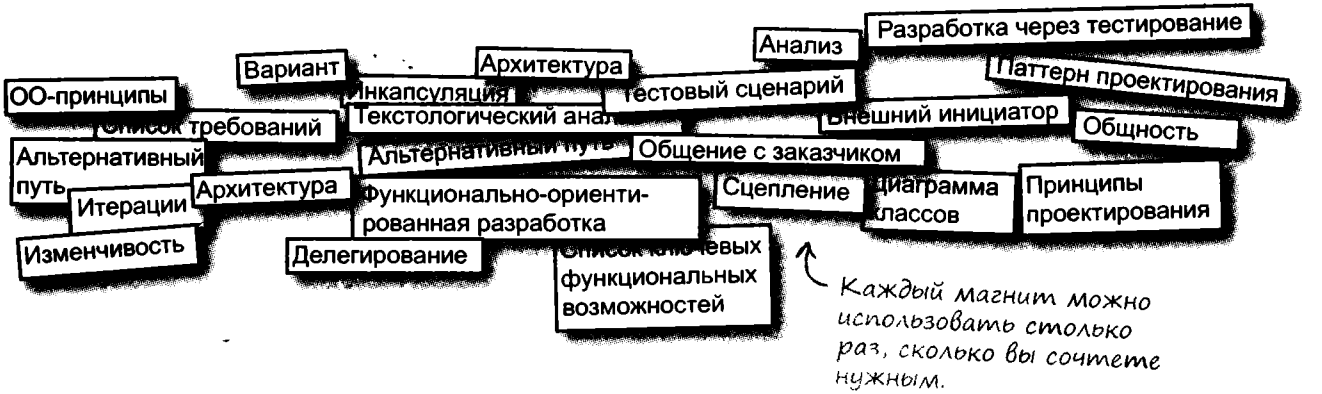




Развлечения с магнитами

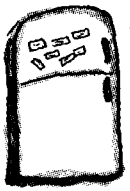
Когда-то давно, на с. 505, мы предложили вам подумать над тем, какое место занимают различные методы и приемы, изученные нами, в жизненном цикле ООАП. Теперь, после написания еще одной хорошей программы, вы готовы повторить это упражнение и сравнить свои ответы с нашими. Не изменится ли расположение некоторых магнитов при повторном выполнении?

Да, и не забудьте: в каждой фазе можно разместить несколько магнитов, а некоторые магниты могут использоваться многократно.



Каждый магнит можно использовать столько раз, сколько вы сочтете нужным.

На этот раз ответы приведены на следующей странице!



Развлечения с магнитами. Решение

Вам было предложено расставить магниты в правильных фазах жизненного цикла ООАП, причем в каждой фазе можно было разместить несколько магнитов. Отличаются ли ваши ответы от наших?

Часто задаваемые вопросы

В: Похоже, почти все магниты можно разместить в каждой фазе... Но ведь это не может быть правильно?

О: Это абсолютно нормально. Хотя в хорошем цикле разработки безусловно присутствуют базовые фазы, большая часть того, что вы узнали об ООАП, ОО-принципах, проектировании, анализе, требованиях и т. д., может применяться почти на каждой стадии разработки.

Самый эффективный и успешный путь написания хороших программ — владение максимально широким набором инструментов с возможностью выбора одного (или многих) из них на каждой стадии цикла разработки. Чем больше у вас инструментов, тем больше вариантов анализа и работы над задачей... а значит, меньше шансов запутаться и не знать, что делать дальше.

Цель ООАП — обилие возможностей. У задачи никогда не существует единственно правильного решения, поэтому чем больше у вас возможностей, тем выше вероятность того, что вам удастся найти хорошее решение.

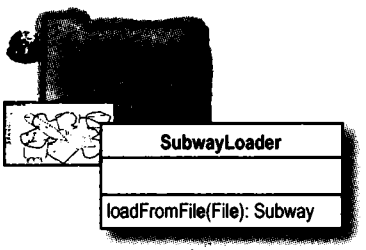
Нужна ли итерация № 3?

Нет, мы не будем искать себе новые задачи проектированяи. Но вы должны понимать, что для совершенствования структуры приложения RouteFinder можно сделать очень многое. Мы предложим несколько идей на случай, если вам не терпится сделать еще один проход по жизненному циклу ООАП.

Улучшите расширяемость загрузки

В настоящий момент у нас имеется один класс, выполняющий загрузку, и он получает в качестве входных данных только объект Java **File**. Попробуйте разработать решение, которое позволит вам загружать данные сети метро из разных источников (начните с **File** и **InputStream**). Также упростите возможность добавления новых источников, например баз данных. Помните: при добавлении новой функциональности следует свести к минимуму изменения в существующем коде; возможно, вам стоит пойти по пути определения интерфейса или абстрактного базового класса.

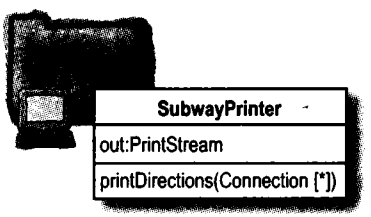
Хорошие программы, построенные с использованием ООАП, динамичны... Вы всегда можете добавить новую функциональность или улучшить структуру кода.



Поддержка других источников (и форматов!) вывода

Наша программа может выводить инструкции только в файл, при этом инструкции всегда форматируются строго определенным образом. Посмотрите, удастся ли вам спроектировать гибкий модуль вывода, позволяющий выдавать маршруты в разные источники (например, **File**, **OutputStream** и **Writer**) в разных форматах (например, подробный формат, совпадающий с уже использованным ранее, компактная форма, содержащая только инструкции по пересадкам, и форма XML для использования другими программами).

Подсказка: возможно, паттерн «Стратегия» из книги «Паттерны проектирования» пригодится вам в решении этой задачи.



А теперь примените ООАП в своих собственных проектах!

Мы прекрасно провели время в Объективиле, и расставаться будет немного грустно. Но для вас начинается самое интересное: возьмите то, что вы узнали, и используйте в своих проектах. Впрочем, в конце книги вас еще ждет пара приложений, а потом наступит время применять полученные знания на практике. Нам очень интересно, что из этого выйдет, — оставьте свои комментарии на сайте Head First Labs (http://www.headfirstlabs.com) и расскажите, что ООАП дало лично вам.

Десять главных тем (не рассмотренных в книге)

Хотите верьте, хотите нет, но это еще не все. Да, позади осталось более 550 страниц, и кое-что в них не уместилось. И хотя каждая из последних десяти тем заслуживают разве что краткого упоминания, мы решили выдать вам на дорогу чуть больше информации по каждой из них. Теперь вам будет о чем поговорить во время рекламных пауз... да и кто не любит время от времени поговорить об ООАП?

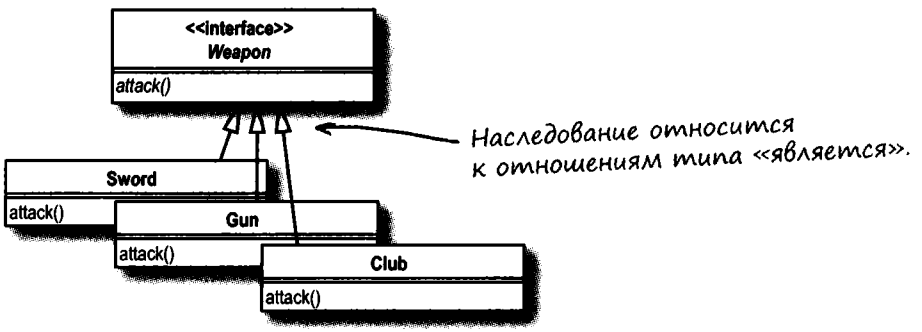
А когда закончите читать это приложение, останется еще одно... И немного рекламы... А потом — действительно всё. Честное слово!

1. Отношения «является» и «содержит»

В кругах ОО-программирования часто приходится слышать об отношениях «является» и «содержит».

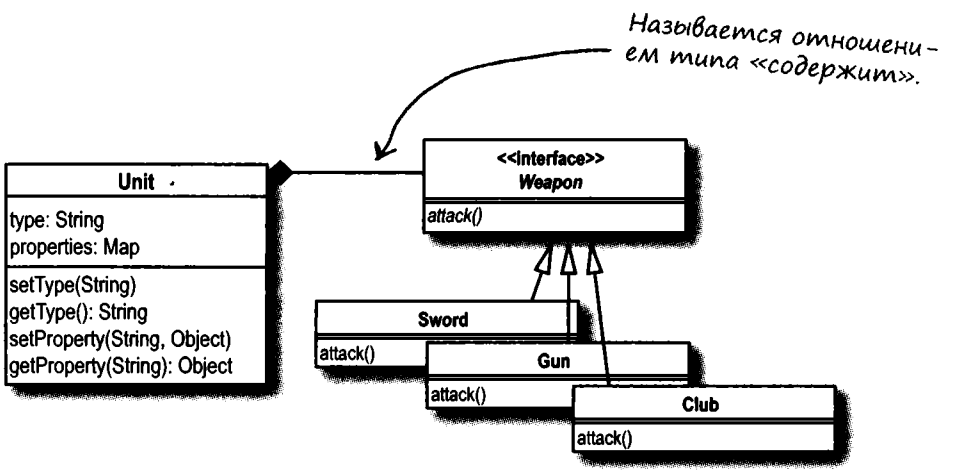
«Является» — наследование

Обычно отношения типа «является» относятся к наследованию, например: «Меч ЯВЛЯЕТСЯ оружием, следовательно, класс Sword должен расширять класс Weapon».



Отношения «содержит» относятся к композиции и агрегированию

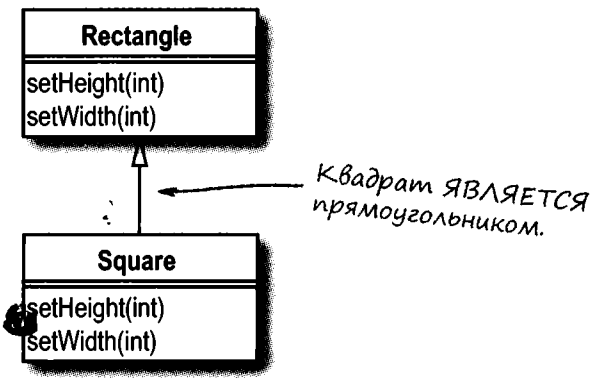
Под отношением «содержит» подразумевается композиция и агрегирование, поэтому используются выражения типа «Военная техника СОДЕРЖИТ оружие, поэтому в Unit следует включить объект Weapon».



Проблемы с «содержит» и «является»

Мы не стали рассматривать отношения «содержит» и «является» в книге, потому что в некоторых ситуациях они не работают. Для примера возьмем классы моделирования геометрических фигур, такие как **Circle**, **Rectangle** и **Diamond**.

Квадрат является частным случаем прямоугольника, поэтому к ним можно применить отношение «является»: квадрат является прямоугольником. Значит, класс **Square** должен расширять **Rectangle**, верно?



Но вспомните принцип LSP: замена базового типа субтипом не должна создавать проблем в программе. Таким образом, класс **Square** должен быть прямой заменой для **Rectangle**. Но что произойдет в следующем коде?

```
Rectangle square = new Square();
square.setHeight(10);
square.setWidth(5);

System.out.println("Height is " + square.getHeight());
```

Что вернет этот метод? И что еще важнее, что он ДОЛЖЕН вернуть?

При задании ширины в методе **setWidth()** для **Square** также задается и высота, поскольку квадраты имеют одинаковую ширину и высоту. Таким образом, хотя квадрат ЯВЛЯЕТСЯ прямоугольником, он не ведет себя как прямоугольник. Приведенный выше вызов **getHeight()** вернет 5, а не 10, — это означает, что поведение квадратов отличается от поведения прямоугольников и квадраты не могут использоваться в качестве их замены, а это нарушение принципа LSP.

Наследование применяется тогда, когда один объект ведет себя как другой объект, а не только при наличии отношений типа «является».

2. Форматы вариантов использования

Несмотря на наличие вполне стандартного определения для того, *что* представляет собой вариант использования, не существует стандартного способа записи вариантов – известно несколько разных способов записи:

Собачья дверь для Тодда и Джины, версия 2.0
Как должна работать дверь

1. Фидо лает, чтобы его выпустили.
2. Тодд или Джина слышат лай.
3. Тодд или Джина нажимают knopky на пульте.
4. Дверь открывается.
5. Фидо выходит на улицу.
6. Фидо делает свои дела.
   6.1 Дверь автоматически закрывается.
   6.2 Фидо лает, чтобы вернуться
   6.3 Тодд или Джина слышат лай (
   6.4 Тодд или Джина нажимают k
   пульте.
   6.5 Дверь открывается (повто
7. Фидо заходит в дом.
8. Дверь автоматически закрывает

Формат, который мы использовали до настоящего момента. Это простой пошаговый формат, который хорошо подходит для большинства ситуаций.

Вариант использования в неформальном стиле. Действия, которые должны выполняться системой, записаны в виде текста.

Собачья дверь для Тодда и Джины, версия 2.0
Как должна работать дверь

Фидо лает, чтобы его выпустили. Услышав его лай, Тодд или Джина нажимают knopky на пульте. Кнопка открывает собачью дверь, Фидо выходит наружу. Там он делает свои дела и возвращается обратно. Дверь при этом закрывается автоматически.

Если Фидо слишком долго остается снаружи, то дверь закроется. Фидо лает, чтобы вернуться в дом, на что Тодд или Джина снова нажимают knopky на пульте. Собачья дверь открывается, и Фидо снова заходит в дом.

Все альтернативные пути обычно добавляются в конец текста в форме «если то».

Ориентированность на взаимодействия

Этот формат в основном стремится выделить сущность системы и способы взаимодействия субъектов, находящихся за пределами системы, с вашей программой.

Этот формат ориентирован на выделение внешних факторов (субъекты) и того, что делает сама система.

Собачья дверь для Тодда и Джины, версия 2.0

Как должна работать дверь

Вспомните: субъекты существуют вне системы. Они выполняют операции с системой или используют ее.

Субъект	Система
Фидо лает, чтобы его выпустили.	
Тодд или Джина слышат лай.	
Тодд или Джина нажимают кнопку на пульте.	
	Дверь открывается.
Фидо выходит на улицу.	
Фидо делает свои дела.	
Фидо возвращается в дом.	
	Дверь автоматически закрывается.

В случае с дверью Тодда и Джины система относительно проста, а ее работа в основном сводится к реакции на действия Тодда, Джины или Фидо.

Расширения

Если Фидо остается снаружи, а дверь закрывается до его возвращения, Фидо может залаять, чтобы его впустили. Тодд или Джина снова нажимают кнопку на пульте, и Фидо возвращается в дом.

Этот формат не предоставляет удобных средств определения альтернативных путей, которые просто добавляются в конце варианта.

Формализованный вариант использования

Аналог используемого нами формата, но с дополнительными подробностями.

Собачья дверь для Тодда и Джины, версия 2.0

Как должна работать дверь

Субъекты — внешние факторы, влияющие на работу системы.

Первичный субъект: Фидо

Вторичные субъекты: Тодд и Джина

Предусловие: Фидо находится в доме и хочет выйти наружу.

В разделе предусловий подробно описаны все допущения относительно системы.

Цель: Фидо сделал свои дела и хочет вернуться в дом, при этом Тодду и Джине не приходится вставать и открывать/закрывать дверь.

Вариант использования направлен на достижение этой цели.

Основной путь

1. Фидо лает, чтобы его выпустили.
2. Тодд или Джина слышат лай Фидо.
3. Тодд или Джина нажимают кнопку на пульте.
4. Дверь открывается.
5. Фидо выходит на улицу.
6. Фидо делает свои дела.
7. Фидо заходит в дом.
8. Дверь автоматически закрывается.

Расширения

Альтернативные пути также называются расширениями. В этом формате они перечисляются под описанием шагов основного пути.

6.1 Дверь автоматически закрывается.

6.2 Фидо лает, чтобы вернуться в дом.

6.3 Тодд или Джина слышат лай (повторно).

6.4 Тодд или Джина нажимают кнопку на пульте.

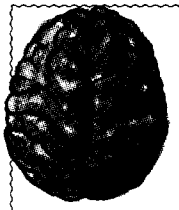
6.5 Дверь открывается (повторно).

Шаги альтернативного пути перечисляются отдельно.

При этом все варианты использования говорят одно и то же... Вы сами (и вероятно, ваш начальник) должны определить, какой формат лучше подходит для ваших целей.

3. Антипаттерны

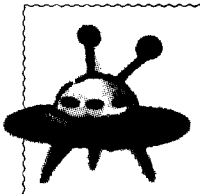
В этой книге мы достаточно много говорили о паттернах проектирования:



Паттерны проектирования

Паттерны проектирования — это проверенные решения конкретных задач. Применение паттернов способствует формированию структуры приложения, упрощающей понимание и сопровождение кода, а также улучшающей гибкость.

Но существует и другая разновидность паттернов, о которой тоже следует знать, — так называемые антипаттерны:



Антипаттерны

Антипаттерны являются противоположностью паттеров проектирования: это распространенные ПЛОХИЕ решения. Старайтесь своевременно распознавать ловушки и избегать их.

Антипаттерны возникают тогда, когда некоторая задача часто решается одним способом, но этот способ оказывается ПЛОХИМ. Например, под одним из типичных антипаттернов — «Газовая фабрика» — понимается излишне сложная структура кода, которая усложняет его сопровождение. Соответственно вам стоит избегать паттерна «Газовая фабрика» в своем коде.

Абсолютно серьезно! Не забудьте на следующей встрече разработчиков упомянуть о том, как вы старались избежать антипаттерна «Газовая фабрика».

Паттерны проектирования помогают узнавать и реализовывать ХОРОШИЕ решения стандартных задач.

Антипаттерны помогают узнавать ПЛОХИЕ решения стандартных задач и избегать их.

4. Карты CRC

Сокращение CRC означает «класс, обязанность, сотрудник» (Class, Responsibility, Collaborator). По карте CRC можно быстро определить, какие обязанности выполняет класс и с какими другими классами он взаимодействует.

Обычно каждая карта CRC представляет отдельный класс. Она состоит из двух столбцов: в одном столбце перечисляются обязанности класса, а в другом — классы, с которыми он взаимодействует для выполнения этих обязанностей.

Тот класс, для которого определяются обязанности.

Класс: BarkRecognizer

Описание: класс определяет интерфейс к оборудованию распознавания лая.

Обязанности:

Название	Сотрудник
Приказать двери открыться.	DogDoor

Список всех обязанностей класса.

Если в выполнении задачи участвуют другие классы, перечислите их в этом столбце.

Класс: DogDoor

Описание: представляет физическую собачью дверь. Предоставляет интерфейс к оборудованию, которое управляет дверью.

Обязанности:

Название	Сотрудник
Открыть дверь	
Закрыть дверь	

Здесь записываются как операции, выполняемые классом самостоятельно, так и операции, в выполнении которых участвуют другие классы.

У этих операций классов-сотрудников нет.

Карты CRC помогают в реализации принципа SRP

Карты CRC помогают убедиться в том, что ваши классы следуют принципу единственной обязанности (SRP). Они также тесно связаны с процессом анализа SRP:

Анализ SRP для класса Automobile

Автомобиль	запускается
Автомобиль	останавливается
Автомобиль	меняет шины
Автомобиль	едет
Автомобиль	моется
Автомобиль	проверяет уровень масла
Автомобиль	получает информацию о уровне масла

Если какой-либо из ваших классов нарушает принцип SRP, вы можете воспользоваться картами CRC, чтобы определить, что именно должны делать те или иные классы в системе.

Вероятно, выполнение этой операции не является обязанностью данного класса.

Класс: Automobile

Описание: класс представляет автомобиль и его функциональность

Обязанности:

Название	Сотрудник
Запускается.	
Останавливается.	
Меняет шины.	Механик, шины
Едет.	Водитель
Моется.	Автомойка, служащие
Проверяет уровень масла.	Механик
Сообщает уровень масла.	

5. Метрики

Иногда бывает трудно оценить качество архитектуры — слишком силен фактор субъективности. В таких случаях вам могут помочь метрики. Хотя они не дают полной картины системы, по ним иногда можно оценить сильные и слабые стороны, а также найти потенциальные проблемы. Обычно специальные программы получают в качестве входных данных исходный код ваших классов и вычисляют метрики на основании кода и его структуры.

Эти метрики — важнее, чем обычные числа. Например, простой подсчет строк кода в приложении почти наверняка окажется напрасной тратой времени. Полученное значение — обычное число, не имеющее контекста (которое к тому же сильно зависит от ваших пристрастий в написании кода — об этом мы еще поговорим позднее в данном приложении). Но если подсчитать количество дефектов на 1000 строк кода, то результат превратится во вполне полезную метрику.

$$\textbf{плотность дефектов} = \frac{\text{обнаружено ошибок в коде}}{\text{всего строк кода / 1000}}$$

Число дает представление о том, насколько хорошо написан ваш код. При высокой плотности дефектов следует поискать возможные структурные недостатки.

Метрики также могут использоваться для оценки качества использования абстракций в вашем коде. В хорошей архитектуре используются абстрактные классы и интерфейсы, чтобы другие классы могли программировать для этих интерфейсов вместо конкретных классов реализации. Таким образом, абстракция отделяет части кода от изменений в других частях, по крайней мере в той степени, в какой это возможно в вашей системе. Для оценки этого аспекта может использоваться **метрика абстрактности**:

$$A = N_a / N_c$$

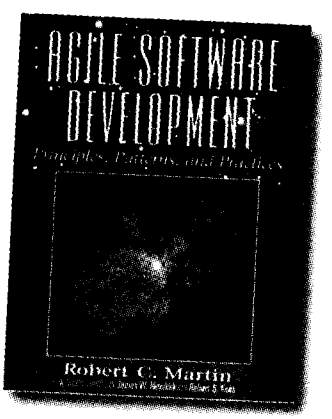
Число всегда лежит в диапазоне от 0 до 1. Большие значения указывают на более высокий уровень использования абстракции, меньшие — на более низкий уровень.

N_a — количество абстрактных классов (и интерфейсов) в конкретном пакете или модуле.

N_c — общее количество классов в одном пакете или модуле.

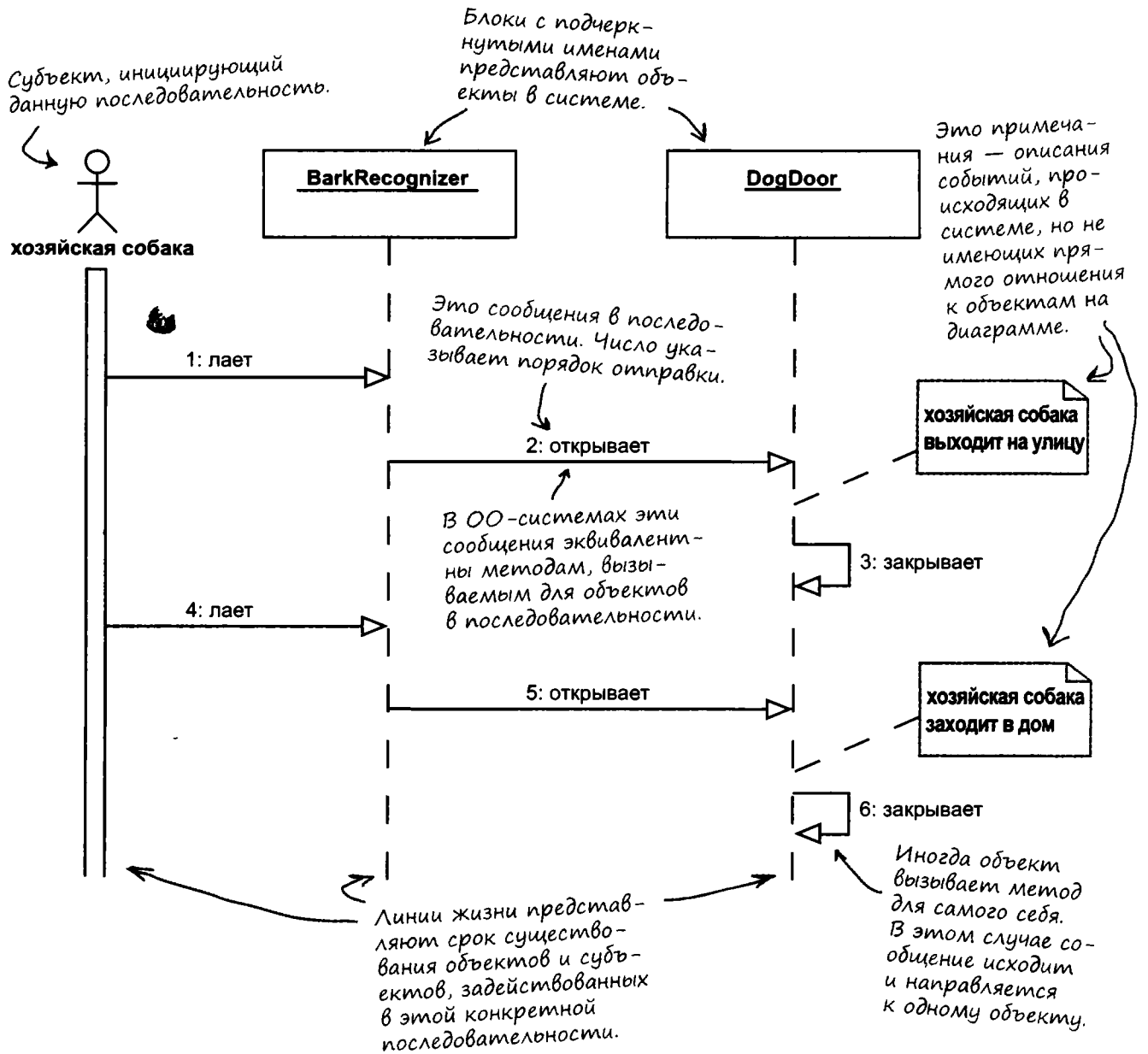
Для пакетов с большим количеством абстракций характерны большие значения A, и наоборот. В общем случае желательно, чтобы каждый пакет программного продукта зависел только от пакетов с более высоким значением A. Это означает, что пакеты всегда зависят от более абстрактных пакетов. Построенные по данному принципу программы проще адаптируются к изменениям.

В книге Роберта Мартина «Agile Software Development» описаны и другие метрики, относящиеся к ОО-проектированию.



6. Диаграммы последовательности

Начиная работать над собачьей дверью для Тодда и Джины, мы разработали несколько альтернативный путей (и у одного альтернативного пути был свой альтернативный путь). Чтобы понять, как в вашей системе реализованы эти альтернативные пути, стоит воспользоваться **диаграммой последовательности** UML. Диаграмма последовательности предоставляет визуальные средства для описания взаимодействий между субъектом и вашей системой.

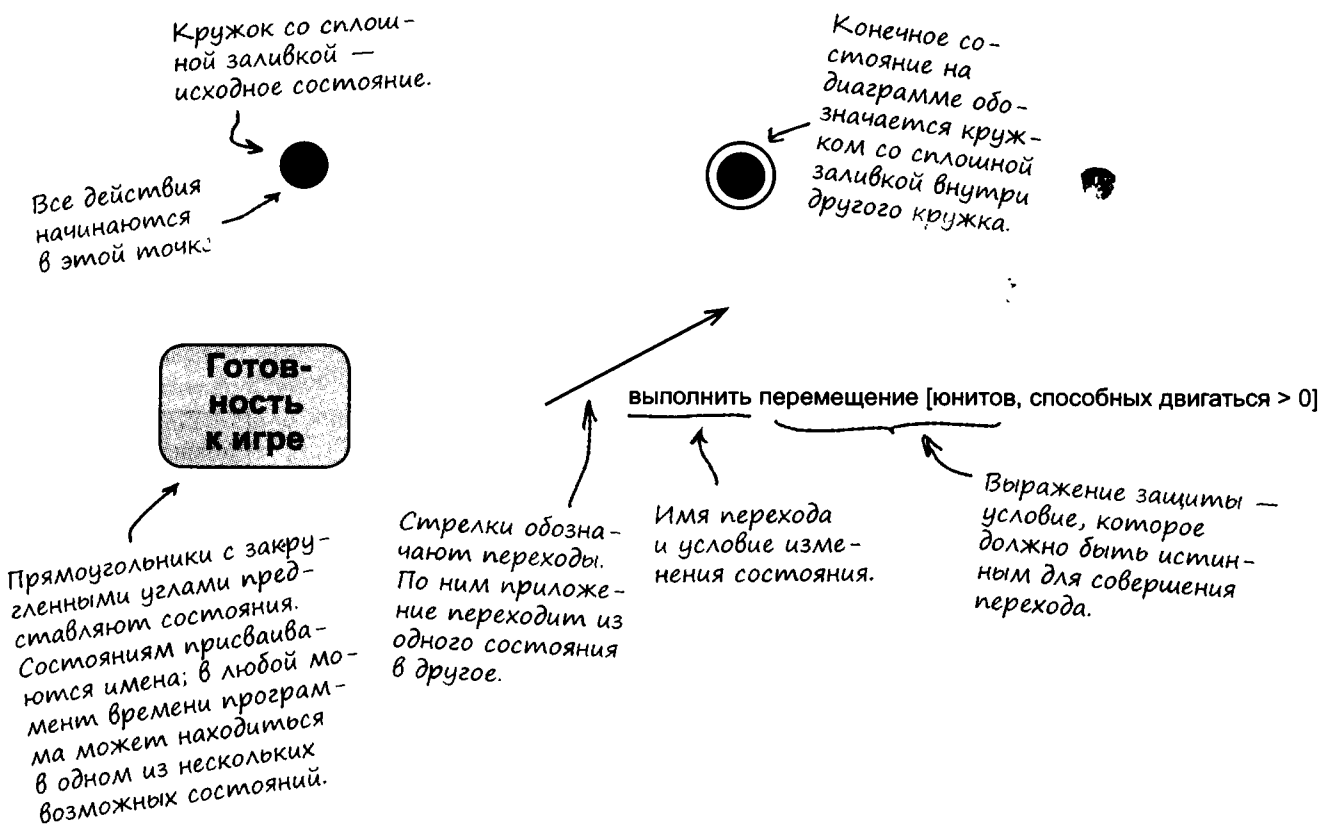


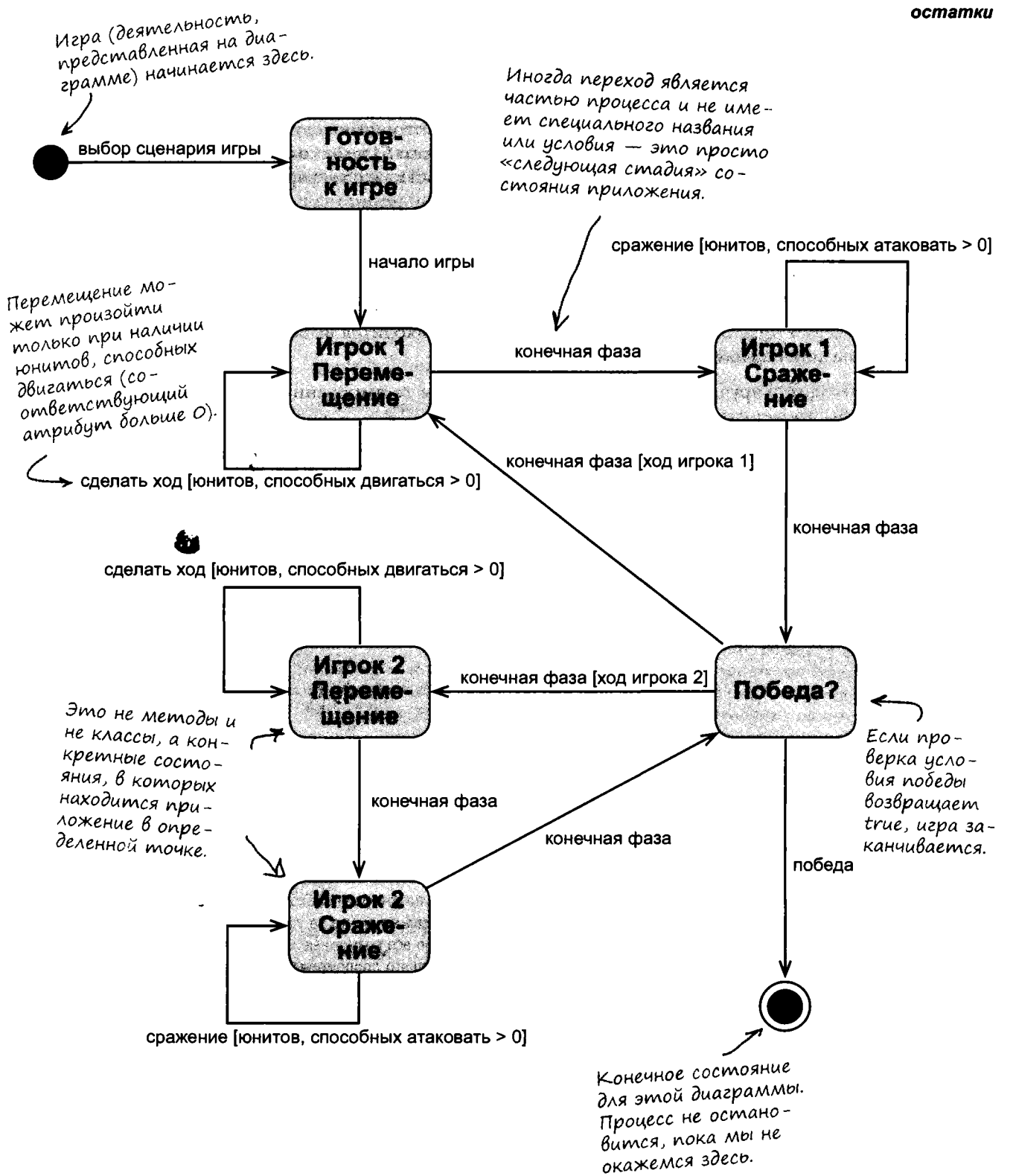
7. Диаграммы состояния

Вы уже видели диаграммы классов и диаграммы последовательности. В UML также существуют диаграммы, называемые диаграммами состояния. Они описывают часть системы, представляя ее различные состояния и действия, которые приводят к их изменению. Диаграммы состояния хорошо подходят для визуального описания сложного поведения.

Диаграммы состояния вступают в игру при наличии нескольких действий и событий, выполняемых одновременно. Справа представлен пример: диаграмма состояния для возможного использования игровой библиотеки Гэри разработчиком игры. Игра, написанная разработчиком на базе библиотеки, с большой вероятностью будет работать именно так, как показано на диаграмме.

Условные обозначения на диаграммах состояния




Игра (деятельность, представленная на диаграмме) начинается здесь.
выбор сценария игры
Готовность к игре
начало игры
Иногда переход является частью процесса и не имеет специального названия или условия — это просто «следующая стадия» состояния приложения.
сражение [юнитов, способных атаковать > 0]
Перемещение может произойти только при наличии юнитов, способных двигаться (соответствующий атрибут больше 0).
Игрок 1 Перемещение
конечная фаза
Игрок 1 Сражение
сделать ход [юнитов, способных двигаться > 0]
конечная фаза [ход игрока 1]
конечная фаза
сделать ход [юнитов, способных двигаться > 0]
Игрок 2 Перемещение
конечная фаза [ход игрока 2]
Победа?
Это не методы и не классы, а конкретные состояния, в которых находится приложение в определенной точке.
Если проверка условия победы возвращает true, игра заканчивается.
конечная фаза
конечная фаза
победа
Игрок 2 Сражение
сражение [юнитов, способных атаковать > 0]
Конечное состояние для этой диаграммы. Процесс не остановится, пока мы не окажемся здесь.

8. Модульное тестирование

В каждой главе, в которой мы работали над приложением, строилась служебная программа для тестирования кода, такая как **`SubwayTester`** или **`DogDoorSimulator`**. Все эти программы выполняют своего рода модульное тестирование. Каждый класс тестируется с определенным набором входных данных или серией вызовов методов.

Хотя такое решение помогает получить представление о том, как будет работать ваше приложение у пользователя, у него есть свои недостатки:

1. Для каждого варианта использования приходится писать полную программу.
2. Для проверки правильности работы программы приходится выводить некие результаты — на консоль или в файл.
3. Вам приходится вручную просматривать результаты тестирования при каждом запуске, чтобы проверить его правильность.
4. Со временем ваши тесты начинают тестировать настолько крупные функциональные блоки, что о тестировании более мелких функциональных возможностей говорить уже не приходится.

К счастью, существуют специальные тестовые системы, которые не только позволяют тестировать очень мелкие функциональные блоки, но и автоматизируют большую часть процесса. Самая популярная система такого рода для Java называется JUnit (http://www.junit.org); она интегрируется со многими популярными средами разработки Java, такими как Eclipse.

Часто задаваемые вопросы

В: Если тесты, написанные в основной части книги, осуществляли высокоуровневую проверку работы нашей программы, зачем нужны дополнительные тесты? Разве этого недостаточно, чтобы убедиться в правильности работы программы?

О: Большинство написанных нами тестов проверяет конкретный сценарий — открыть дверь, выпустить собаку, услышать лай, впустить собаку обратно. Модульные тесты обладают гораздо большей детализацией. Они тестируют функциональность каждого класса по отдельным фрагментам.

Причина, по которой необходимы оба типа тестов, заключается в том, что вам ни за что не удастся создать вариант, тестирующий абсолютно все возможные комбинации функциональных возможностей вашего приложения. Все мы люди и наверняка забудем одну-две нестандартные ситуации.

С тестами, проверяющими каждый отдельный блок функциональности ваших классов, можно не сомневаться, что программа работает в любом варианте, даже если тесты не писались для этого конкретного варианта. Модульные тесты позволяют убедиться в том, что в приложении работают все мелкие фрагменты; это позволяет с высокой степенью вероятности предположить, что объединение этих фрагментов тоже будет работать.

Как выглядит тестовый пример

Тестовый пример содержит тестовые методы для всех без исключения функциональных блоков тестируемого класса. Таким образом, для класса **DogDoor** следует протестировать операции открывания и закрывания двери. JUnit генерирует тестовый класс, который выглядит примерно так:

```
import junit.framework.TestCase;

/**
 * Тестовый пример проверяет работу собачьей двери
 * с использованием кнопки на пульте.
 */
public class RemoteTest extends TestCase
{
    public void testOpenDoor()
    {
        DogDoor door = new DogDoor();
        Remote remote = new Remote(door);
        remote.pressButton();
        assertTrue(door.isOpen());
    }

    public void testCloseDoor() {
        DogDoor door = new DogDoor();
        Remote remote = new Remote(door);
        remote.pressButton();
        try {
            Thread.currentThread().sleep(6000);
        } catch (InterruptedException e) {
            fail("interrupted thread");
        }
        assertFalse(door.isOpen());
    }
}
```

TestCase — базовый класс JUnit для тестирования программных продуктов.

Метод assertTrue() проверяет, что заданный метод возвращает true, как это должно быть в нашем случае.

Для каждого функционального блока DogDoor определяется метод.

Этот метод проверяет автоматическое закрытие двери (в отличие от простого вызова door.close(), что не соответствует обычному использованию двери).

assertFalse() проверяет, что метод НЕ возвращает true.

Тестирование кода в контексте

Обратите внимание: вместо прямого тестирования методов **open()** и **close()** класса **DogDoor** в этом тесте используется класс **Remote**; это соответствует использованию двери в реальном мире. Тесты должны имитировать условия реального использования, даже если они проверяют относительно малый функциональный блок.

То же самое происходит в **testCloseDoor()**. Вместо вызова метода **close()** тест открывает дверь с пульта, выжидает время, необходимое для автоматического закрытия двери, и затем проверяет, закрылась ли дверь. Так дверь будет использоваться, поэтому именно так она должна тестироваться.

9. Стандарты программирования и удобочитаемость кода

Чтение исходного кода имеет много общего с чтением книг. Читатель должен понять, что происходит, и даже если у него возникают вопросы, просто продолжить чтение. Хороший разработчик и проектировщик не жалеет времени на написание удобочитаемого кода, потому что тем самым он улучшает возможности сопровождения и повторного использования.

Перед вами снабженная комментариями, удобочитаемая версия класса **DogDoor**, написанная нами в главах 2 и 3.

```
/**
 * Класс представляет интерфейс к физическому устройству собачьей двери.
 *
 * @author Gary Pollice
 * @version Aug 11, 2006
 */
public class DogDoor
{
    // Количество команд на открытие двери
    private int numberOfOpenCommands = 0;

    boolean doorIsOpen = false;
    /**
     * @return true if the door is open
     */
    public boolean isOpen()
    {
        return doorIsOpen;
    }

    /**
     * Открыть дверь, а через пять секунд закрыть ее.
     */
    public void open( )
    {
        // Код, приказывающий оборудованию открыть дверь
        doorIsOpen = true;
        numberOfOpenCommands++;
        TimerTask task = new TimerTask() {
            public void run() {
                if (--numberOfOpenCommands == 0) {
                    // Здесь размещается код, приказывающий оборудованию закрыть дверь
                    doorIsOpen = false;
                }
            }
        };
        Timer timer = new Timer();
        timer.schedule(task, 5000);
    }
}
```

Комментарии JavaDoc упрощают чтение кода, а также используются при генерировании документации служебной программой javadoc.

Имена методов и переменных содержательны, их смысл достаточно очевиден.

В коде расставлены отступы.

Даже если переменная используется только в пределах одного метода, ей дается содержательное, понятное имя.

Все команды, смысл которых неочевиден, снабжаются комментариями.

Хорошая программа — нечто большее, чем работающий код

Многие разработчики скажут вам, что стандарты форматирования и оформления кода – дело хлопотное. Но посмотрим, что происходит, если вы не позаботитесь об удобочитаемости своего кода:

```
public class DogDoor
{
    private int noc = 0;
    boolean dio = false;

    public boolean returndio( ) { return dio; }

    public void do_my_job( )
    {
        dio = true;
        noc++;
        TimerTask tt = new TimerTask() {
        public void run() {
         if (--noc == 0) dio = false;
        }
        };
        Timer t = new Timer();
        t.schedule(tt, 5000);
    }
}
```

Ни одного комментария... Разработчик должен разобраться во всем самостоятельно.

Совершенно непонятно, для чего используются эти переменные....

...или эти методы. Имена не содержат никакой информации.

Отсутствие отступов только затрудняет понимание.

С чисто функциональной точки зрения эта версия **DogDoor** ничем не отличается от версии с предыдущей страницы. Но сейчас вы уже знаете, что хорошая программа – это не просто работоспособный код, а код, пригодный для повторного использования и удобный в сопровождении. Можно только пожалеть разработчика, которому придется иметь дело с этой версией **DogDoor**; слишком трудно понять, что делает такой код и где могут возникнуть проблемы. А теперь представьте, что вам приходится разбираться в *10 000* таких строк, а не в 25.

Удобочитаемый код упрощает сопровождение и повторное использование — как для вас, так и для других разработчиков.

10. Рефакторинг

Рефакторингом называется процесс модификации структуры кода без изменения его поведения. Рефакторинг выполняется для улучшения четкости, гибкости и расширяемости кода.

Как правило, операции рефакторинга относительно просты и сосредоточены на одном конкретном аспекте структуры кода. Пример:

```
public double getDisabilityAmount() {
  // Проверка права на получение пособия
  if (seniority < 2)
    return 0;
  if (monthsDisabled > 12)
    return 0;
  if (isPartTime)
    return 0;
  // Вычисление и возвращение величины пособия
}
```

Рефакторинг изменяет внутреннюю структуру кода БЕЗ изменения его поведения.

Хотя этот код не имеет явных недостатков, он не настолько прост в сопровождении, насколько нам хотелось бы. **getDisabilityAmount()** в решает две задачи: сначала он проверяет критерий права на получение пособия, а потом вычисляет сумму пособия.

К настоящему моменту вы уже знаете, что это является нарушением принципа единственной обязанности. Код проверки следует отделить от кода вычисления величины пособия. В результате *рефакторинга* код принимает следующий вид:

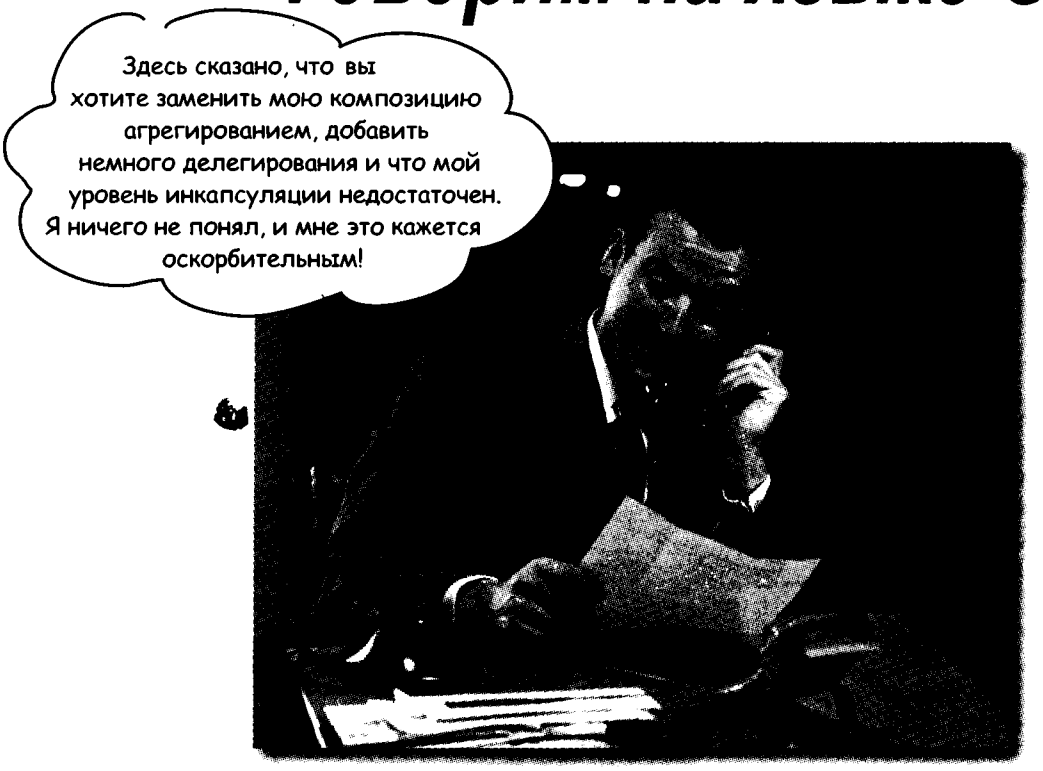
```
public double getDisabilityAmount() {
  // Проверка права на получение пособия
  if (isEligibleForDisability()) {
    // Вычисление и возвращение величины пособия
  } else {
    return 0;
  }
}
```

Две обязанности разделяются на два метода в соответствии с принципом SRP.

Теперь при изменении требований для получения пособия необходимо будет модифицировать только метод **isEligibleForDisability()**, а метод, отвечающий за вычисление суммы, изменяться не будет.

Рефакторинг можно рассматривать как своего рода «текущий осмотр» кода. Это постоянный процесс, так как со временем повторное использование кода только затрудняется. Вернитесь к своему старому коду и переработайте его на основании того, что вы узнали в этой книге. Программисты, которые будут заниматься сопровождением и повторным использованием вашего кода, будут вам благодарны.

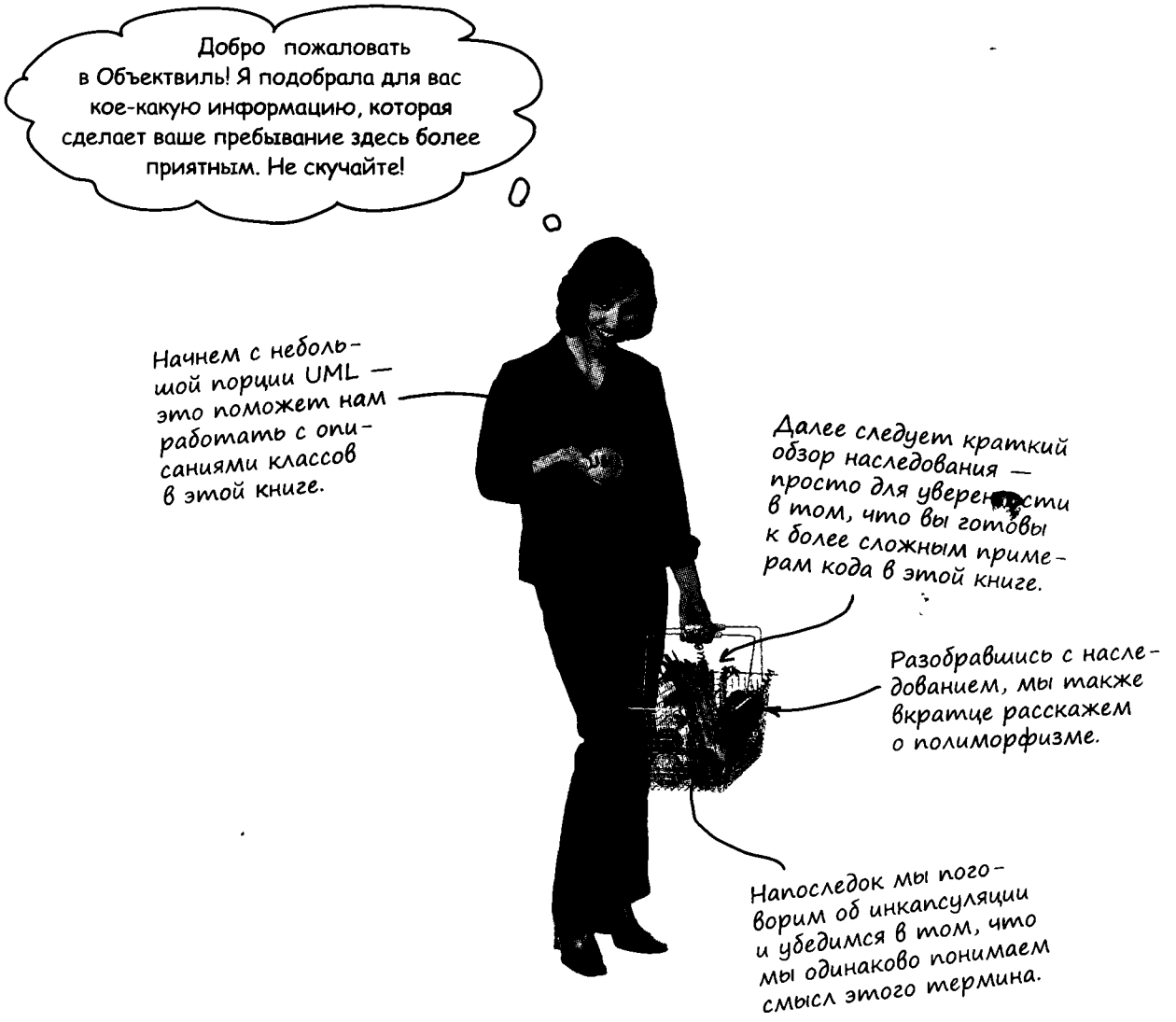
Говорим на языке ООП



Приготовьтесь к путешествию в другую страну. Пришло время навестить Объектвиль — место, где объекты делают то, что им положено, все приложения хорошо инкапсулированы (вскоре вы узнаете, что это означает), а программные структуры обеспечивают простоту расширения и повторного использования. Но прежде чем браться за дело, необходимо кое-что знать заранее — в частности, овладеть кое-какими языковыми навыками. Не беспокойтесь, это не займет много времени. Вы и не заметите, как начнете говорить на языке ООП так, словно вы уже давно живете в элитном районе Объектвиля.

Добро пожаловать в Объектвиль

Неважно, бывали вы в Объективиле прежде или это ваша первая поездка, — другого такого места не найдешь. Но здесь многое кажется непривычным, поэтому мы поможем вам немного освоиться перед тем, как вы возьметесь за основной материал книги.



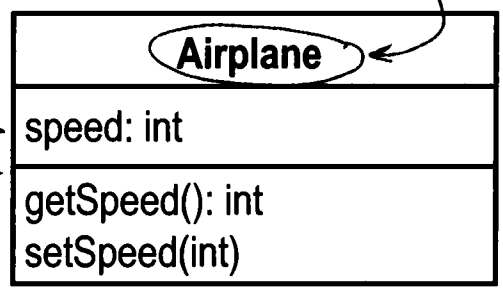
UML и диаграммы классов

В этой книге довольно много говорится о классах и объектах, но по 200 строкам кода составить общее представление о классе довольно трудно. По этой причине мы будем использовать UML (*Unified Modeling Language*) – *язык, используемый для описания* структуры кода и приложений. Синтаксис UML передает информацию, необходимую другим разработчикам и клиентам, без *лишних* подробностей.

Изображение класса на диаграмме классов. Так в UML передается подробная информация о классах в приложении.

Имя класса всегда выделяется жирным шрифтом и располагается в верхней части диаграммы классов.

Переменные экземпляров класса. Каждая переменная обладает именем, а после двоеточия указывается ее тип.



Линия отделяет переменные экземпляров от методов класса.

Методы класса. Для каждого метода указывается имя и список передаваемых при вызове параметров, а после двоеточия указывается тип возвращаемого значения.

По диаграмме классов легко получить общее представление о классе: вы можете с первого взгляда сказать, что делает класс. При желании даже можно не указывать переменные и/или методы, если это поможет более эффективной передаче информации.

Возьми в руку карандаш

Напишите заготовку класса Airplane.

Попробуйте написать простейшую заготовку для класса Airplane по приведенной выше диаграмме. Обнаружили ли вы какую-либо важную информацию, отсутствующую на диаграмме? Запишите:

_____

_____

_____

Возьми в руку карандаш
Решение

Напишите заготовку класса Airplane.

Вам было предложено написать простейшую заготовку класса Airplane по диаграмме классов на с. 591. Вот что получилось у нас:

```
public class Airplane {

  private int speed;

  public Airplane() {
  }

  public void setSpeed(int speed) {
    this.speed = speed;
  }

  public int getSpeed() {
    return speed;
  }
}
```

На диаграмме классов не указано, с каким уровнем доступа должна быть объявлена переменная speed — public, private или protected.

Вообще говоря, диаграммы классов могут содержать эту информацию, но обычно она опускается для более ясной передачи информации.

На диаграмме классов ничего не сказано о конструкторе. Вы также могли написать конструктор, получающий исходное значение speed, и это тоже было бы нормально.

На диаграмме классов не указано, что делает этот метод... Мы сделали разумные предположения, но не можем быть уверены в том, что предполагалось именно это.

Часто задаваемые вопросы

В: Так значит, диаграмма классов не является полным представлением класса?

О: Нет, но она и не должна им быть. Диаграммы классов всего лишь предоставляют базовую информацию о переменных и методах класса. Кроме того, они позволяют обсуждать код без просмотра сотен строк Java, C или Perl.

В: Я использую свое визуальное представление классов; что в этом плохого?

О: Ничего, но другим разработчикам будет труднее разобраться в ваших обозначениях. Использование стандарта, такого как UML, позволяет нам говорить на одном языке и быть уверенными в том, что мы одинаково понимаем тот или иной элемент диаграммы.

В: А кто придумал UML?

О: Спецификация UML была разработана в Rational Software под руководством Грэди Буча, Айвара Джейкобсона и Джима Рамбо (три очень умных человека). В наши дни она находится под управлением OMG (Object Management Group).

В: Не слишком ли много шума для простой диаграммы классов?

О: UML — не только диаграммы классов. В UML имеются диаграммы для представления состояния объектов, последовательности событий в приложениях и даже представления требований заказчика и их взаимодействий с вашей системой. Да и возможности диаграмм классов намного шире, чем описано в книге.

Сейчас нам достаточно основ, представленных на с. 591. О другой информации, которая может отображаться на диаграммах классов, а также о других типах диаграмм, будет рассказано позднее, когда они понадобятся нам в книге.

Наследование

Наследование — одна из фундаментальных концепций программирования в Объективиле. Один класс *наследует* поведение от другого класса и может изменить его в случае необходимости. Давайте посмотрим, как механизм наследования работает в Java; в других языках происходит примерно то же самое.

Jet расширяет класс Airplane. Это означает, что он наследует все поведение Airplane и использует его как свое собственное поведение.

Класс Jet является субклассом по отношению к Airplane. Airplane называется суперклассом для Jet.

```
public class Jet extends Airplane {

  private static final int MULTIPLIER = 2;

  public Jet() {
    super();
  }

  public void setSpeed(int speed) {
    super.setSpeed(speed * MULTIPLIER);
  }

  public void accelerate() {
    super.setSpeed(getSpeed() * 2);
  }

}
```

Субкласс может добавить собственные переменные к переменным, унаследованным от Airplane.

super — специальное ключевое слово. Оно обозначает класс, от которого текущий класс наследует свое поведение. Таким образом, в данном случае вызывается конструктор Airplane суперкласса Jet.

Субкласс может изменить поведение своего суперкласса, а также вызывать его методы. Это называется переопределением поведения суперкласса.

Субкласс может добавить собственные методы к методам, унаследованным от суперкласса.

Также можно использовать вызов super.getSpeed(), но это приведет к обычному вызову getSpeed(), словно getSpeed() является обычным методом, определенным в Jet.

Jet также наследует от Airplane метод getSpeed(). Но поскольку Jet использует ту же версию метода, что и Airplane, нам не придется писать код для изменения этого метода. Хотя в Jet этот метод явно не определяется, вызов getSpeed() для экземпляра Jet абсолютно законен.

Наследование позволяет строить классы на базе других классов без дублирования кода.

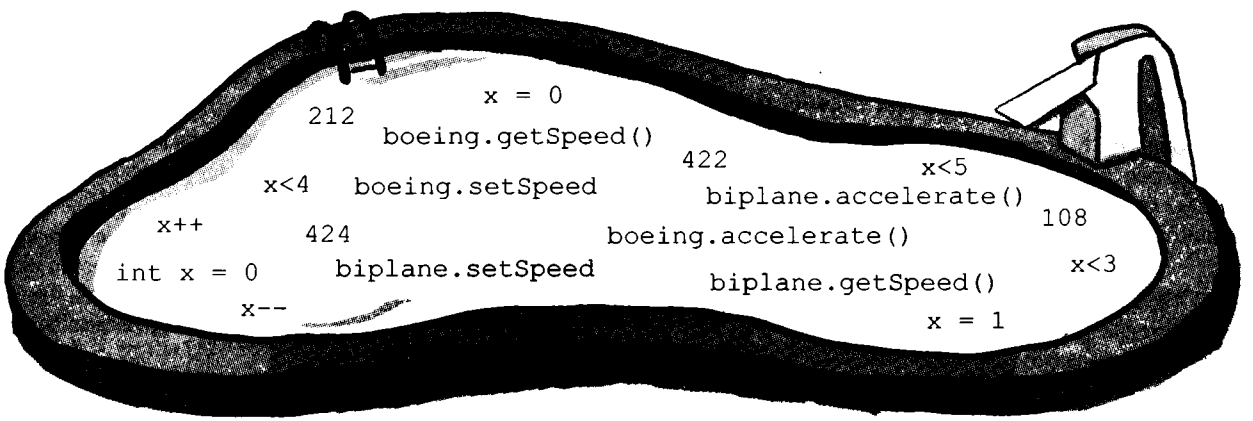
Головоломка у бассейна

Ваша задача — выловить фрагменты кода из бассейна и расставить их на свободные места в коде. Фрагменты можно использоваться повторно, использовать все фрагменты не обязательно. Цель — создание класса, который будет компилироваться, выполняться и давать приведенный результат.

```
public class FlyTest {
  public static void main(String[] args) {
    Airplane biplane = new Airplane();
    biplane.setSpeed(_____);
    System.out.println(_________________);
    Jet boeing = new Jet();
    boeing.setSpeed(_____);
    System.out.println(_________________);
    __________;
    while (______) {
      ___________________;
      System.out.println(___________________);
      if (___________________ > 5000) {
        _________________(___________________ * 2);
      } else {
        ___________________;
      }
      _______;
    }
    System.out.println(___________________);
  }
}
```

Результат:

```
File Edit Window Help LeavingOnAJetplane
%java FlyTest
212
844
1688
6752
13504
27008
1696
```



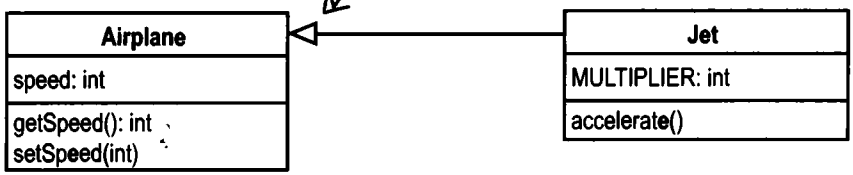
Ответы на с. 601

И еще полиморфизм...

Полиморфизм тесно связан с наследованием. Когда один класс наследует от другого, полиморфизм позволяет **экземплярам субкласса заменять экземпляры суперкласса**.

Еще одна диаграмма классов, на этот раз с двумя классами.

Стрелка означает, что класс Jet наследует от Airplane. Пока не обращайте внимания на это обозначения, позднее мы гораздо подробнее рассмотрим представление наследования на диаграммах классов.



Jet является субклассом Airplane. Это означает, что везде, где может использоваться Airplane...

```
Airplane plane = new Airplane();
```

...также можно использовать Jet.

```
Airplane plane = new Jet();
```

Слева указывается суперкласс...

```
Airplane plane = new Airplane();
```

...а справа может использоваться суперкласс ИЛИ любой из его субклассов.

```
Airplane plane = new Airplane();
Airplane plane = new Jet();
Airplane plane = new Rocket();
```

Считаем, что Rocket — еще один субкласс Airplane.

Часто задаваемые вопросы

В: И чем так полезен полиморфизм?

О: Вы можете написать код, который работает с суперклассом (например, **Airplane**), но также будет работать с любыми типами субклассов, такими как **Jet** или **Rocket**. Таким образом код становится более гибким.

В: Не понимаю, как полиморфизм сделает мой код более гибким.

О: Если вам понадобится новая функциональность, вы можете написать новый субкласс, наследующий от **Airplane**. Но поскольку ваш код работает с суперклассом, новый субкласс будет работать без каких-либо изменений в остальном коде! Это означает, что код становится более гибким и его будет проще изменять.

И наконец, инкапсуляция

Под инкапсуляцией понимается сокрытие реализации класса, обеспечивающее простоту его использования и изменения. Класс работает по принципу «черного ящика»: он предоставляет сервис своим пользователям, но скрывает свою реализацию, чтобы другие программисты не могли изменить или некорректно использовать его. **Инкапсуляция играет ключевую роль в соблюдении принципа открытости/закрытости. Допустим, мы переписали класс `Airplane`** в следующем виде:

Переменная speed объявлена открытой (public) вместо приватной (private), и теперь любая часть вашего приложения сможет напрямую обращаться к ней.

```
public class Airplane {

  public int speed;

  public Airplane() {
  }

  public void setSpeed(int speed) {
    this.speed = speed;
  }

  public int getSpeed() {
    return speed;
  }
}
```

Инкапсуляцией называется защита информации в программном коде от некорректного использования.

Теперь кто угодно может задать значение speed напрямую

Внесенное изменение означает, что остальному коду приложения уже не обязательно вызывать **`setSpeed()`**, чтобы задать скорость самолета; вместо этого значение переменной можно задать напрямую. Таким образом, следующий код успешно откомпилируется:

```
public class FlyTest2 {
  public static void main(String[] args) {
    Airplane biplane = new Airplane();
    biplane.speed = 212;
    System.out.println(biplane.speed);
  }
}
```

Использовать методы setSpeed() и getSpeed() уже не обязательно... Можно обратиться к переменной speed напрямую.

Опробуйте этот код... Вас ничего не удивляет в полученных результатах?

И из-за чего столько шума?

И что тут такого особенно страшного? Но что произойдет, если создать экземпляр **Jet** и задать значение speed следующим образом:

```
public class FlyTest3 {
  public static void main(String[] args) {
    Jet jet1 = new Jet();
    jet1.speed = 212;
    System.out.println(jet1.speed);

    Jet jet2 = new Jet();
    jet2.setSpeed(212);
    System.out.println(jet2.getSpeed());
  }
}
```

Использование Jet без инкапсуляции.

Так как Jet наследует от Airplane, переменную speed его суперкласса можно использовать так, словно она является частью Jet.

Использование Jet с инкапсуляцией

Так мы задаем значение и получаем доступ к переменной speed, когда она закрыта для прямого доступа.

Возьми в руку карандаш

Какую пользу приносит инкапсуляция данных?

Введите, откомпилируйте и запустите код FlyTest3.java, приведенный выше. Как будет выглядеть результат? Запишите две строки вывода в следующих полях:

Speed экземпляра jet1: _____

Speed экземпляра jet2: _____

Как вы думаете, что здесь произошло? Запишите, как, по вашему мнению, были получены значения speed для каждого экземпляра Jet:

_____

_____

_____

Напишите, в чем, по-вашему, состоит ценность инкапсуляции:

_____

_____

_____

Возьми в руку карандаш
Решение

Какую пользу приносит инкапсуляция данных?

Введите, откомпилируйте и запустите код FlyTest3.java, приведенный выше. Как будет выглядеть результат? Запишите две строки вывода в следующих полях:

Speed экземпляра jet1: **212**

Speed экземпляра jet2: **424**

Как вы думаете, что здесь произошло? Запишите, как, по вашему мнению, были получены значения speed для каждого экземпляра Jet:

В классе Jet метод setSpeed() получает переданное значение и умножает его на 2, прежде чем задавать speed. При ручном задании переменной speed умножение на 2 не производится.

Ваша формулировка не обязана совпадать с нашей слово в слово, но у вас должно было получиться что-то похожее.

Напишите, в чем, по-вашему, состоит ценность инкапсуляции:

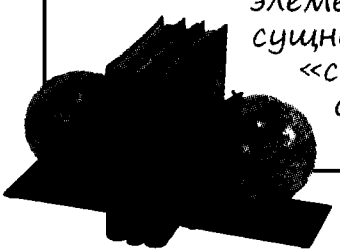
Инкапсуляция защищает данные от некорректного использования. При инкапсуляции выполняются все необходимые вычисления или проверки данных, запрограммированные в классе, поскольку обратиться к данным напрямую невозможно.

Итак, инкапсуляция не ограничивается простым сокрытием информации; она гарантирует, что при работе с информацией будут использоваться методы, написанные вами для этой цели!

Посещаете курсы программирования? Здесь дано официальное определение инкапсуляции... Если вы будете сдавать экзамен, используйте именно его.

За партой

Инкапсуляция. Процесс размещения программных элементов в больших, более абстрактных сущностях. Также используются термины «сокрытие информации» и «разделение обязанностей».



Часто задаваемые вопросы

В: Значит, суть инкапсуляции в том, чтобы объявить все переменные приватными?

О: Нет, суть инкапсуляции в том, чтобы изолировать информацию от других частей приложения, которые не должны некорректно работать с этой информацией, поэтому вы и отделяете эти данные, объявляя их приватными. Если данные должны обновляться, вы можете предоставить методы, ответственно работающие с этими данными (как мы сделали с классом **`Airplane`**, используя методы **`getSpeed()`** и **`setSpeed()`**).

В: Существуют ли другие способы использования инкапсуляции, кроме как с переменными?

О: Безусловно. В главе 1 мы рассматриваем возможность инкапсуляции группы свойств от объекта и предотвращение некорректного использования этих свойств объектом. Хотя в этом случае мы имеем дело с целым набором свойств, речь все равно идет об изоляции информации от остальных частей приложения.

В: Так значит, основная цель инкапсуляции — защита данных?

О: Не только! Инкапсуляция также помогает изолировать поведение от других частей приложения. Таким образом, вы можете разместить в методе большой объем кода и включить этот метод в класс; поведение изолируется от остальных частей приложения, а приложение должно использовать новый класс и метод для обращения к этому поведению. Впрочем, при этом используются те же принципы, что и при работе с данными: вы изолируете части своего приложения, чтобы защитить их от некорректного использования.

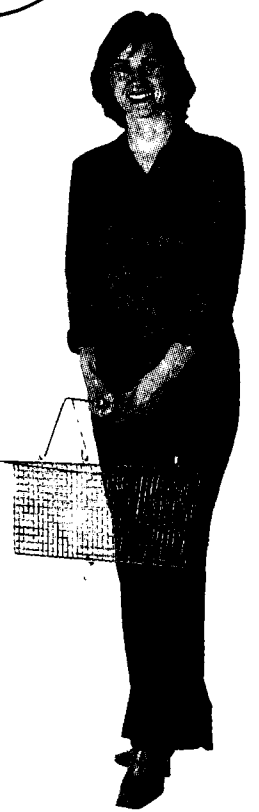
В: Ух... Что-то у меня голова кругом идет. Что делать?

О: Просто продолжайте читать. Убедитесь в том, что вы поняли решения на с. 598, — это означает, что вы готовы к главе 1. Мы еще проведем достаточно много времени за изучением всех ОО-принципов и концепций; не считайте, что вам обязательно нужно во всем разобраться прямо сейчас.

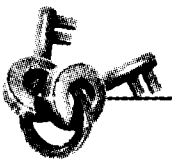
Инкапсуляция отделяет данные от поведения приложения.

Это позволяет вам управлять тем, как каждая часть приложения используется его остальными компонентами.

Уже все просмотрели? Тогда вы определенно готовы к основному материалу книги. И еще раз добро пожаловать в хорошо спроектированную часть Объектвиля... нам здесь нравится. Уверена, что понравится и вам.



КЛЮЧЕВЫЕ МОМЕНТЫ



- Сокращение UML образовано от Unified Modeling Language («Унифицированный язык моделирования»).
- UML помогает описывать структуру приложений для других разработчиков, клиентов и руководителей.
- Диаграмма классов дает общий обзор класса, его методов и переменных.
- Наследованием называется расширение классом другого класса для повторного использования или модификации поведения последнего.
- Класс, от которого наследуется поведение, называется суперклассом; наследующий класс называется субклассом.
- Субкласс автоматически получает все поведение своего суперкласса.
- Субкласс может переопределять поведение суперкласса для изменения работы его методов.
- Полиморфизм — это «подстановка» субкласса на место своего суперкласса.
- Полиморфизм улучшает гибкость приложения и его способность к изменениям.
- Инкапсуляцией называется изоляция или сокрытие некоторой части кода от остального кода.
- Простейшей формой инкапсуляции является объявление приватных переменных классов. Доступ к таким переменным осуществляется только через методы класса.
- Также можно инкапсулировать группы данных (и даже поведение) для управления доступом к ним.

Головоломка у бассейна. Решение

Вам было ***поручено*** вытащить фрагменты кода из бассейна и расставить их в пропусках в коде справа. Один фрагмент **может** использоваться многократно, и вы не обязаны использовать все фрагменты. Ваша ***цель*** заключалась в создании класса, который компилируется, выполняется и выводит указанный результат.

```
public class FlyTest {
  public static void main(String[] args) {
    Airplane biplane = new Airplane();
    biplane.setSpeed(212);
    System.out.println(biplane.getSpeed());
    Jet boeing = new Jet();
    boeing.setSpeed(422);
    System.out.println(boeing.getSpeed());
    int x = 0;
    while (x<4) {
      boeing.accelerate();
      System.out.println(boeing.getSpeed());
      if (boeing.getSpeed() > 5000) {
        biplane.setSpeed(biplane.getSpeed() * 2);
      } else {
        boeing.accelerate();
      }
      x++;
    }
    System.out.println(biplane.getSpeed());
  }
}
```

Наше решение головоломки у бассейна.

Результат:

```
File Edit Window Help LeavingOnAJetplane
%java FlyTest
212
844
1688
6752
13504
27008
1696
```

Маклафлин Б., Поллайс Г., Уэст Д.

Объектно-ориентированный анализ и проектирование

Перевел с английского Е. Матвеев

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Кривцов*
Ведущий редактор	*Ю. Сергиенко*
Художественный редактор	*Л. Адуевская*
Корректор	*Н. Викторова*
Верстка	*Е. Лвля*

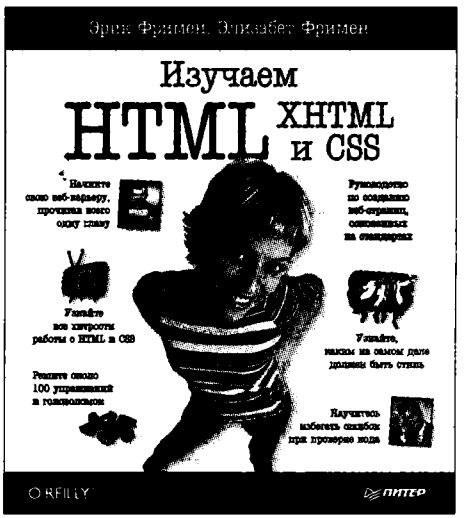
ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 26.12.12. Формат 84x108/16. Усл. п. л. 63,840. Тираж 2000. Заказ С-144.

Отпечатано в полном соответствии с качеством
предоставленного электронного оригинал-макета
в типографии филиала ОАО «ТАТМЕДИА» «ПИК «Идел-Пресс».
420066, г. Казань, ул. Декабристов, 2.

Изучаем HTML, XHTML и CSS

Head First HTML with CSS & XHTML

Э. Фримен, Э. Фримен



ISBN: 978-5-49807-113-8
Объем: 656 с.

Устали от чтения таких книг по HTML, которые понятны только специалистам в этой области? Тогда самое время взять в руки наше издание. Хотите изучить HTML так, чтобы уметь создавать веб-страницы, о которых вы всегда мечтали? Так, чтобы более эффективно общаться с друзьями, семьей и привередливыми клиентами? Хотите действительно обслуживать и улучшать HTML-страницы по прошествии времени, чтобы они работали во всех браузерах и мобильных устройствах? Тогда эта книга для вас. Прочитав ее, вы узнаете все секреты создания веб-страниц. Благодаря ей вам больше не придется думать, какие цвета нужно использовать, чтобы они сочетались между собой, как правильно применять шрифты, чтобы они не «плавали» по экрану и верно отображались в различных браузерах. Вы узнаете, как работают профессионалы, чтобы получить визуально привлекательный дизайн, и как максимально эффективно использовать HTML, CSS и XHTML, чтобы создавать такие веб-страницы, мимо которых не пройдет ни один пользователь.

Изучаем JavaScript

Head First JavaScript

М. Моррисон

ISBN: 978-5-459-00322-2

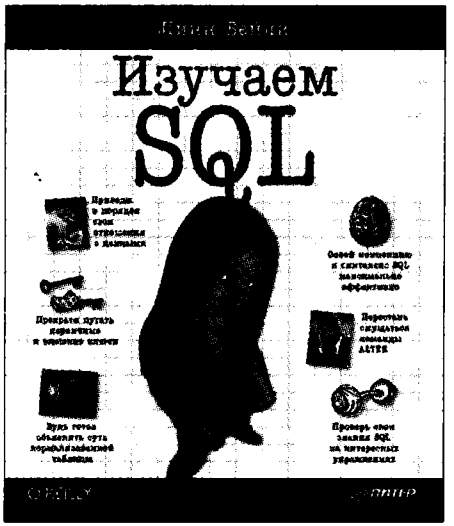
Объем: 592 с.

Вы готовы сделать шаг вперед в своей практике веб-программирования 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 java-скрипта до создания сложных веб-проектов, которые будут работать во всех современных браузерах. Особенностью данного издания является уникальный способ подачи материала, выделяющий серию «Head First» издательства O'Reilly в ряду множества скучных книг, посвященных программированию.

Изучаем SQL

Head First SQL

Л. Бейхли



ISBN: 978-5-459-00421-2

Объем: 592 с.

В современном мире наивысшую ценность имеет информация, но не менее важно уметь этой информацией управлять. Эта книга посвящена языку запросов SQL и управлению базами данных. Материал излагается, начиная с описания базовых запросов и заканчивая сложными манипуляциями с помощью объединений, подзапросов и транзакций. Если вы пытаетесь разобраться в организации и управлении базами данных — эта книга будет отличным практическим пособием и предоставит вам все необходимые инструменты. Особенностью данного издания является уникальный способ подачи материала, выделяющий серию «Head First» издательства O'Reilly в ряду множества скучных книг, посвященных программированию.

Паттерны проектирования

Head First Design Patterns

Э. Фримен, Э. Фримен, К. Сьерра, Б. Бейтс

ISBN: 978-5-459-00435-9

Объем: 656 с.

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.

Программируем для iPhone и iPad

Head First iPhone and iPad Development, 2nd Ed.

Д. Пилон, Т. Пилон

ISBN: 978-5-459-00375-8

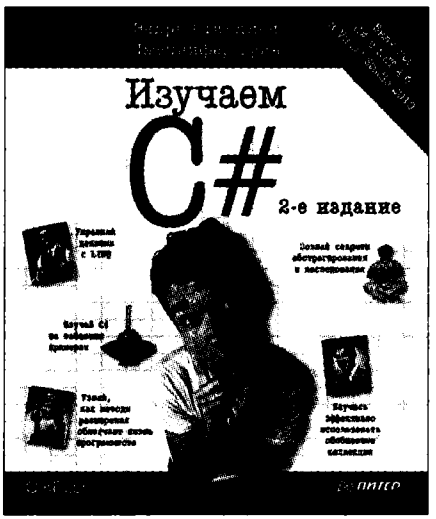
Объем: 592 с.

Книга предназначена для профессиональных программистов, желающих создавать приложения для iPhone и iPod с помощью Apple SDK и языка программирования Objective-C. В отличие от большинства книг по программированию, построенных на основе скучного изложения спецификаций и примеров, с этой книгой читатель сможет приступить к написанию собственного кода с самого начала. Вы освоите минимальный набор инструментов и примете участие в забавных и интересных программных проектах: от разработки игр до создания серьезных бизнес-приложений. Нестандартный способ подачи материала облегчает понимание и значительно увеличивает скорость восприятия, заставляя вас думать головой. Не пугайтесь, даже если ранее вы не написали ни одной строчки кода — благодаря своему уникальному визуальному формату подачи материала эта книга с легкостью проведет вас по всему пути создания качественного приложения для iPhone и iPad.

Изучаем C#. 2-е изд.

Head First C#, 2nd Ed.

Э. Стиллмен, Дж. Грин



ISBN:978-5-459-00422-9

Объем: 696 с.

В отличие от большинства книг по программированию, построенных на основе скучного изложения спецификаций и примеров, с этой книгой читатель сможет сразу приступить к написанию собственного кода на языке программирования C# с самого начала. Вы освоите минимальный набор инструментов, а далее примете участие в забавных и интересных программных проектах: от разработки карточной игры до создания серьезного бизнес-приложения. Второе издание книги включает последние версии C# .NET 4.0 и Visual Studio 2010 и будет интересно всем изучающим язык программирования C#. Особенностью данного издания является уникальный способ подачи материала, выделяющий серию «Head First» издательства O'Reilly в ряду множества скучных книг, посвященных программированию.