Шаблоны реализации корпоративных приложений

Implementation Patterns

Kent Beck

ADDISON-WESLEY

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokio • Singapore • Mexico City

Шаблоны реализации корпоративных приложений

Кент Бек

Москва • Санкт-Петербург • Киев
2008

ББК 32.973.26-018.2.75
Б42
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *А.В. Чеботарева*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Бек, Кент.

Б42 Шаблоны реализации корпоративных приложений. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2008. — 176 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1406-4 (рус.)

Кент Бек, один из самых креативных и признанных лидеров в индустрии программного обеспечения, собрал 77 шаблонов, предназначенных для обслуживания задач ежедневного программирования и написания более читаемого кода. Эта новая коллекция шаблонов предназначена для реализации многих аспектов разработки, включая классы, состояние, поведение, методы, коллекции, инфраструктуры и т.д. Автор использует диаграммы, истории, примеры и эссе для того, чтобы увлечь читателя по ходу описания шаблонов. Вы обнаружите проверенные решения для управления всем, от именования переменных до проверки исключений.

Эта книга предназначена для программистов всех уровней подготовки, особенно для тех, кто применяет в своей практике шаблоны проектирования и методы быстрой разработки. Книга также окажется неоценимым ресурсом для команд разработчиков, ищущих более эффективные методы совместной работы и построения более управляемого ПО.

ББК 32.973.26-018.2.75

ISBN 978-5-8459-1406-4 (рус.)
ISBN 0-321-41309-1 (англ.)

Оглавление

Содержание

Отзывы о книге

"Кент — мастер кода и отличный рассказчик, книги которого можно легко и с удовольствием читать и понимать. Каждая глава этой книги содержит прекрасные объяснения и проникает в самую суть маленьких, но важных решений, постоянно принимаемых нами, когда мы создаем код и классы".

Эрик Гамма (Erich Gamma), ведущий инженер IBM

"У многих команд есть ведущий разработчик, выдающий ряд правильных решений каждый день. Его мысли всегда движутся в верном направлении. Получающийся код легко понять и модифицировать, он безопасен и удобен. Эта книга поможет вам стать ведущим разработчиком команды. Ширина и глубина затронутых тем не оставят равнодушными профессионалов, которые увидят здесь новые приемы и улучшат старые методы, в то время как простота изложения делает книгу доступной и новичкам".

Русс Руфер (Russ Rufer), Silicon Valley Patterns Group

"Многие люди не понимают, насколько читабельным может быть код и насколько ценна эта читабельность. Кент научил меня очень многому, и я рад, что эта книга дает шанс научиться и остальным".

Мартин Фоулер (Martin Fowler), ведущий ученый ThoughtWorks

"Код должен заслуживать прочтения не только компилятором, но и людьми. Кент Бек превратил свой опыт в единый набор шаблонов реализации. Его советы сделают ваш код действительно заслуживающим прочтения".

Грегор Хохпе (Gregor Hohpe), автор Enterprise Integration Patterns

"В этой книге Кент Бек показывает, как из приложения простых принципов вытекает написание чистого и читабельного кода. *Шаблоны реализации* помогут разработчикам писать раскрывающий свой замысел код, который одновременно легок для понимания и гибок для дальнейшего расширения. Книгу должны обязательно прочитать все серьезные разработчики".

Свен Гортс (Sven Gorts)

"*Шаблоны реализации* заполняют пробел между проектированием и программированием. Бек предлагает новый способ думать о программировании, основываясь на ценностях и принципах".

Диомидис Спинеллис (Diomidis Spinellis), автор Code Reading и Code Quality

Посвящается Синди. Спасибо тебе за поддержку, настойчивость, питание, ободрение, стимуляцию, редактирование и за чай. Моя преданность этой книге — как муравей рядом со слоном, по сравнению с тем, что ты сделала для меня. Будь здорова.

Предисловие

Эта книга о программировании, а именно о написании понятного кода. Нет никакой магии в создании кода, который могут читать другие люди. Любое писательство предполагает знание целевой аудитории, четкое понимание общей структуры предмета, выражение деталей так, чтобы они составляли единую историю. Язык Java предлагает хорошие средства коммуникации. Изложенные шаблоны реализации — это привычки Java-программиста, приводящие к написанию читабельного кода.

Другой взгляд на шаблоны реализации — это способ мышления: "Что я хочу рассказать читателю об этом коде?" Программисты проводят так много времени в своих собственных размышлениях, что попытка взглянуть на код с чужой точки зрения — значительный прогресс. Полезно задуматься не только над тем, "Что компьютер будет делать с этим кодом?", но и "Как я могу пояснить свой образ мысли другим людям?" Это здоровый и потенциально прибыльный сдвиг перспективы, так как очень много денег в сфере разработки ПО тратится на понимание существующего кода.

В Америке есть шоу "Джеопарди", в котором ведущий предлагает ответы, а соперники пытаются отгадать вопросы. "Человек, умеющий работать по дереву. — Столяр? — Правильно!"

Программирование сходно с "Джеопарди". Язык Java предоставляет ответы в форме базовых конструкций. Программистам обычно нужно выразить вопросы для самих себя, понять, какие проблемы решаются каждой конструкцией языка. Если ответ — это объявление поля `Set`, то вопросом может быть "Как я могу рассказать другим программистам, что набор не содержит копий?" Шаблоны реализации составляют каталог общих проблем программирования и свойств Java, относящихся к этим проблемам.

Контроль объема работ при написании книги так же важен, как и в разработке ПО. Настоящая книга не является руководством по стилю, поскольку она содержит слишком много объяснений и оставляет последнее слово за читателем. Это не книга по проектированию, потому что здесь затрагиваются в основном решения мелкого масштаба, которые типичный программист делает по многу раз в день. Это не сборник шаблонов, так как их формат создан для частного применения. Наконец, это не описание языка, хотя книга и затрагивает многие свойства Java, но предполагается, что читатель уже знаком с этим языком.

Фактически книга основана на достаточно хрупком предположении — значимости хорошего кода. Я видел слишком много плохих программ, принесших большие деньги, чтобы не верить, что качество кода необходимо или достаточно для его коммерческого успеха или повсеместного использования. Однако я все еще верю в значимость качественного кода, даже если он не обеспечивает контроля над будущим.

Программы, способные уверенно развиваться и идти в народ, менять вектор развития в зависимости от возможностей и конкуренции и не терять свой дух перед сложными проблемами и неудачами, будут более успешны, чем программы с убогим, полным ошибок кодом.

Даже если тщательность в программировании не принесет долгосрочных экономических результатов, я все равно буду писать как можно более качественный код. Семьдесят лет жизни состоят из немногим более двух миллиардов секунд. Не хотелось бы их потратить на работу, которой я не горжусь. Хорошее программирование приносит удовольствие само по себе и дает знание, что другие будут способны понять, оценить, использовать и расширить мою работу.

Наконец, это книга об ответственности. Как программисту вам даны время, талант, деньги и возможности. Что вы сделаете для ответственного использования этих даров? На последующих страницах приведен ответ на этот вопрос для меня: программируй для других, как для самого себя и своего приятеля, центрального процессора.

Благодарности

Сначала, впоследствии и всегда я буду благодарен Синтии Андрес (Cynthia Andres) — партнеру, редактору, группе поддержки и главному “стимулятору”. Мой друг Пол Петралио (Paul Petralia) был вместе со мной во время работы над этим проектом и ободрял меня своими телефонными звонками. Мой редактор Крис Гужиковски (Chris Guzikowski) и я в ходе выполнения проекта наконец-то научились работать вместе. Он обеспечил мне поддержку со стороны редакции в Пирсоне, что позволило закончить книгу. Спасибо производственной бригаде из Пирсона: Джули Нахиль (Julie Nahil), Джону Фуллеру (John Fuller) и Синтии Когут (Cynthia Kogut). Дженнифер Конке (Jennifer Kohnke) изготовила информативные и отличные иллюстрации. Мои критики, Эрик Гамма (Erich Gamma), Стив Метскер (Steve Metsker), Диомидис Спинеллис (Diomidis Spinellis), Том де Марко (Tom deMarco), Михаэль Физерс (Michael Feathers), Дож Леа (Doug Lea), Брэд Абрамс (Brad Abrams), Клифф Клик (Cliff Click), Пекка Абрахамсон (Pekka Abrahamson), Грегор Хохпе (Gregor Hohpe) и Мишель Марчези (Michele Marchesi), обеспечивали четкую и своевременную обратную связь.

Спасибо Дэвиду Саффу (David Saff) за объяснение симметрии между состоянием и поведением. Мои дети, Линкольн, Линдсей, Форрест и Джолли Андре-Бек, оставались дома и поторапливали меня с окончанием.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было бы сделать лучше, и что вы бы еще хотели увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши электронные адреса:

E-mail: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

в Украине: 03150, Киев, а/я 152

Глава 1

Вступление

И вот мы вместе. Вы взяли в руки мою книгу, теперь она ваша. Вы уже писали код, вероятно, у вас уже есть опыт и выработался собственный стиль.

Цель этой книги — помочь вам передать ваши замыслы в коде. Книга начинается с обзора программирования и шаблонов (главы 2–4). Оставшаяся часть книги (главы 5–8) содержит набор шаблонов и коротких заметок относительно написания читаемого кода на Java. Книга завершается главой о том, как нужно модифицировать приведенные советы в случае, если вы создаете не приложение, а инфраструктуру (framework). В целом книга фокусируется на технологиях программирования, которые расширяют возможности коммуникации.

Существует несколько шагов к началу взаимодействия через код. Для начала я должен был научиться программировать сознательно. Прошло несколько лет, прежде чем я начал воплощать свои мысли в шаблонах. Я был поражен тем фактом, что, хотя решения приходили ко мне легко и быстро, я не мог объяснить своей уверенности в том, что метод должен называться так-то или что данный предмет логически объясним. Первым шагом к началу взаимодействия стало замедление с целью осознать свои мысли и преодолеть мнение, будто я программирую инстинктивно.

Вторым шагом стало понимание важности остальных людей. Я находил удовлетворение в программировании, но был замкнут на себе. Перед тем как я смог написать взаимодействующий код, я должен был поверить, что остальные люди так же важны, как и я сам. Программирование вряд ли является просто контактом между человеком и машиной. Забота об остальных людях — это сознательное решение, требующее практики.

И таким образом я пришел к третьему шагу. С тех пор как я открыл свои мысли солнечному свету и свежему воздуху и понял, что другие люди имеют столько же прав на существование, как и я, у меня появилась необходимость продемонстрировать открывшиеся перспективы на практике. Я использую эти воплощенные принципы для того, чтобы программировать сознательно, для других так же, как для самого себя.

Вы можете прочитать эту книгу только ради технических деталей — полезных приемов с объяснениями. Однако я считаю нужным уведомить вас, что здесь есть нечто гораздо большее, по крайней мере для меня.

Вы можете найти эти технические части, пролистывая главы с шаблонами. Эффективная стратегия заключается в прочтении материала перед его использованием. Чтобы читать “непосредственно во время”, я предлагаю пропустить все, вплоть

до главы 5, и бегло просмотреть остаток до конца, после чего держать книгу под рукой, когда вы программируете. После того как вы используете некоторые шаблоны, всегда можно вернуться назад к вступительному материалу, чтобы ознакомиться с философской основой использованных идей.

Если же вы заинтересованы в тщательном осмыслении изложенного материала, то можете читать непосредственно с самого начала. В отличие от остальных моих книг, главы в этой книге довольно длинные, поэтому потребуется сконцентрироваться, чтобы прочесть целую часть от начала до конца.

Большая часть материала этой книги построена по принципу шаблонов. Большинство программных решений похожи на применявшиеся ранее. Вы можете определить миллион переменных за время вашей программистской карьеры. У вас не появится совершенно нового подхода к именованию каждой переменной. Общие ограничения к именованию всегда те же: вы должны донести назначение, тип и время жизни переменной до читателей, вы должны выбрать легкочитаемое имя, вы должны выбрать имя, которое легко писать и форматировать. Добавьте к этому общие ограничения к специфике отдельной переменной, и вы получите ее рабочее имя. Наименование переменных является примером шаблона: решение и его ограничения повторяются, хотя вы можете создавать каждый раз другое имя.

Я думаю, шаблоны часто нуждаются в различных представлениях. Иногда шаблон лучше объяснить содержательной заметкой, иногда диаграммой, поучительной историей или примером. Вместо того чтобы втискивать каждый шаблон в жесткий формат, я описываю каждый из них наиболее соответствующим образом. Эта книга содержит 77 однозначно именованных шаблонов, каждый из которых затрагивает свой аспект написания читаемого кода. В дополнение здесь есть также много меньших шаблонов или их вариаций, которые я хотел бы упомянуть. Цель данной книги — посоветовать, как выполнять самые обыденные, ежедневные задачи программирования таким образом, чтобы будущие читатели могли понять, для чего нужен данный код.

Эта книга находится где-то посередине между книгой *Шаблоны проектирования* (Design Patterns) и руководством по Java. В книге о *шаблонах проектирования* речь идет о решениях, которые принимаются по несколько раз в день во время разработки приложений, обычно относящихся к регулировке взаимодействий между объектами. Вы применяете *шаблоны реализации* каждые несколько секунд, когда программируете. В то время как руководства по языку хороши для описания того, что вы можете сделать на Java, в них не говорится, почему вам следует использовать определенную конструкцию кода, или о выводах, которые может сделать кто-то, читающий ваш код.

Часть моей философии при написании этой книги заключалась в том, чтобы придерживаться хорошо знакомых мне тем. Результаты параллелизации, например, не входят в эти шаблоны реализации не потому, что многопоточность не является важным результатом, но потому, что это не то, о чем здесь следовало бы сказать. Моей стратегией многопоточности всегда была максимальная изоляция параллельных частей приложения. Пока что это мне в целом удается, так что я не хочу это разъяснять.

Я рекомендую книгу *Java Concurrency in Practice* для практического знакомства с многопоточностью.

Другая тема, не затронутая в данной книге, — это понятие процесса программирования. Советы относительно "взаимодействия через код" применимы ко времени, когда написание программы подходит к концу после длинного цикла, или к моменту написания теста на ошибки. Всегда лучше иметь недорогое программное обеспечение, чем какие-то ни было социологические исследования, следуя которым оно было написано.

Я также вкратце очертил границы применения Java. Я вынужден быть консервативным в выборе технологии, поскольку я слишком часто терпел фиаско, используя новые возможности на всю катушку (это хорошая стратегия обучения, однако слишком рискованная для большинства разработок). Так что вы найдете только прозаичное подмножество Java. Если вы нацелены на изучение последних нововведений в Java, обратитесь к другим источникам.

Краткое содержание

Как видно на рис. 1.1, книга разделена на семь основных частей.

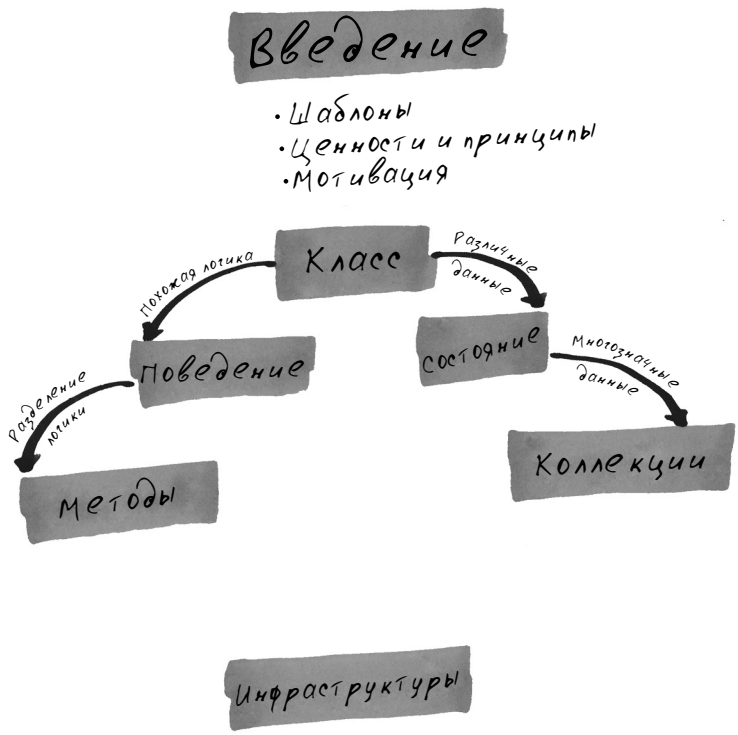


Рис. 1.1. Структура книги

Вступление. Эти короткие главы описывают важность и ценность коммуникации через код и философию, стоящую за шаблонами.

Класс. Шаблоны, описывающие, как и почему вы должны создавать классы, и каким образом в них закодирована логика.

Состояние. Шаблоны для сохранения и получения состояния.

Поведение. Шаблоны, представляющие логику, особенно альтернативные пути.

Метод. Шаблоны для написания методов, напоминающие, что хотелось бы увидеть читателям в результате вашего выбора разбиения на методы и имена.

Коллекции. Шаблоны для выбора и использования коллекций.

Развертывание инфраструктур. Вариации на тему использования шаблонов при построении инфраструктур вместо приложений.

Итак...

...приступим. Если вы читаете книгу с самого начала, всего лишь переверните страницу (я думаю, вы уже вычислили это действие сами). Если же вы хотите просмотреть сами шаблоны, начните с главы 5. Счастливой реализации.

Глава 2

Шаблоны

Многие решения в программировании уникальны. Ваш подход к созданию веб-сайта будет существенно отличаться от подхода к созданию кардиостимулятора. Однако, по мере того как решения становятся более ясными технически, улучшается и понимание вопроса. Разве все, что я написал, — это только код? Программирование станет более эффективным, если программисты будут тратить меньше времени на рутинные, повторяющиеся части своей работы, чтобы иметь возможность более продуктивно решать действительно уникальные проблемы.

Большинство программ подчиняется небольшому набору правил, перечисленных ниже.

- Программы читают чаще, чем пишут.
- Разработка не может быть окончательной. Намного больше вложений занимает не создание новых программ, а модификация старых.
- Код структурирован с помощью базового набора состояний и управляющих методов.
- Читателю необходимо понимать как общую концепцию, так и детали программы. Иногда его внимание переключается на концепцию, иногда на детали.

Шаблоны основаны на этой общности. Например, каждый программист решал когда-либо задачу оформления цикла. К тому времени, когда вы задумаетесь о написании цикла, большинство вопросов общего характера будут уже решены и вам останутся чисто технические проблемы: цикл должен быть легким для чтения, легким для написания, проверки и модификации, а также эффективным.

Приведенный список — начало шаблона. Ограничения или факторы, описанные выше, влияют на каждый написанный цикл в программе. Факторы предсказуемо повторяются и являются основой того, что возникает желание создать шаблон, поскольку это шаблон реакций на повторяющиеся факторы.

Существует несколько разумных способов написать цикл. Каждый из них подразумевает различные приоритеты указанных ограничений. Если для вас важнее производительность, то стоит написать цикл одним способом, тогда как простота модификации может потребовать другого подхода.

Каждый шаблон можно рассматривать с точки зрения относительных приоритетов факторов. Большинство шаблонов сопровождаются краткой заметкой об альтернативных способах решения проблемы и пояснением, почему было выбрано имен-

но такое решение. Раскрытие причины использования какого-то шаблона побуждает читателей находить для себя способы решения повторяющихся задач.

Как было указано выше, каждый шаблон несет в себе элемент решения. Шаблон для использования коллекции в цикле предполагает: "Используйте цикл `for` Java 5, чтобы выразить итерацию". Шаблоны наводят мост между абстрактными принципами и практикой. Шаблоны помогают писать код.

Шаблоны работают вместе. Например, шаблон, предполагающий цикл `for`, одновременно включает в себя и задачу именования переменной цикла. Вместо того чтобы упаковать все в один шаблон, лучше поместить задачу в другой, отдельно оговаривающий именование переменных.

Стиль представления шаблонов в книге сильно варьируется. Иногда они имеют четкое название, с разделами, обсуждающими силы и решения. Некоторые меньшие шаблоны, однако, внедрены в более крупные. Для описания такого маленького шаблона требуется всего одно-два предложения.

Работа по шаблонам может казаться ограничивающей, но их использование сберегает время и энергию. Например, уборка постели требует гораздо меньше энергии, если это делается по привычке, чем если бы нужно было постоянно обдумывать каждый шаг и правильную последовательность. Вы действуете по устоявшемуся шаблону, который значительно упрощает эту рутину. Если кровать находится у стены или простыня слишком маленькая, вы приспособитесь к ситуации, но процедура в целом может быть выполнена механически, а мозг занят другими, более интересными и оригинальными задачами. Когда шаблон стал привычкой, я заметил, что отпала необходимость в спорах только ради написания цикла. Если целая команда разработчиков не удовлетворена шаблоном, то можно обсудить какие-то элементы для построения нового.

Ни один набор шаблонов не будет работать во всех встречающихся ситуациях. Я перечислил здесь шаблоны, хорошо применимые в практике разработки приложений (с беглым отступлением относительно разработки инфраструктур). Слепое копирование чужого стиля не так эффективно, как внедрение и обсуждение своего собственного стиля в команде.

Шаблоны лучше всего работают как дополнение к процессу принятия решений человеком. Некоторые шаблоны реализации в итоге прокладывают себе путь в программировании — например, структуры вида `setjmp()/longjmp()` сегодня стали частью системы обработки исключений. Между тем шаблоны зачастую требуют адаптации перед использованием.

Эта глава началась с обзора более дешевого, быстрого, менее энергоемкого пути решения общих задач программирования. Использование шаблонов помогает программистам находить обоснованные решения различных проблем с учетом влияющих факторов и дает прикладные советы по созданию удовлетворительного решения. Результатом является более качественная/дешевая/быстрая работа над скучными частями программы, поэтому появляется больше времени и энергии для раздумий над более сложными задачами.

В следующей главе, посвященной теории программирования, описываются ценности и принципы, находящиеся в основе данного стиля программирования.

Глава 3

Теория программирования

Любой список шаблонов, даже самый исчерпывающий, не может охватить все ситуации, встречающиеся в программировании. Иногда (или даже чаще всего) вы сталкиваетесь с проблемой, решить которую нельзя ни одним заранее известным способом. Эта необходимость найти новые подходы является причиной для изучения теории программирования. Другая причина заключается в повышении уровня владения предметом, который слагается из понимания того, как именно и что именно делать. Рассуждения о программировании также более интересны, если они затрагивают и теорию и практику.

Каждый шаблон несет в себе частичку теории. Однако существуют и более влиятельные и распространенные силы в программировании, которые невозможно описать в рамках одного шаблона. В данной главе рассказывается об их взаимоотношениях. Они делятся на два типа: ценности и принципы. Ценности универсальны, они образуют свод в здании программирования. Когда я работаю хорошо, я понимаю важность коммуникации с остальными людьми, устраняю излишнюю сложность в коде и оставляю за собой право выбора. Эти ценности — общение, простота и гибкость — влияют на любое решение, которое я принимаю во время программирования.

Описанные здесь принципы не так далеко заходят и не настолько распространены, как ценности, но каждый из них выражен во многих шаблонах. Принципы объединяют ценности, которые универсальны, но не применяются на практике, и шаблоны, которые легко применимы, но весьма специфичны. Я нахожу принципы полезными в ситуациях, когда не обнаруживается подходящего шаблона или же когда одинаково подходят два взаимно исключающих шаблона. При встрече с двусмысленностью понимание принципов позволяет мне “примирить” что-то новое с предыдущим опытом и выйти из ситуации наилучшим образом.

Эти три вещи — ценности, принципы и шаблоны — формируют уравновешенное выражение стиля программирования. Шаблоны описывают действие. Ценности дают мотивацию. Принципы помогают претворить побуждение в действие.

Ценности, принципы и шаблоны вытекают из моей собственной практики, критики и обсуждений с другими программистами. Мы все отталкиваемся от опыта предыдущих поколений. Результатом становится некий стиль разработки, не являющийся стилем разработки. Различные ценности и принципы приводят к различным стилям. Выработанные ценности, принципы и шаблоны делают более легким разрешение производственных конфликтов. Если вы хотите что-то сделать одним способом,

а я — другим, то мы можем определить уровень нашего разногласия и избежать траты времени. Если наши принципы не схожи, то спор о принадлежности фигурных скобок не является первопричиной нашего несогласия.

Ценности

Три ценности, помогающие достичь совершенства в программировании, — это общение, простота и гибкость. В большинстве случаев они дополняют друг друга, хотя иногда могут конфликтовать. Хорошая программа включает много возможностей для будущего расширения, не содержит лишнего и легка для чтения и понимания.

Взаимодействие

Код участвует во взаимодействии, когда пользователь может понять его, изменить и использовать. У программиста во время работы появляется соблазн думать только о компьютере. Однако во время программирования стоит подумать и о других людях. Это позволяет получить более чистый и результативный код, очистить мысли, посмотреть на вещи свежим взглядом, снизить уровень беспокойства и разрешить некоторые социальные вопросы. Я пришел к программированию отчасти потому, что оно давало возможность общения с внешним миром. Но я не хотел иметь дело с неразговорчивыми, непонятными, надоедливыми человеческими существами. Через пару десятков лет я перестал видеть программирование как вид деятельности, не имеющий ничего общего с людьми. Я утратил вкус к построению воздушных замков.

Одним из первых событий, обративших мое внимание на общение, стала книга *Literate Programming* (“Литературное программирование”) Д. Кнута. В ней утверждалось, что программа должна читаться, как книга. Она должна иметь сюжет, ритм и приятные небольшие отступления.

Когда Вард Кённингэм и я впервые прочитали о литературном программировании, мы решили его испробовать. Мы выбрали чистый и понятный фрагмент кода из образа Smalltalk — ScrollController — и попытались расписать его, как историю. Несколько часов спустя код был полностью переписан (в разумном объеме). Гораздо легче переписать часть кода, сложного для понимания, чем объяснять, почему он непонятен. Требования взаимодействия изменили наш взгляд на программирование.

Фокусировка на общении во время программирования имеет прочную экономическую основу. Большая часть вложений в программное обеспечение делается после его введения в работу. Анализируя мой опыт модификации кода, я нашел, что гораздо больше времени было потрачено на чтение кода, чем на написание нового. Поэтому дешевая программа должна быть легко читаема.

Сфокусированность на взаимодействии улучшает образ мышления, делает его более реалистичным. Это происходит потому, что задействуется большая часть мозга, чем обычно. Когда я думаю “Как кто-то другой увидит это?”, включаются другие ней-

роны, чем когда я фокусируюсь только на себе и своем компьютере. Я шагаю в сторону от своей изолированной перспективы и вижу проблему и ее решение по-новому. Другая часть улучшений происходит от осознания того, что я произвожу правильные действия, заботясь о результате. Наконец, для общественно-ориентированного вида определенная направленность на социальные проблемы более реалистична, чем утверждение, что таких проблем не существует.

Простота

В книге *The Visual Display of Quantitative Information* Эдвард Тюфт демонстрирует упражнение, в котором из графа стираются все вершины, не содержащие новой информации. Получившийся граф непохож на оригинал, хотя и намного проще его. Устранение излишней сложности облегчает понимание программ в плане их чтения, использования и модификации. В то же время некоторые усложнения отражают сущность проблемы и являются необходимыми. Однако зачастую сложность отражает следы борьбы, которую мы вели за то, чтобы заставить программу вообще заработать. Эта избыточная сложность обесценивает программное обеспечение, делая его менее стабильным, а внесение изменений — более трудоемким. В программировании полезно переоценивать проделанную работу, чтобы отделить зерна от плевел.

У каждого свое представление о простоте. То, что просто для эксперта, владеющего всеми особенностями ремесла, может казаться невозможным новичку. Так же как и хорошая литература, программа должна иметь свою целевую аудиторию. Иногда можно слегка озадачить свою аудиторию, но чрезмерное усложнение приведет к ее потере.

Компьютерные вычисления проходили периоды сложности и упрощения. Архитектура мэйнфреймов делалась все более и более причудливой, до тех пор, пока не появился персональный компьютер. Его возникновение не решило всех проблем мэйнфреймов, но показало, что для многих приложений эти проблемы не так уж важны. Языки программирования также становились то сложнее, то проще. Язык C породил C++, а тот — Java, который теперь развивается сам по себе.

В погоне за простотой рождаются инновации. Инструмент JUnit вытеснил более сложные инструменты тестирования. В результате появилось множество клонов, дополнений и новых технологий программирования/тестирования. Последний релиз, JUnit 4, уже не оставляет такого “кондового” ощущения, хотя я участвовал в принятии каждого усложняющего решения. Когда-то кто-нибудь откроет намного более легкий путь тестирования, чем с использованием JUnit. Новая идея приносит очередную волну инноваций.

Применяйте упрощение ко всем уровням. Форматируйте код так, чтобы все его части несли какую-то информацию. Проектируйте без лишних элементов. Меняйте требования и оставляйте только действительно необходимые. Вычеркивание лишнего кода подчеркивает оставшийся, предоставляя шанс заново оценить его.

Коммуникация и упрощение часто работают вместе. Чем проще система, тем легче ее понять. Чем больше вы фокусируетесь на взаимодействии, тем легче увидеть, какие усложнения могут быть отброшены. Однако, если я вижу, что упрощение мешает пониманию, приоритет отдается коммуникации. Эти случаи редки и обычно указывают на более глубокий, недоступный мне уровень упрощения.

Гибкость

Из трех упомянутых ценностей гибкость служит оправданием для самых неэффективных конструкторских и программистских решений. Чтобы получить значение константы, программы обращаются к переменной окружения, содержащей имя каталога, в котором находится файл с необходимой константой. К чему такие сложности? Ради гибкости. Программы должны быть гибкими, но только в способах своего изменения. Если константа никогда не меняется, то все эти сложности бессмысленны.

Код должен легко меняться, так как большая часть вложений осуществляется после запуска программы в производство. Гибкость, которая, по моим представлениям, будет необходима завтра, все же, не то, что нужно, когда меняется код. Именно поэтому гибкость более эффективна в целях упрощения и всестороннего тестирования, чем в области теоретического проектирования.

Выбирайте наиболее гибкие шаблоны, которые дают мгновенные результаты. Терпеливость является лучшей стратегией при использовании очевидных, но не дающих немедленной прибыли шаблонов, — лучше отложить их до тех пор, пока они не понадобятся. Впоследствии, при случае, их можно будет применить в точности так, как они того требуют.

Ценой гибкости может стать усложнение. Например, пользовательские опции конфигурации обеспечивают гибкость, но увеличивают сложность файла конфигурации и требуют постоянно обращать на себя внимание программиста. Простота может дополнять гибкость — если вы найдете способ убрать конфигурируемые опции, сохранив ценность программы, то это позволит легче изменять ее в будущем.

Расширение коммуникабельности программного обеспечения также повышает его гибкость. Чем больше людей смогут читать, понимать и изменять код, тем больше возможностей дальнейшей модификации получает ваша организация.

Приведенные здесь шаблоны достаточно гибки, чтобы помочь программистам создавать простые, понятные приложения, которые могут быть легко изменены.

Принципы

Шаблоны реализации не используются “потому, что так надо”. Каждый из них выражает одну из ценностей коммуникации, простоты или гибкости. Принципы — это следующий уровень общих идей, имеющий больше отношения к программированию, чем ценности. Принципы также являются основой шаблонов.

Изучение принципов имеет смысл по нескольким причинам. Ясные и понятные принципы могут порождать новые шаблоны, так же как периодическая таблица Менделеева привела к открытию новых элементов. Принципы раскрывают причины использования того или иного шаблона в более общем виде, чем какие-то конкретные идеи. Выбор из нескольких противоречащих друг другу шаблонов часто лучше обсуждать в терминах принципов, чем непосредственной специфики ситуации. Наконец, понимание принципов позволяет лучше находить решения при столкновении с новой проблемой.

Например, когда я сталкиваюсь с новым языком программирования, мое понимание принципов позволяет развить эффективный стиль программирования. Я не копирую уже существующие стили или, что хуже, не цепляюсь за мой стиль программирования на другом языке (к примеру, на любом языке можно писать, как на FORTRAN, но лучше этого не делать). Понимание принципов дает возможность быстро учиться и действовать более собранно в необычных ситуациях. Рассмотрим эти принципы перед тем, как перейти к шаблонам реализации.

Локализация последствий

Структурируйте код так, чтобы изменения имели только локальные последствия. Если изменение *здесь* приводит к изменению *там*, то его себестоимость существенно возрастает. Код с локализованными последствиями эффективен в коммуникации. Этот принцип может быть усвоен постепенно, без необходимости охватывать его сразу же целиком.

Поскольку поддержание низкой цены вносимых изменений выступает первичной мотивацией многих шаблонов реализации, то принцип локализации последствий является частью рассуждений, стоящих за многими шаблонами.

Минимизация повторений

Принцип, который вносит свой вклад в локализацию последствий, — минимизация повторений. Если в программе имеется несколько копий одного и того же кода и вы изменили одну из них, вам приходится решать, менять ли все остальные. Ваше изменение перестает быть локальным. Чем больше копий кода, тем больше цена изменений.

Одинаковый код — только одна из форм повторения. Параллельные иерархии классов также повторяются и нарушают принцип локализации. Если одно концептуальное изменение требует вмешательства в две или более иерархии классов, тогда оно имеет слишком далеко идущие последствия. Переделка кода таким образом, чтобы изменения снова стали локальными, улучшает код.

Копирование не всегда очевидно до того, как оно будет сделано, и может никак не проявляться в дальнейшем. Хотя оно может быть замечено, но не всегда находится правильный способ его устранения. Копирование само по себе не есть зло, оно всего лишь повышает стоимость изменений.

Один из способов борьбы с копированием — разбиение программы на множество мелких кусочков: небольших состояний, методов, объектов и пакетов. Большой кусок логики имеет тенденцию включать в себя части других таких же кусков. Эта общность и делает возможными шаблоны — в то время как между частями кода есть различия, они также имеют много сходств. Внятные сведения о том, какие части программы идентичны, какие достаточно похожи, а какие полностью различны, делают программы более читаемыми и легко модифицируемыми.

Объединение логики и данных

Следующий принцип, вытекающий из принципа локализации последствий, — это объединение логики и данных. Располагайте их рядом, в одном методе, объекте или, по крайней мере, в одном пакете, если это возможно. При модификации кода логика и данные должны изменяться одновременно. Если они сведены вместе, то последствия внесения изменений будут также локальными.

Поначалу общность логики и данных может быть неочевидна. Я могу написать код А и понять, что мне нужны данные из кода Б. Только после получения работоспособного кода я осознал, что он располагается слишком далеко от данных. В таком случае мне приходится делать выбор: переместить код к данным, переместить данные к коду, объединить их во вспомогательном объекте или же отложить эту задачу на время, так как в данный момент я не могу эффективно выстроить их взаимодействие.

Симметрия

Другой постоянно используемый принцип — это симметрия. Программы изобилуют ею. Метод `add()` сопровождается методом `remove()`. Группе методов передаются одни и те же параметры. Все поля объекта имеют одно время жизни. Распознание и выражение симметрии делают код более читаемым. Как только читатель осваивает одну половину симметрии, он тут же понимает и вторую.

Симметрия часто описывается в пространственных терминах: двусторонняя, ротационная и т.п. В программах она выражается в концепции, редко графически. Симметрия кода — это выражение той же идеи тем же способом, как и в остальном коде.

Вот пример кода, которому недостает симметрии:

```
void process() {
  input();
  count++;
  output();
}
```

Вторая строчка более конкретна, чем два сообщения. Я бы переписал это с точки зрения симметрии вот так:

```
void process() {
  input();
  incrementCount();
  output();
}
```

Тем не менее и в этом методе ее недостаточно. Операции `input()` и `output()` названы в соответствии с их целями, `incrementCount()` — по реализации. В поисках симметрии я подумал о том, *почему* здесь инкрементируется `count`, и получил следующее:

```
void process() {
  input();
  tally();
  output();
}
```

Зачастую поиск и выделение симметрии предваряет процесс устранения повторений. Если одна и та же мысль присутствует в разных частях кода, то придание ему симметрии становится первым шагом к унификации.

Описательные выражения

Другой принцип, лежащий в основе шаблонов реализации, — это передача своих замыслов описательным способом. Императивное программирование — мощный и гибкий инструмент, но чтение программы, написанной в этом стиле, требует следовать за нитью исполнения. Приходится строить умозрительную модель состояния программы, процесса и данных. Те же части программы, являющие "вещь в себе", без последовательностей или условий, содержат более простой, описательный код.

Например, в предыдущих версиях JUnit классы могли иметь статический метод `suite()`, возвращающий набор запускаемых тестов:

```
public static junit.framework.Test suite() {
Test result= new TestSuite();
...сложные вычисления...
return result;
}
```

Теперь задайте себе простой вопрос: какие тесты будут запущены? В большинстве случаев метод `suite()` только собирает вместе тесты из ветви классов. Однако, так как это общий метод, мне нужно будет проанализировать код, чтобы обрести уверенность.

С другой стороны, JUnit 4 использует принцип описательных выражений, чтобы решить ту же проблему. Вместо метода, возвращающего набор тестов, вводится метод, запускающий тесты на наборе классов (общий случай):

```
@RunWith(Suite.class)
@TestClasses({
  SimpleTest.class,
  ComplicatedTest.class
})
class AllTests {
}
```

Если известно, что тесты обрабатываются этим методом, то все, что нужно, чтобы узнать, какие именно тесты будут запущены, — это посмотреть на объявление `TestClasses`. Так как данный набор объявлен описательным выражением, нет нужды предполагать наличие в нем каких-то подводных камней. Это решение обеспечивает универсальность и мощь оригинального метода `suite()`, но описательный стиль делает код более читаемым. Объявление `RunWith` предоставляет даже более гибкий механизм для запуска тестов, но это тема для отдельной книги.

Частота изменений

Последний принцип заключается в объединении логики или данных, которые изменяются с одинаковой частотой, и их разделении, если частота изменений разная. Частота изменений является формой временной симметрии. Иногда этот принцип относится к изменениям, вносимым программистом. К примеру, если я пишу бухгалтерское программное обеспечение, то я разделяю код, ответственный за общие вычисления, и код, актуальный только для данного года. Код меняется с разной частотой. Когда я вношу изменения для следующего года, мне нужно быть уверенным, что код предыдущего года еще работает. Их разделение обеспечивает локализацию последствий от вносимых изменений.

Частота изменений применима к данным. Грубо говоря, все поля в объекте должны изменяться с одной частотой. Например, поля, изменяемые только однажды во время активации, должны быть объявлены как локальные переменные. Два поля, которые меняются синхронно между собой, но не одновременно со своими соседями, вполне могут относиться к вспомогательному объекту. Если значения стоимости и оборота в финансовом инструменте могут меняться вместе, то их, вероятно, лучше выразить в виде вспомогательного объекта `Money`:

```
setAmount(int value, String currency) {
this.value= value;
this.currency= currency;
}
```

становится

```
setAmount(int value, String currency) {
  this.value= new Money(value, currency);
}
```

а потом в

```
setAmount(Money value) {
this.value= value;
}
```

Частота изменений — это приложение симметрии, хотя и временной. В приведенном примере два оригинальных поля, `value` и `currency`, симметричны. Они меняются одновременно. Однако они не симметричны относительно других полей объекта. Выражение симметричности путем помещения их в отдельный объект передает их взаимоотношение читателям и, скорее всего, предоставляет дальнейшие возможности для устранения дублирования и локализации последствий.

Выводы

Эта глава является введением в теоретические основы шаблонов реализации. Ценности общения, простоты и гибкости отлично обосновывают использование шаблонов. Принципы локализации последствий, минимизации повторений, объединения логики и данных, симметрии, описательных выражений и частоты изменений помогают привести ценности в действие. Теперь мы перейдем к шаблонам — особенным решениям для повторяющихся в программировании тактических проблем.

В следующей главе, посвященной мотивации, описываются экономические факторы, обосновывающие ценность взаимодействия через код.

Глава 4

Мотивация

Тридцать лет назад Йордон и Константин (Yourdon and Constantine) в книге *Structured Design* определили экономику как двигатель проектирования программ. ПО должно быть спроектировано так, чтобы уменьшилась его общая стоимость. Она делится на начальную стоимость и стоимость сопровождения:

$$\text{стоимость}_{\text{общая}} = \text{стоимость}_{\text{разработки}} + \text{стоимость}_{\text{сопровождения}}$$

Когда в индустрии был накоплен достаточный опыт разработки ПО, для многих стал сюрпризом тот факт, что начальная стоимость гораздо ниже стоимости сопровождения. (Проекты, не предполагающие сопровождения, должны использовать совершенно другой набор шаблонов реализации, нежели приведенный в данной книге.)

Сопровождение обходится дорого, поскольку понимание кода — это процесс, занимающий много времени и подверженный ошибкам. Внесение изменений существенно облегчается, если известно, что именно требуется изменить. Изучение существующего кода является самой трудоемкой частью. После внесения изменений они должны быть протестированы и внедрены.

$$\text{стоимость}_{\text{сопровождения}} = \text{стоимость}_{\text{понимания}} + \text{стоимость}_{\text{изменений}} + \\ + \text{стоимость}_{\text{тестирования}} + \text{стоимость}_{\text{поставки}}$$

Одной из стратегий уменьшения общей стоимости являются большие вложения в начальную разработку, в надежде уменьшить стоимость поддержки или же вовсе избавиться от ее необходимости. Но, если код будет непредвиденно меняться, не имеет значения, сколько вы потратите на его подготовку к изменениям. Необдуманные попытки обобщения кода с целью обеспечить будущие нужды, часто пересекаются с непредвиденными изменениями, которые оказываются необходимыми.

Увеличение только лишь авансовых затрат на ПО противоречит двум важным экономическим принципам: временной стоимости денег и неопределенности будущего. Если сегодняшний доллар стоит больше завтрашнего, то применяющаяся стратегия должна в целом учитывать возможные затраты. Также в этой стратегии непосредственная выгода должна иметь больший приоритет, чем долгосрочная. Хотя это и звучит призывом бросаться в бой сломя голову, но шаблоны реализации концентрируются на способах получения немедленной выгоды так же, как и на чистоте кода ради возможности будущего развития.

Моя стратегия уменьшения общей стоимости заключается в уменьшении стоимости понимания кода в фазе поддержки. Для этого я обращаю внимание программистов на важность общения, взаимодействия друг с другом. Эта стратегия приносит и краткосрочные выгоды — уменьшение количества дефектов, легкость использования общих библиотек, постепенность разработки.

Как только набор шаблонов реализации становится привычным, я программирую быстрее и меньше отвлекаюсь. Когда я начинал писать первый набор шаблонов (*The Smalltalk Best Practice Patterns*, Prentice Hall, 1996), мне казалось, что я опытный программист. Чтобы сосредоточиться на шаблонах, я решил, что не напечатаю ни одного символа кода, пока соответствующий шаблон не будет написан. Это было ужасно, как если бы мои пальцы были склеены. Поначалу каждая минута кодирования предварялась часом записывания шаблонов. Через неделю я обнаружил, что многие основные шаблоны уже записаны и бо́льшую часть времени я следую им. К исходу третьей недели я программировал гораздо быстрее, чем когда-либо до этого, так как я пересмотрел собственный стиль и испытывал меньше сомнений.

Записанные здесь шаблоны лишь частично являются моим изобретением. Так, стиль написания был в основном скопирован у предыдущих поколений программистов. У них были привычки, результатом которых стал код, который легко читать, понимать и изменять, а копирование этих привычек помогло мне самому писать более быстро и гладко. Я смог подготовиться к будущему и создавать код быстрее за то же время. При написании шаблонов реализации для этой книги я вдохновлялся существующим кодом и своими собственными привычками. Я читал и сравнивал код из JDK, Eclipse и моих предыдущих разработок. Целью получившихся шаблонов является вразумительное объяснение того, как писать код, понятный другим людям. Другие точки зрения или исходные данные дали бы другие шаблоны. Например, я выделил шаблоны разработки инфраструктур в отдельную главу. Они основаны на других приоритетах и поэтому отличаются от моего стиля реализации.

Шаблоны служат человеку так, как того требует экономика. Код разрабатывается людьми и для людей. Программисты могут использовать шаблоны реализации для удовлетворения человеческих потребностей, таких, как необходимость гордиться хорошо сделанной работой или быть почетным членом общества. В последующих главах шаблоны будут обсуждаться с точки зрения экономических и человеческих факторов.

Глава 5

Класс

Идея классов возникла задолго до Платона. Его идеальные тела были классами, воплощения которых существовали в реальном мире. Сфера Платона была абсолютно идеальна, но нематериальна. Вокруг нас есть много сфер, но все они не идеальны в какой-то мере.

Объектно-ориентированное программирование взяло эту идею в изложении поздних западных философов. Программы поделили на классы, которые являются общим описанием целого множества похожих вещей, или объектов.

Классы важны для взаимодействия, поскольку они могут описывать многие специфические вещи. Они охватывают широкий диапазон шаблонов реализации, в то время как шаблоны проектирования обычно относятся к взаимоотношениям классов.

В этой главе появятся следующие шаблоны.

- Класс. Используется, чтобы сказать: “Эти данные и логика есть одно целое”.
- Простое имя суперкласса. Наименование корневого класса иерархии простым именем, производным от метафоры.
- Специальное имя субкласса. Наименование субклассов для обозначения сходств и различий с суперклассом.
- Абстрактный интерфейс. Разделяет интерфейс и имплементацию.
- Интерфейс. Определяет абстрактный, редко меняющийся интерфейс с помощью Java-интерфейса.
- Интерфейс версии. Безопасно расширяет интерфейс до субинтерфейса.
- Абстрактный класс. Описывает абстрактный интерфейс (который, возможно, будет меняться) с помощью абстрактного класса.
- Объект-значение. Объект, который ведет себя как математическое значение.
- Специализация. Разграничивает сходства и различия взаимосвязанных вычислений.
- Субкласс. Выражает одномерную вариацию в субклассе.
- Имплементор. Переопределяет метод для другого варианта вычислений.
- Вложенный класс. Заключает локально используемый код в приватный класс.

- Контекстно-зависимое поведение. Варьирует логику в зависимости от воплощения.
- Условный шаблон. Меняет логику в соответствии с определенными условиями.
- Делегирование. Логика изменяется путем делегирования одному из нескольких типов объектов.
- Сменный селектор. Изменения логики отражаются при выполнении метода.
- Анонимный вложенный класс. Меняет логику, переопределяя один или два метода при создании объекта.
- Библиотечный класс. Представляет функциональный пакет, не подходящий ни одному объекту, в качестве набора статических методов.

Класс

Данные более подвержены изменениям, чем логика. Классы работают именно благодаря этому. Каждый из них есть декларация: "Логика — одно целое и меняется медленнее, чем данные, которыми она оперирует. Данные тоже объединены, меняются с одной частотой и используются соответствующей логикой". Строгое разделение между изменчивыми данными и не меняющейся логикой не является абсолютным. Иногда логика слегка модифицируется в зависимости от данных; иногда изменения очень глубоки. Иногда данные не меняются в процессе вычислений. Изучение того, как объединять логику в классы и выражать вариации в логике, — это часть эффективного объектного программирования.

Организация иерархии классов — это способ сжатия кода путем объявления суперкласса и текстового включения его в субклассы. Как и любая технология сжатия, она делает код плохо читаемым. Приходится вникать в контекст суперкласса, чтобы понять субклассы.

Другой аспект эффективного программирования объектов — разумное использование наследования. Создание субкласса выражается фразой: "Я такой же, как и суперкласс, только другой". (Не странно ли то, что мы говорим о *переопределении* метода в *субклассе*? Насколько лучшими программистами мы можем стать, если будем внимательно выбирать метафоры?)

Классы — относительно дорогой элемент проектирования объектно-ориентированных программ. Класс должен делать что-то значительное. Уменьшение числа классов в системе — это улучшение, если только оставшиеся классы не раздуваются от этого.

Приведенные далее шаблоны поясняют, как взаимодействовать с помощью классов.

Простое имя суперкласса

Само нахождение правильных имен — один из самых приятных элементов программирования. Вы боретесь с идеей. Часто код получается сложным и выглядит не так, как должно. А иногда, напротив, кто-то видит программу, округляет глаза и говорит: "Да, точно! Это действительно `Scheduler`!" Правильное имя является результатом последовательных упрощений и улучшений.

Имена классов являются наиболее важными. Класс — это центральная концепция, "якорь" проектирования. Как только у класса возникает имя, появляются имена и у всех остальных операций. Редко случается обратное, разве что класс был изначально плохо назван.

При именовании возникает конфликт между краткостью и выразительностью. Например, в беседе можно выразить имя класса так: "Вы помните, что нужно повернуть `Figure` перед ее смещением?" Имена должны быть короткими и точными. Однако часто приходится использовать несколько слов, чтобы дать соответствующее имя.

Разрешить эту дилемму можно, выбирая подходящую характеру вычислений метафору. С этим мощным средством каждое слово приносит богатую сеть ассоциаций, взаимосвязей и смысловых линий. Например, в инфраструктуре HotDraw мое первое имя для объекта рисования было `DrawingObject`. Вард Куннингэм (Ward Cunningham) предложил типографскую метафору: рисование — это как печать страницы. Графические элементы на странице — фигуры, так что класс стал называться `Figure`. В контексте метафоры `Figure` — сравнительно более короткое, точное и богатое имя, чем `DrawingObject`.

Порой поиск хорошего имени занимает много времени. Законченный код может работать недели, месяцы или (как было в одном случае со мной) годы, пока не отыщется лучшее имя для класса. Иногда приходится порядочно потрудиться для поиска имени: достать словарь, сделать список наиболее подходящих имен, выйти на прогулку. В некоторых случаях нужно продвигаться вперед, доверяя времени, и ваше подсознание подскажет лучшее имя.

Общение — инструмент, постоянно помогающий мне в поиске имен. Разъяснение сути объекта другому человеку приводит меня к ряду богатых и запоминающихся образов, которые могут быть преобразованы в новые имена.

Для наименования важных классов лучше использовать одно слово.

Специальное имя субкласса

Имена субклассов имеют два назначения. Они должны рассказывать, на что *похож* класс и от чего *отличен*. Опять же необходимо соблюдение баланса между длиной и выразительностью. В отличие от корневых классов иерархии, имена субклассов используются не так часто, поэтому их можно делать длиннее в ущерб лаконичности.

Для формирования имени субкласса можно добавить приставку к названию суперкласса.

Здесь есть единственное исключение: если субклассы используются строго для реализации механизма разделения, имеют в основе свою собственную, отдельную концепцию и находятся во главе собственной иерархии, то они могут получать более короткие имена. Например, HotDraw содержит класс `Handle`, предоставляющий операции редактирования фигуры, когда она выбрана. Называя класс `Handle`, а не расширяя имя от `Figure`, мы подчеркиваем, что может быть целое семейство классов `Handle` — например, `StretchyHandle` или `TransparencyHandle`. Таким образом, `Handle` стоит во главе собственной иерархии классов и поэтому имеет простое название суперкласса, а не специальное имя субкласса.

Другое препятствие в наименовании субклассов — многоуровневые иерархии. Часто они только ждут своего появления, но когда это случается, классам требуются хорошие имена. Вместо того чтобы слепо модифицировать непосредственный суперкласс приставками, подумайте об этом с точки зрения читателя. Какие классы он должен счесть похожими? Используйте соответствующий суперкласс в качестве основы для имени.

Назначение имен классов — во взаимодействии с людьми. Для нужд компьютера классы могут быть просто пронумерованы. Имена классов слишком длинны и сложны для чтения и форматирования. Слишком короткие имена утомляют краткосрочную память читателя. Кластеры классов с не относящимися друг к другу именами трудно осмыслить и вспомнить. Используйте имена классов, чтобы поведать историю вашего кода.

Абстрактный интерфейс

Старая поговорка о разработке ПО гласит: программируют для интерфейса, а не ради затеи. Это другая сторона предположения о том, что проектные решения не должны зримо присутствовать там, где в этом нет нужды. Если бо́льшая часть кода знает только лишь, что я работаю с коллекцией, то мне легко будет сменить конкретный класс позже. Однако на каком-то этапе вам действительно понадобится подключить конкретный класс, чтобы выполнились вычисления.

Под интерфейсом я подразумеваю здесь набор операций без реализации. Это может быть представлено в Java как интерфейсом, так и суперклассом. К различным случаям подходят соответствующие шаблоны.

Каждый слой интерфейса имеет стоимость, так как является еще одной вещью, которую нужно выучить, понять, документировать, отладить, организовать, осмотреть и наименовать. Максимизация числа интерфейсов не снижает стоимости ПО. Платите за интерфейсы, только когда вам нужна присущая им гибкость. В то время как зачастую вы можете не знать наперед, когда появится нужда в интерфейсах, комбинируйте предположения о том, где гибкость не нужна, а где ее следует ввести.

Мы очень много жалуемся на негибкость ПО, но существует множество вещей, где мы не нуждаемся ни в какой системе гибкости. Со времени фундаментальных изменений, таких как количество битов в integer, до сегодняшних широкомасштабных начинаний в виде новых моделей бизнеса большинство ПО, как правило, не нуждается в гибкости.

Другой экономический фактор, говорящий в пользу интерфейсов, — это непредсказуемость рынка ПО. Наша индустрия кажется вовлеченной в идею о том, что если мы правильно спроектировали программу, то аппаратная часть систем не должна меняться. Я недавно прочитал список причин модификации ПО. В нем были программисты, плохо делающие свою работу по предъявленным требованиям, изменчивость спонсоров и т.п. Единственный фактор, отсутствовавший в списке, — это закономерные изменения. Список предполагает, что изменение — всегда ошибка. Почему бы не сделать предсказание погоды на все времена? Потому что она меняется непредсказуемо. В таком случае мы можем условиться раз и навсегда, что система должна быть гибкой, так как изменения требований и технологии непредсказуемы. Это не освобождает нас от обязанности наилучшим образом удовлетворять нужды сегодняшних потребителей, но устанавливает какие-то умственные ограничения на ценность "защищенного от будущего" ПО.

Сведение всех этих факторов вместе — необходимость гибкости, ее цена, непредсказуемость того, где она понадобится, — заставляют меня верить, что вводить ее надо только в случае явной нужды. При этом вам придется заплатить за изменения в существующем ПО. Если же вы не можете этого сделать лично, то стоимость возрастет пропорционально, как описывается ниже в главе, посвященной развертыванию инфраструктур.

Механизмы Java — интерфейсы и суперклассы — имеют неодинаковую себестоимость использования.

Интерфейс

Один из способов сказать: "Эту часть кода я хотел бы завершить, хотя детали пока что меня не волнуют", — объявить интерфейс в Java. Интерфейсы — важное нововведение, впервые появившееся на массовом рынке программных языков именно в Java. Они имеют хороший баланс, предлагая гибкость множественного наследования, не отягощенную сложностью и неопределенностью. Класс может быть объявлен как разделяющий множество интерфейсов. Также они раскрывают только операции, а не поля, что эффективно защищает пользователей интерфейса от изменений реализации.

Когда же изменение касается реализации, то сам по себе интерфейс остается прежним. Любая добавка или модификация в интерфейсе требует изменения всех воплощений. Если вы не можете менять воплощения, то широкое применение интерфейсов становится существенным тормозом в развитии проекта.

Одна особенность интерфейсов, ограничивающая их способность к взаимодействию, — это объявление всех методов публичными. Я часто хотел, чтобы область видимости операций интерфейса оставалась в рамках пакета. Создание чуть более, чем необходимо, публичных элементов проектирования не так важно для личного использования, но если интерфейс предполагается широко обнародовать, то лучше быть точным сначала, чем встраивать в код инерцию к будущим изменениям.

В зависимости от способа мышления есть два стиля наименования интерфейсов. Если это классы без имплементации, то их так и следует именовать: простое имя суперкласса, специальное имя субкласса. Единственная проблема в этом — использование хороших имен до того, как вы перейдете к определению собственно классов. Интерфейс `File` должен иметь имплементации, названные примерно как `ActualFile`, `ConcreteFile` или (внимание!) `FileImpl` (одновременно и суффикс, и сокращение). Вообще-то, существенно знать, с чем происходит взаимодействие — с абстрактным или конкретным объектом, — тогда как способ реализации объекта в виде интерфейса или суперкласса не столь важен. Данный способ именования вносит различия в названия интерфейсов и суперклассов, оставляя вам выбор в будущем на случай необходимости.

Иногда скрыть использование интерфейса не так важно для взаимодействия, как назвать конкретный класс. В таком случае к имени интерфейса добавляйте префикс I. Если интерфейс будет назван `IFile`, то класс можно назвать просто `File`.

Абстрактный класс

Другой способ выразить различие между абстрактным интерфейсом и конкретной реализацией в Java — это использовать суперклассы. Они абстрактны в том смысле, что имплементация всегда может быть заменена субклассом, независимо от применения ключевого слова `abstract`.

Решение об использовании абстрактного класса вместо интерфейса сводится к двум вопросам: изменения в интерфейсе и нужда в едином классе для одновременной поддержки множества интерфейсов. Абстрактные интерфейсы могут изменяться дважды — в имплементации и сами по себе как интерфейсы. Последнее — не лучшая черта Java. Любая модификация интерфейса требует изменения имплементаций. С повсеместно реализованным интерфейсом это запросто приводит к "параличу" существующих проектов, в дальнейшем развивающихся только в версиях интерфейсов.

Абстрактные классы не страдают от этого ограничения. Когда обозначена базовая реализация, в абстрактный класс могут быть добавлены новые операции без разрушения имплементаций.

Единственный недостаток этого метода состоит в том, что классы могут принадлежать только одному суперклассу. Если необходим другой вид тех же классов, то следует использовать интерфейсы.

Появление слова `abstract` в объявлении класса говорит читателю о том, что ему предстоит создать реализацию, прежде чем он сможет применить данный класс. Если только есть шанс сделать корневой класс полезным и завершенным самим по себе — делайте это. Однажды шагнув на путь абстракций, можно зайти слишком далеко и создать кучу бесполезного кода. Борьба за воплощение корневых классов приведет к устранению массивных абстракций.

Интерфейсы и классы не взаимоисключают друг друга. Можно создать интерфейс, сказав: "Вот путь доступа к этому виду функциональности", и суперкласс — как способ воплощения той же функциональности. В этом случае переменные должны быть объявлены как интерфейсы, чтобы разработчики в будущем могли менять имплементации как им заблагорассудится.

Интерфейс версий

Что вы делаете, когда возникает необходимость изменить интерфейс, хотя это невозможно? Обычно такая ситуация возникает при добавлении новых операций. Так как внесение новой операции поломает все существующие имплементации, это сложно реализовать. Однако можно объявить дополнительный интерфейс, расширяющий текущий. Пользователи, нуждающиеся в новой функциональности, могут использовать новый интерфейс, тогда как остальные просто не обратят внимания на его существование. Тогда везде, при обращении к новой операции, необходимо проверять тип объекта и приводить к новому типу.

Например, обсудим конструкцию

```
interface Command {
void run();
}
```

Допустим, что этот интерфейс используется тысячи раз и его изменение довольно дорого. Однако вам понадобилась операция отмены команд. Тогда получится следующая версия интерфейса.

```
interface ReversibleCommand extends Command {
void undo();
}
```

Существующее воплощение `Command` работает, как и прежде. Воплощения `ReversibleCommand` так же функциональны, как и `Command`. Для использования новой операции нужно привести тип.

```
...
Command recent= ...;
if (recent instanceof ReversibleCommand) {
ReversibleCommand downcasted= (ReversibleCommand) recent;
```

```
downcasted.undo();
]
...
```

Использование конструкции `instanceof` в целом снижает гибкость за счет привязывания кода к определенным классам. Однако в этом случае она может быть оправдана, так как применяется с целью развития интерфейса. Если у вас появляется несколько альтернативных интерфейсов, то клиентам требуется много работы, чтобы учесть все вариации. Это сигнализирует о том, что надо подумать над конструкцией в целом.

Альтернативные интерфейсы — неправильное решение неверно поставленных проблем. Интерфейсы не приспосабливаются к изменениям в своей структуре с той же легкостью, как это делают их имплементации, хотя и стремятся к модификации, как все конструкторские решения. Мы изучили все о проектировании имплементаций и поддержке. Альтернативные интерфейсы создают новый язык, похожий на Java, но с другими правилами. Написание своего языка — игра с более жесткими ограничениями, чем создание приложений. Однако, если вы столкнетесь с ситуацией, когда необходимо будет расширить интерфейс, лучше знать, как это сделать.

Объект-значение

Объекты с меняющимся состоянием — не единственный способ размышлять о вычислениях. Математика развивалась более тысячелетия как способ осмысления ситуаций, сводящихся к абстрактному миру абсолютной правды и определенности, где состояния могут быть выражены через вечные истины.

Наш текущий язык программирования — смесь обоих стилей. Так называемые примитивы в Java (в большинстве случаев) принадлежат к миру математики. Когда я прибавляю единицу к числу, я произвожу математическое сложение (кроме того случая, когда кто-то решает, что компьютер может считать только до 2^{32} или до 2^{64}, и все приходится начинать заново). Я не меняю значения переменной, когда прибавляю 1, — я создаю новое значение. Не существует способа изменить 0, как это возможно с большинством объектов.

Этот функциональный стиль программирования никогда не меняет состояний — он создает новые значения. Когда появляется (может быть, на секунду) статичная ситуация, которую нужно сформулировать или задать ей вопросы, то функциональный стиль отлично подходит. Если ситуация нестабильна во времени, то и состояние меняется соответственно. Некоторые вещи могут быть описаны обоими способами. Как решить, какой из них больше подходит конкретному случаю?

Например, рисование картины можно представить как изменение состояния некоего графического средства, такого, как точечный рисунок. Одновременно ту же картину можно описать статическим методом (рис. 5.1).

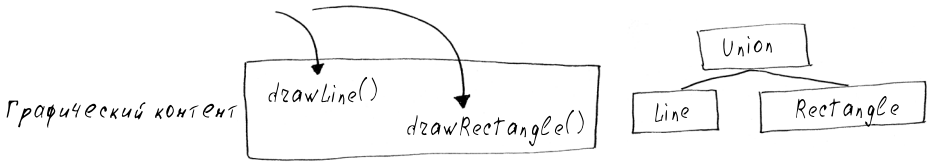


Рис. 5.1. Графика, представленная процедурами и объектами

Любая репрезентация применима в какой-то степени на основании собственного опыта, но также имеет значение и сложность картины, и скорость ее изменения.

Процедурные интерфейсы более универсальны, чем функциональные. Их единственный недостаток — необходимость выполнять вызовы процедур в определенной последовательности, что становится частично сутью интерфейса. Изменение таких программ часто рискованно и затруднено, так как сравнительно небольшие модификации имеют нежелательные последствия, если меняют внутреннюю сущность последовательности.

Математическое представление более красиво, оно заставляет последовательность что-то значить саму по себе и позволяет создавать мир, в котором будут только абсолютные, вневременные состояния. Вводите математические микромиры где только возможно. Управление может осуществляться в объекте с переменным состоянием.

Например, реализуем систему расчетов, используя базовые транзакции без изменения математических значений (рис. 5.2).

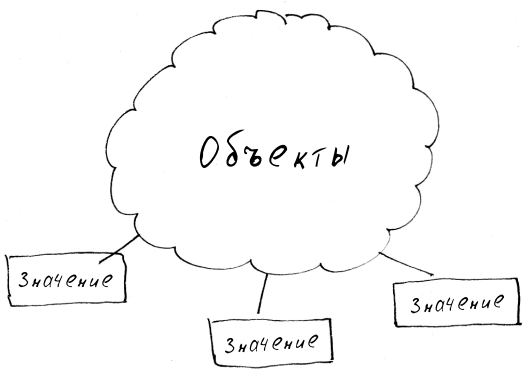


Рис. 5.2. Объекты с переменным состоянием, ссылающиеся на неизменные объекты по краям

```
class Transaction {
int value;
Transaction(int value, Account credit, Account debit) {
this.value= value;
credit.addCredit(this);
debit.addDebit(this);
```

```
}
int getValue() {
return value;
}
}
```

С момента создания объекта `Transaction` не существует способа изменить его значение. Более того, конструктор делает предположение, что все транзакции относятся к двум счетам. Во время прочтения этого кода не возникнет беспокойства о "зависших", неучтенных транзакциях, или о транзакциях, изменивших значение после объявления.

Для реализации объектов в стиле "значение" (т.е. объектов, ведущих себя как integer, в отличие от объектов-контейнеров, содержащих текущее состояние) проведите для начала границу между миром состояний и миром значений. В приведенном примере `Transaction` является примером значения, тогда как `Accounts` — контейнер состояния. В объектах-значениях все состояния устанавливаются в конструкторах, не оставляя возможности изменить их после. Операции на таких объектах всегда возвращают новые объекты, которые должны быть сохранены вызывающей стороной.

```
bounds.translateBy(10, 20); // mutable Rectangle
bounds=bounds.translateBy(10, 20); // value-style Rectangle
```

Самым существенным возражением против этого подхода всегда была производительность. Необходимость создавать все эти промежуточные объекты может создать нагрузку на память управляющей системы. В программировании в целом этот факт обычно не учитывается, поскольку большинство частей программы не являются узкими местами. Другая трудность использования значений — это непонимание стиля и сложность проведения четких границ между системами с изменяющимся состоянием и статичными системами. Объекты-значения часто плохи для обоих случаев, тогда как интерфейсы более совершенны, хотя бывает сложно делать предположения об их состоянии.

Забираясь в такие дебри, я лишь хочу сказать, что гораздо больше должно быть написано об этих трех стилях программирования — объектах, функциях и процедурах — и об их эффективном совместном употреблении. Но, учитывая назначение этой книги, я закругляюсь, еще раз напомнив, что программы лучше всего выражать как комбинацию объектов-значений и объектов-состояний.

Специализация

Выражение в программе сходств и различий вычислений делает ваши программы более легкими для чтения, использования и модификации. Практически не существует уникальных программ. Многие из них выражают одни идеи, как делают зачастую и разные части одной программы. Передача пользователю этих сходств и различий

помогает понять код, найти соответствие между замыслами пользователя и существующими вариациями, или, если соответствий нет, приспособить написанный код к своим нуждам или написать что-то совершенно новое.

Простейшие вариации — отличия состояний. Строка “abc” отличается от “def”. Алгоритмы, оперирующие на этих строках, идентичны. Например, длина строк вычисляется одним и тем же способом.

Наиболее сложные вариации — различия в логике. Процедура символического интегрирования не имеет ничего общего с процедурой вывода математического текста, хотя они могут принимать одни и те же входные данные.

Между двумя крайностями — идентичной логикой с различающимися данными и различной логикой с одинаковыми данными — лежит пространство обычного программирования. Данные могут быть в основном похожими, хотя и немного различными. То же самое относится и к логике. (Я думаю, процедуры символического интегрирования и вывода математического текста имеют все же небольшую общую часть.) Размыта даже линия между логикой и данными. Булев флаг — это данные, но он может управлять исполнением. Вспомогательный объект может храниться в поле, но влиять на вычисления.

Нижеприведенные шаблоны являют собой техники, указывающие преимущественно на логические сходства и различия. Вариации данных не кажутся сложными или запутанными. Эффективное выражение сходств и различий в логике открывает новые возможности для дальнейшего расширения кода.

Субкласс

Объявляя субкласс, мы говорим: “Эти объекты совсем как те, за исключением...” Если у вас есть правильный суперкласс, то создание субкласса — это мощное средство программирования. Переопределяя нужный метод, вы можете создавать вариации всего в нескольких линиях кода.

Когда объекты только стали популярными, субклассы казались волшебной пилюлей. Вначале они использовались для классификации — `Train` наследовался от `Vehicle` независимо от реального разделения имплементации. Со временем некоторые люди заметили, что наследование позволяет разделять имплементацию и может быть использовано, чтобы вынести за скобки ее общие части. Однако вскоре стали заметны ограничения наследования. Во-первых, этой картой можно сыграть только раз. Если обнаруживается, что какой-то набор вариаций недостаточно хорошо выражается субклассами, то приходится поработать над распутыванием кода перед его реструктурированием. Во-вторых, необходимо изучить суперкласс до того, как вы сможете понять субкласс. Чем более сложен суперкласс, тем больше неудобств это доставляет. В-третьих, изменения суперкласса рискованны, так как его использование в субклассах может быть незаметным. Наконец, все эти проблемы усугубляются в глубоких иерархиях наследования.

Частично разрушительный эффект наследования проявляется в параллельных иерархиях, когда каждому субклассу в *этой* иерархии соответствует класс в *той* иерархии. Это одна из форм дублирования, подразумевающая взаимное копирование. Для успешного внедрения новой вариации приходится менять обе иерархии. Хотя не всегда есть способ немедленно устранить такую ситуацию, попытки сделать это улучшают конструкцию.

Рис. 5.3. Параллельные иерархии

Примером может служить страховая система (рис. 5.3). В этой картине что-то определенно не так, потому что `InsuranceContract` никак не касается `PensionContract`, как бы ни было привлекательно сместить поле результата вниз в субклассы. Потребовался год, чтобы прийти к решению (так и не реализованному). Оно заключалось в смещении вариации таким образом, чтобы `Contract` работал одинаково и для страховки, и для пенсии. Это потребовало создания нового объекта для представления ожидаемых движений наличности (рис. 5.4).

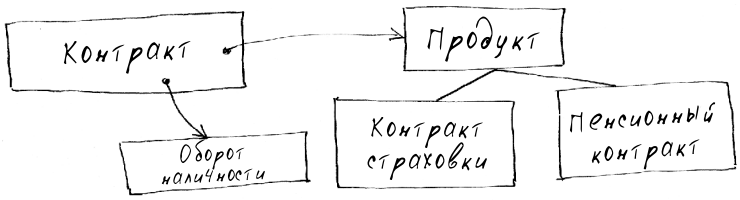


Рис. 5.4. Иерархия без дубликатов

Не упуская из виду все эти предосторожности, субклассы можно сделать мощным средством для выражения тематических и вариативных вычислений. Правильный субкласс помогает описать в точности то, что надо, в одном-двух методах. Один из аспектов использования субклассов — тщательное планирование логики методов суперкласса, делающих одну работу. Впоследствии должна быть возможность переопределить хотя бы один метод. Если методы суперкласса слишком велики, можно скопировать код и отредактировать его (рис. 5.5).

Скопированный код представляет нежелательное, но подразумеваемое дублирование между двумя классами. Невозможно безопасно поменять код в суперклассе без проверки и потенциального изменения всех скопированных мест в субклассе.

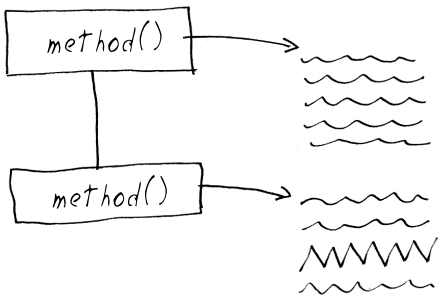


Рис. 5.5. Код, скопированный и модифицированный в субклассе

Своей целью в проектировании я ставлю способность переключаться между стратегиями по желанию, в зависимости от нужд кода, если он существует к тому моменту. Визуализируйте код с помощью условий, субклассов и делегирования. Разве вам не кажется, что есть преимущества и в другой стратегии, отличной от используемой вами сейчас? Сделайте несколько шагов в этом направлении и посмотрите, не улучшился ли код.

Последнее ограничение субклассов — невозможность передать меняющуюся логику. Вариация должна быть известна вам до создания объекта и не может быть изменена потом. Чтобы выразить такую ситуацию, используйте условия и делегирование.

Имплементор

Полиморфное сообщение — это фундаментальный способ выразить выбор в программе, построенной из объектов. Чтобы сообщение сделало свою работу выбора, должно быть более одного потенциального объекта-получателя.

Имплементация одного протокола несколько раз с помощью Java-интерфейса, или объявления `implements`, или субкласса, выраженного через `extends`, — это способ сказать: "Если произошло что-то, соответствующее замыслу данного кода, то с его точки зрения детали происшедшего несущественны".

Сила полиморфных сообщений в том, что они открывают путь вариациям. Если часть программы передает несколько байтов в другую систему, то внедрение абстракции типа `Socket` заставляет имплементацию сокета варьировать поведение, не влияя на вызывающий код. В сравнении с процедурной реализацией того же замысла, с ее однозначной и закрытой условной логикой, версия объект/сообщение более ясна, она отделяет выражение замысла (записать несколько байтов) от имплементации (вызов TCP/IP стека с определенными параметрами). В то же время выражение вычислений через объекты и сообщения открывает систему к таким будущим вариациям, о каких даже не мечтали первоначально программисты. Эта неожиданная комбинация ясности выражения и гибкости и делает объектные языки доминирующей парадигмой программирования.

Этот превосходный ресурс легко истощить, делая процедурные программы на Java. Шаблоны созданы для того, чтобы помочь вам выражать одновременно ясную и способную к расширению логику.

Вложенный класс

Иногда требуется упаковать часть вычислений, но не хочется делать этого ценой создания нового класса с отдельным файлом. Объявление маленьких, приватных (вложенных) классов дает дешевый способ использовать многие преимущества отдельного класса.

Иногда вложенный класс расширяет только `Object`. Некоторые вложенные классы расширяют другие суперклассы, что целесообразно для выражения усовершенствований, полезных лишь локально.

Одна из отличительных черт вложенного класса заключается в том, что при создании воплощения ему негласно передается копия создающего объекта. Это удобно, если по какой-либо причине нужно получить доступ к внешнему классу без явной регламентации взаимоотношений двух объектов.

```
public class InnerClassExample {
  private String field;

  public class Inner {
    public String example() {
    return field; // Используется переменная field
                  //  экземпляра внешнего класса
  }

@Test public void passes() {
  field= "abc";
  Inner bar= new Inner();
  assertEquals("abc", bar.example());
}
}
```

Таким образом, конструктор в приведенном примере все-таки принимает аргумент, хотя в объявлении его и нет. Это становится проблемой при создании воплощений вложенных классов по образу внешнего.

```
public class InnerClassExample {
  public class Inner {
   public Inner() {
   }
  }

 @Test(expected=NoSuchMethodException.class)
  public void innerHasNoNoArgConstructor() throws Exception {
```

```
  Inner.class.getConstructor(new Class[0]);
}
}
```

Чтобы получить вложенный класс, полностью независимый от обрамляющего воплощения, объявляйте его как `static`.

Контекстно-зависимое поведение

Теоретически все воплощения класса оперируют одной логикой. Устранение этого ограничения дает нам новые стили выражения. Однако все они недешево стоят. Когда логика кода полностью определяется классом, читатели могут посмотреть на класс и понять, что произойдет. Как только появляются воплощения с разным поведением, становится необходимо изучать работающие примеры или анализировать поток данных, чтобы понять, как поведет себя конкретный объект.

Следующий шаг обходится еще дороже. Речь идет об изменении логики по ходу работы программы. Для облегчения понимания старайтесь устанавливать контекстно-зависимое поведение сразу же после создания объекта и после уже не менять его.

Условия

Простейшими формами контекстно-зависимого поведения являются операторы `if/then` и `switch`. Используя условия, различные объекты выполняют разные задачи, основываясь на входных данных. Условия имеют то преимущество, что вся логика остается в одном классе. Читателям не приходится искать возможные пути исполнения программы. Однако есть и недостатки этого метода. Один из них — невозможность модификации поведения без изменений кода проблемного объекта.

Каждый путь выполнения программы может быть правильным с какой-то вероятностью. Таким образом, чем больше независимых путей, тем более ошибочна может быть программа в целом. Конечно, ошибки не полностью независимы, но все же в достаточной степени, чтобы программа с большим количеством путей имела больше дефектов. Умножение условий уменьшает надежность.

Эта проблема усложняется, когда условия дублируются. Обсудим простой графический редактор. Фигуры должны иметь метод `display()`.

```
public void display() {
  switch (getType()) {
  case RECTANGLE :
    //...
    break;
  case OVAL :
    //...
    break;
```

```
  case TEXT :
    //...
    break;
  default :
    break;
  }
}
```

Также нужен метод, определяющий принадлежность точки фигуре.

```
public boolean contains(Point p) {
  switch (getType()) {
    case RECTANGLE :
      //...
      break;
    case OVAL :
      //...
      break;
    case TEXT :
      //...
      break;
    default :
      break;
  }
}
```

Теперь предположим, что в программе добавился еще один вид фигур. Во-первых, вам нужно ввести новый пункт во всех блоках `switch`. Во-вторых, чтобы произвести это изменение, придется затронуть класс `Figure`, рискуя существующей функциональностью. Наконец, каждый, кто захочет добавлять новые фигуры, должен координировать изменения в единственном классе.

Все эти проблемы могут быть устранены путем преобразования условной логики в сообщения, используя субклассы или делегирование (какую технику лучше применить, зависит от кода). Дублированная условная логика, или же логика, в которой обработка сильно отличается в разных ветвях условий, лучше выражается сообщениями, чем однозначным указанием. То же относится к часто меняющейся условной логике — использование сообщений упрощает изменение одной ветви, сводя к минимуму влияние на другие (рис. 5.6).

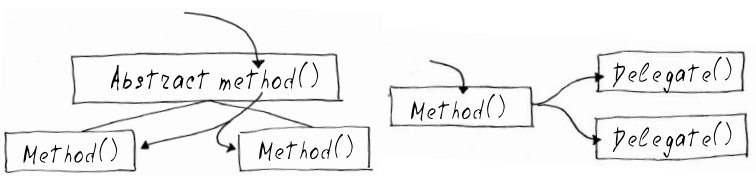


Рис. 5.6. Условная логика, представленная субклассами и делегированием

Таким образом, преимущества условий — простота и локальность — обращаются в недостатки при слишком широком использовании.

Делегирование

Другой способ исполнять различную логику в различных воплощениях — передача работы нескольким возможным видам объектов. Общая логика содержится в направляющем классе, вариации делегируются.

Пример использования делегирования для перехвата вариации — обработка пользовательского ввода в графическом редакторе. Иногда щелчок на кнопке означает создание прямоугольника, иногда — перемещение фигуры и т.п.

Один из способов сделать это заключается в применении условной логики.

```
public void mouseDown() {
  switch (getTool()) {
    case SELECTING :
      //...
      break;
    case CREATING_RECTANGLE :
      //...
      break;
    case EDITING_TEXT :
      //...
      break;
    default :
      break;
  }
}
```

Этот подход имеет все недостатки, обсуждаемые ранее: добавление нового инструмента требует модификации кода, а дублирование условий (в `mouseUp()`, `mouseMove()` и т.д.) усложняет эту задачу.

Субклассы также не являются непосредственным ответом, так как редактор должен менять инструменты во время работы. Делегирование предоставляет необходимую возможность.

```
public void mouseDown() {
  getTool().mouseDown();
}
```

Код, который существовал в пунктах блока `switch`, был перемещен в различные инструменты. Теперь новые инструменты могут быть представлены без модификации кода редактора или существующих инструментов, хотя чтение кода требует больше перемещений, так как логика нажатия кнопки распределена между несколькими классами. Понимание того, как ведет себя редактор в данном случае, требует понимания используемого инструмента.

Делегаты могут храниться в полях (“сменный объект”), но могут и вычисляться на лету. JUnit 4 динамически вычисляет объекты, использующиеся для проведения теста над данным классом. В зависимости от стиля теста (старый или новый) создаются различные делегаты. Это смесь условной логики (для создания делегатов) и собственно делегирования.

Тот же подход может использоваться для разделения кода в качестве контекстно-зависимого поведения. Объект, обращающийся к `Stream`, может быть вовлечен в контекстно-зависимое поведение, если тип `Stream` меняется во время исполнения, или может разделять имплементацию `Stream` со всеми остальными пользователями.

Обычный трюк — передача делегирующего объекта в качестве параметра делегируемому.

```
GraphicEditor
public void mouseDown() {
  tool.mouseDown(this);
}
RectangleTool
public void mouseDown(GraphicEditor editor) {
  editor.add(new RectangleFigure());
}
```

Если делегат хочет послать сообщение самому себе, то это допускает двоякое толкование. Иногда сообщение должно быть послано делегирующему объекту, иногда — делегируемому. В приведенном примере `RectangleTool` добавляет фигуру, но к делегирующему `GraphicsEditor`, а не к самому себе. `GraphicsEditor` может быть послан как параметр делегируемому элементу `mouseDown()`, но в этом случае проще сохранить постоянную ссылку в инструменте. Передача `GraphicsEditor` параметром делает возможным использование одного инструмента в различных редакторах, но, если это не критично, проще использовать код по обратной ссылке.

```
GraphicEditor
public void mouseDown() {
  tool.mouseDown();
}
RectangleTool
private GraphicEditor editor;
  public RectangleTool(GraphicEditor editor) {
  this.editor= editor;
}
public void mouseDown() {
  editor.add(new RectangleFigure());
}
```

Сменный селектор

Давайте предположим, что нам нужно контекстно-зависимое поведение, но только в одном-двух методах, и включение всех вариаций кода в один класс не требуется. В этом случае сохраняйте имя необходимого метода в поле и вызывайте его по мере надобности.

Изначально каждый тест из JUnit сохраняется в своем собственном классе (рис. 5.7). Каждый субкласс имеет только один метод. Такой подход смотрится концептуально тяжелым для представления единственного класса.

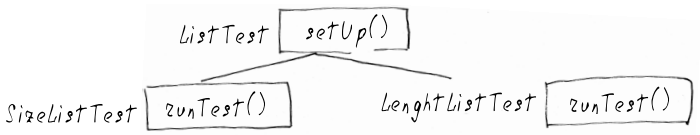


Рис. 5.7. Небольшие субклассы для представления одного теста

Имплементируя базовый метод `runTest()`, `ListTests` запускает разные методы тестов с различными именами. Предполагается, что имя теста совпадает с именем вызываемого метода во время работы теста. Ниже приведена простая версия кода, воплощающего сменный селектор исполняемого теста.

```
String name;
public void runTest() throws Exception {
   Class[] noArguments= new Class[0];
   Method method= getClass().getMethod(name, noArguments);
   method.invoke(this, new Object[0]);
}
```

Упрощенная иерархия использует единственный класс (рис. 5.8). Как происходит со всеми техниками сжатия кода, этот модифицированный код легко читается, только если вы понимаете “фишку”.

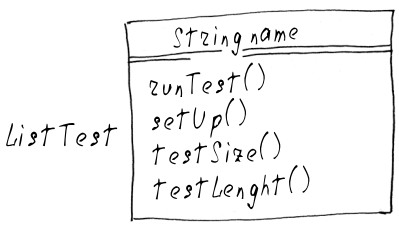


Рис. 5.8. Сменный селектор помогает упаковать тесты в один класс

Когда сменные селекторы появились в обращении, их стали использовать слишком часто. Вы будете смотреть на какой-то код, решив, что он не может быть испол-

нен, и получите останов системы, так как этот фрагмент был где-то вызван сменным селектором. Цена использования этой технологии может быть огромной, но ограниченное использование для решения сложных проблем оправдывает эту цену.

Анонимный вложенный класс

Java предлагает еще одну альтернативу контекстно-специфическому поведению — анонимные вложенные классы. Идея состоит в том, чтобы создать используемый только в одном месте класс, который может переопределять один или более методов строго для локальных целей. По этой причине ссылки на класс могут неявными, в отличие от ссылок по имени.

Эффективное использование анонимных вложенных классов предполагает применение сверхупрощенного API — как имплементация `Runnable` в единственном методе `run()` — или внедрение суперкласса, который обеспечит всю необходимую имплементацию. Код анонимного вложенного класса разрушает репрезентацию обрамляющего кода, поэтому он должен быть коротким, чтобы не отвлекать читателя.

Анонимные вложенные классы имеют ограничение — код воплощения должен быть известен заранее (в отличие от делегатов, которые могут добавляться позже) и не может изменяться после его создания. Такие классы трудно протестировать напрямую, поэтому они не должны вмещать сложную логику. Так как они безымянны, вы не имеете возможности выразить замысел с помощью хорошо подобранного имени.

Библиотечный класс

Куда вы поместите функциональность, не подходящую ни одному объекту? Можно создать статические методы в классе, не содержащем ничего, кроме этих методов. Никто никогда и не будет намереваться создавать воплощения этого класса — он послужит лишь контейнером для библиотечных статических функций.

Хотя такие решения довольно распространены, они плохо масштабируемы. Приходится расплачиваться за помещение всей логики в статические методы наибольшим преимуществом объектного программирования: упрощение кода путем разделения данных приватного пространства имен. Старайтесь превратить библиотечные классы в объекты, где это только возможно.

Иногда это легко, если очевиден собственник метода. Например, библиотечный класс `Collections` имеет метод `sort(List)`. Специфика параметра подсказывает нам, что этот метод, вероятно, лучше приписать классу `List`.

Постепенный путь преобразования библиотечного класса в объект — конвертирование статических методов в методы воплощения. Для начала оставьте тот же интерфейс, используя статические методы для передачи управления воплощению, как делается, например, в классе

```
public static void method(...params...) {
...some logic...
}
```

и преобразуйте в

```
public static void method(...params...) {
  new Library().instanceMethod(...params...);
}
private void instanceMethod(...params...) {
...some logic...
}
```

Теперь, если несколько методов имеют одинаковый список параметров (и если их принадлежность одному классу явно выражена), преобразуйте параметры методов в параметры конструктора.

```
public static void method(...params...) {
  new Library(...params...).instanceMethod();
}
private void instanceMethod() {
...some logic...
}
```

Потом измените интерфейс, переместив создание воплощения в клиентскую часть и уничтожив статические методы.

```
public void instanceMethod(...params...) {
...some logic...
}
```

Этот опыт дает представление о том, как переименовывать классы и методы, чтобы код был хорошо читаемым.

Выводы

Класс хранит вместе связанные между собой состояния. Следующая глава представляет шаблоны, передающие информацию о состоянии.

Глава 6

Состояние

Шаблоны, приведенные в этой главе, показывают, как сделать информативным ваше использование состояний. Обычно объекты представляют собой пакеты *поведения*, которое предназначено для внешнего мира, и *состояния*, которое позволяет реализовать это поведение. Одно из преимуществ объектов заключается в том, что они разделяют общее состояние программы на крошечные фрагменты, каждый из которых сам по себе представляет небольшой компьютер. Большие библиотеки состояний с беспорядочными ссылками могут затруднить дальнейшую модификацию кода, поскольку влияние перемен в коде сложно прогнозировать. С помощью объектов проще сказать, какие изменения произойдут в коде в результате перемен, поскольку пространство имен для состояний, на которые можно ссылаться, значительно меньше.

В этой главе описаны следующие шаблоны.

- Состояние. Вычисления со значениями, которые меняются во времени.
- Доступ. Управление гибкостью для ограничения доступа к состоянию.
- Прямой доступ. Прямой доступ к состоянию внутри объекта.
- Косвенный доступ. Доступ к состоянию через метод для обеспечения большей гибкости.
- Общее состояние. Содержит состояние, общее для всех объектов класса в виде полей.
- Переменное состояние. Хранит состояние, представление которого варьируется от экземпляра к экземпляру, как ассоциативный массив (`map`).
- Внешнее состояние. Хранит состояние специального назначения, связанное с объектом, в ассоциативном массиве, принадлежащем пользователю объекта.
- Переменная. Переменные предоставляют пространство имен для хранения состояния.
- Локальная переменная. Хранит данные для одной области видимости.
- Поле. Хранит состояние в процессе жизненного цикла объекта.
- Параметр. Взаимодействует с состоянием в процессе активизации отдельного метода.
- Собирающий параметр. Передает параметр для сборки сложного результата нескольких методов.

- Объект-параметр. Объединяет часто используемый длинный список параметров в объект.
- Константа. Хранит состояние, которое является неизменным, как константа.
- Имя, указывающее на роль. Наименование переменных с учетом роли, которую они играют в вычислениях.
- Определение типа. Определение общего типа для переменных.
- Инициализация. Декларативная инициализация переменных там, где это возможно.
- Ранняя инициализация. Инициализируйте поля во время создания экземпляра.
- Отложенная инициализация. Инициализируйте поля, значения которых сложно вычислять, перед самым их использованием.

Состояние

Мир постоянен. Если минуту тому назад Солнце было высоко в небе, то вы можете быть уверены, что оно по-прежнему там, хотя и немного сдвинулось. Если меня заинтересуют точные вычисления, то я могу предсказать новое положение на основе моих предыдущих наблюдений, знания скорости вращения Земли и измерений прохождения светила с течением времени.

Думать о мире как об изменяющихся вещах — это определенно полезный подход во временной перспективе. Коренные американцы, живущие по соседству со мной, весной смотрят на гору Маклафлин. Если снег растаял достаточно, чтобы контур летящего орла был различим на фоне оставшегося снега, значит, наступило время спускаться к реке Родж, чтобы застать весенний ход лосося. Состояние снега на горе было удивительным образом связано с наличием плавающего деликатеса в весьма удаленной реке.

Когда пионеры вычислений выбирали метафоры для программирования компьютеров, их просто заклинило на этой идее изменяющегося со временем состояния. Человеческий мозг располагал множеством стратегий, встроенных и приобретенных, чтобы работать с этим состоянием.

Однако состояние также таит и проблемы для программистов. Как только вы начинаете отталкиваться от того, что собой представляет некоторый кусок состояния, — ваш код в опасности. Вы можете сделать неверное предположение, или какое-то состояние может измениться без вашего ведома. Многие полезные программные утилиты, такие как инструменты для автоматического рефакторинга, лучше всего конструировать без предположений о состоянии.

Наконец, параллелизм и состояние не очень-то хорошо уживаются друг с другом. Многие из проблем параллельных вычислений отпадают сами собой, если мы не говорим о состоянии.

Языки функционального программирования вообще обходятся без изменения состояния. И это становится все более популярным. Я считаю, что состояние — полезная для нас метафора, поскольку наш мозг структурирован и ориентирован на изменяющиеся состояния. Одно присвоение или программирование без переменных принуждает нас отбросить слишком многие из эффективных стратегий мышления, чтобы быть привлекательным выбором.

Объектные языки являются копирующей стратегией для работы с состояниями. Они дают шанс избежать проблемы изменения состояний "за нашей спиной", разделяя состояние системы на маленькие дискретные фрагменты, каждый из которых имеет явно ограниченный доступ к другим. Проще отслеживать несколько байтов, чем мегабайты или гигабайты. Проблема в неверном предположении, что некоторое значение по-прежнему неизменно. Но объекты дают вам шанс быстро и аккуратно проследить все случаи доступа к переменной.

Ключ к эффективному управлению состоянием — располагать одинаковые состояния вместе и убедиться, что разные состояния содержатся раздельно. Две подсказки о том, что два фрагмента состояния похожи, — это факт, что два фрагмента используются в одном и том же вычислении, и то, что они создаются и разрушаются в одно и то же время. Если два куска состояния используются вместе и имеют одно и то же время жизни, то расположить их поближе один к другому, вероятно, является хорошей идеей.

Доступ

Один из "водоразделов" в языках программирования — различие между доступом к сохраненным значениям и вызовом вычислений. Две концепции, каждую из которых можно определить в терминах другой. Доступ к памяти напоминает вызов функции, которая возвращает сохраненное ранее значение. Вызов функции похож на чтение места в памяти, содержимое которого только что вычислено, но еще не возвращено. Тем не менее наши языки программирования разделяют вызов вычислений и доступ к памяти, так что нам нужно эффективно сглаживать это различие.

Читабельность, гибкость и производительность программ зависят от того, что вы храните или вычисляете. Иногда эти цели конфликтуют одна с другой, а также с вашими предпочтениями как программиста. Иногда изменяется контекст, так что ваш сегодняшний способ разделения сохраненных и вычисленных значений завтра уже может не иметь смысла. Принимать рабочие решения сегодня и закладывать гибкость для будущих изменений — ключ к разработке хорошего ПО. Это та самая необходимость будущих изменений, которая делает ваш компромисс "хранение против вычислений" таким насущным.

Одно из предназначений объектов — управление хранением. Каждый объект выступает как отдельный компьютер, со своей памятью, до некоторой степени изолиро-

ванный от других маленьких компьютеров. Современные языки, включая Java, размывают границы между объектами, вводя публичные поля. Простота межобъектного доступа не стоит потери независимости между объектами.

Прямой доступ

Самый простой способ сказать: "Я получаю данные" или "Я сохраняю данные" — это использовать прямой доступ к переменной.

```
X=10;
```

Прямой доступ к переменной имеет преимущество ясности выражения. Когда я вижу `X=10`, то знаю, что именно сейчас произойдет. Эта ясность сопровождается потерей гибкости. Если я храню значение в переменной, то это все, что я могу сделать. Если я храню значение в этой же переменной во многих частях программы, а потом меняю ее, то мои изменения должны коснуться всех этих частей программы.

Другой недостаток прямого доступа к переменной — это то, что данная деталь реализации ниже уровня, на котором я мыслю в момент программирования. Установка переменной в значение 1 может привести к открытию дверей моего гаража, но код, который отражает эту деталь реализации, не будет хорошо понятен. Сравните

```
doorRegister= 1;
```

и

```
openDoor();
```

или с использованием объектов

```
door.open();
```

Большинство мыслей, которые приходят ко мне во время программирования, не имеют ничего общего с хранением. Широкое распространение прямого доступа ухудшает взаимодействие. Для тех частей программы, где я действительно думаю, что и где хранится, я использую прямой доступ для согласования этих понятий. Решения по хранению играют различную роль для различных программистов, так что нет единой политики для использования прямого доступа, который бы подходил всем. Люди пытаются сформулировать какие-то правила: прямой доступ только внутри методов доступа и, возможно, также внутри конструктора. Прямой доступ только внутри класса, или внутри класса и всех его субклассов, или только внутри одного пакета. Нет универсального правила. Программистам нужно думать, общаться и учиться. Это часть того, чтобы быть профессионалом.

Косвенный доступ

Вы можете скрыть доступ и изменения к состоянию за вызовом методов. Эти способы доступа дают гибкость ценой ясности и прямолинейности. Клиенты более не предполагают, что некоторое значение хранится непосредственно. Таким образом, вы можете изменить ваше мнение о хранении, не влияя на клиентский код.

Моя стратегия по умолчанию для состояния доступа — позволять прямой доступ внутри класса (включая вложенные классы) и косвенный доступ для клиентов. Эта стратегия имеет преимущество, поскольку обеспечивает прямой и понятный доступ к состоянию в большинстве случаев. Заметьте: если большинство обращений к состоянию объекта происходит вне объекта, то за этим кроется более глубокая архитектурная проблема.

Другая стратегия заключается в исключительном использовании косвенного доступа. Я нахожу, что это заканчивается потерей ясности. Большинство методов чтения и записи тривиальны. Они часто превышают по числу методы, которые делают полезную работу, делая код сложным для чтения. Все эти методы чтения и записи также очень притягательны. Вместо того чтобы вычислять там, где происходят расчеты, часто бывает удобно реализовать их где-нибудь и использовать метод доступа для доставки нужного состояния, чтобы сделать это работающим.

Один случай, определенно подходящий для косвенного доступа, — это когда пара значений связана друг с другом. Иногда эта связь очень непосредственная, как в случае кэшированного значения.

```
Rectangle void setWidth(int width) {
  this.width= width;
  area= width * height;
}
```

Иногда связь менее прямая, как в случае с обработчиком.

```
Widget void setBorder(int width) {
  this.width= width;
  notifyListeners();
}
```

Такие взаимосвязи непривлекательны (легко забыть о поддержке предполагаемых ограничений), но могут оказаться лучшим из возможных вариантов. В таком случае косвенный доступ тоже является лучшим.

Общее состояние

Многие вычисления разделяют одни и те же элементы данных, даже если значения и различаются. Когда вы находите такие вычисления, связывайте их, определяя поля в классе. Например, все вычисления в декартовой системе координат требуют значе-

ний абсциссы и ординаты. Поскольку всем декартовым точкам одинаково присущи эти величины, они вернее всего подходят на роль полей.

```
class Point {
  int x;
  int y;
}
```

Противопоставьте эту технологию переменному состоянию, когда объекты одного класса потенциально могут иметь различные элементы данных. Преимущество общего состояния в том, что из чтения кода ясно, из самих ли полей или из описания конструктора, какие данные нужны для образования хорошо сформированного объекта. Ваш читатель захочет узнать, что значит удачно вызвать функциональность вашего объекта. Общее состояние сообщает об этом ясно и точно.

Общее состояние в объекте должно иметь одно и то же время жизни и область видимости. Иногда я поддаюсь искушению ввести поле, которое используется только в подмножестве методов объекта, или даже имеет значение во время выполнения одного метода. В таких случаях я несомненно могу улучшить качество моего кода, если найду другое место, где бы я мог хранить эти данные, — возможно, параметр или вспомогательный объект.

Переменное состояние

Иногда один и тот же объект требует различных элементов данных, в зависимости от того, как он используется. Это не то же, что просто отличающиеся значения: в объектах одного и того же класса могут присутствовать абсолютно другие элементы.

Переменное состояние часто хранится в виде ассоциативного массива, ключи которого являются именами элементов (представленные как строки или перечисления), а значения являются значениями данных.

```
class FlexibleObject {
  Map<String, Object> properties= new HashMap<String, Object>();
  Object getProperty(String key) {
  return properties.get(key);
  }
  void setProperty(String key, Object value) {
    properties.set(key, value);
  }
}
```

Переменное состояние значительно более гибкое, чем общее состояние. Его главная беда в том, что оно слабо взаимодействует. Какие данные нужно предоставить объекту с только переменным состоянием, чтобы он работал нужным образом? Только внимательное прочтение кода и, возможно, наблюдение за выполнением помогут ответить на этот вопрос.

Я изучал код, где программист слишком часто использовал переменное состояние. Каждый объект данного класса имел те же ключи в массиве свойств. Для меня было бы значительно проще читать код, если бы эти данные были представлены как декларации полей.

Один случай, когда переменное состояние кажется оправданным, — это когда состояние одного поля предполагает необходимость в других разных полях. Например, если наш элемент управления имеет флаг `bordered` и если этот флаг установлен, то я также хотел бы иметь `borderWidth` и `borderColor`. Я могу связать это с переменным состоянием, как показано на рис. 6.1, *сверху*.

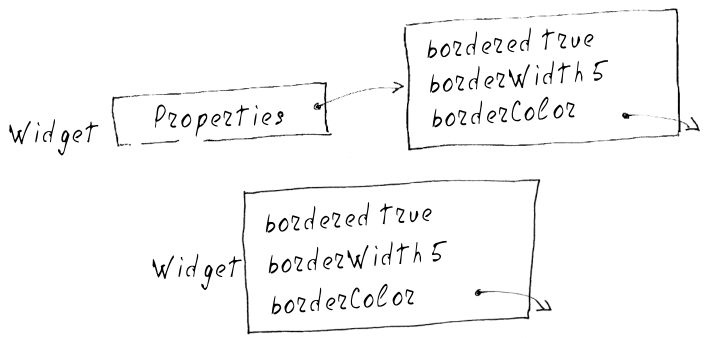


Рис. 6.1. Граница элемента управления, описанная через переменное и общее состояние

Общее состояние также может описать эту ситуацию, как на рис. 6.1, *снизу*.

Решение с общим состоянием нарушает принцип, что все переменные объекта должны иметь одно и то же время жизни. Полиморфизм дает нам еще более простое объяснение этой ситуации. Один класс представляет состояние “окно без границы”, а другой — элемент с “окно с обрамляющей границей”. `Bordered` имеет общее состояние для представления этих параметров (рис. 6.2.).

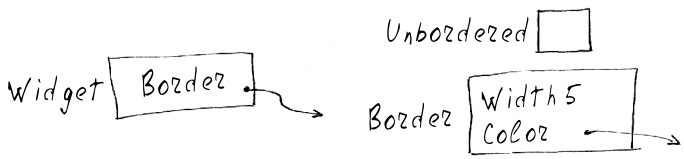


Рис. 6.2. Вспомогательный объект проясняет замысел

Наличие нескольких переменных, которые разделяют общий префикс, — это может быть намеком на то, что может оказаться полезным вспомогательный объект того или иного рода.

Используйте общее состояние, насколько это возможно. Используйте переменное состояние для полей объекта, которые могут как понадобиться, так и нет, в процессе работы.

Внешнее состояние

Иногда часть вашей программы требует некоторое состояние, ассоциированное с вашим объектом, но остальная часть системы — нет. Например, информация о том, где на диске хранится объект, полезна для механизма устойчивости, но не для остального кода. Размещение этих данных в поле будет нарушать принцип симметрии. Все остальные поля полезны для всей системы.

Храните информацию специального назначения, ассоциированную с объектом, неподалеку от того места, где она будет использоваться, а не в самом объекте. В только что рассмотренном примере механизм устойчивости может хранить ассоциативный массив `IdentityMap`, ключи которого представляют собой хранимые объекты, а значения — место, где они хранятся.

Одна слабость внешнего состояния в том, что оно делает сложным копирование объекта. Репликация объекта с внешним состоянием не такая простая, как репликация его полей. Вместо этого все внешнее состояние должно быть также корректно скопировано, что может потребовать различной обработки, в зависимости от того, как это состояние используется. Другая слабость — это сложность отладки объекта с внешним состоянием. Обычный инструмент наблюдения не будет отображать все данные, ассоциированные с объектом. Из-за этих сложностей внешнее состояние редко применяется, однако иногда бывает полезным.

Переменная

В Java на объекты ссылаются через переменные. Читатели кода должны знать об области видимости, времени жизни, роли и типе времени выполнения для переменной. Пока разрабатываются схемы, которые позволят закладывать всю эту информацию в имя переменной, желательно упрощать код насколько возможно, давая переменным простые имена.

Область видимости переменных, отрезок кода, на протяжении которого на них можно ссылаться, может быть трех типов: локальные переменные, доступ только из текущей области видимости; поля могут быть доступны из любой части объекта; статические переменные могут быть доступны из любого объекта заданного класса. Область видимости полей может быть расширена с помощью модификаторов `public`, `package` (по умолчанию, хотя не является лучшим выбором, поскольку используется меньше всего), `protected` и `private`.

Если вы свободно пользуетесь всеми возможными комбинациями, то для читателей будет важно, чтобы вы делали ясные различия в момент ссылки, используя область видимости в имени переменной. Однако для снижения зависимостей вы должны в основном пользоваться локальными переменными и полями, и только иногда — статическими полями и модификатором `private`. Используя этот ограниченный набор возможных комбинаций, читателю достаточно контекста, чтобы по-

нять, видит ли он поле или локальную переменную. Это исключает необходимость включать информацию об области видимости в имя переменной. Взамен вы получите код с унифицированными, просто читаемыми именами. Для этих целей вы можете разбить ваш код на маленькие фрагменты, свойство, которого вы можете достичь, применяя другие шаблоны реализации, в частности методы композиции.

Время жизни переменной может быть меньше, чем ее область видимости. Поле может быть действительным, только пока один из методов активен в стеке. Это может быть безобразно. Прилагайте усилия, чтобы убедиться, что время жизни переменной соответствует ее области видимости. Дополнительно убедитесь, что одноранговые переменные (те, что размещены в одной области видимости) имеют одно и то же время жизни.

Тип переменной адекватно связывается с именем декларации. Убедитесь, что тип декларации сам говорит за себя. Одно исключение к этому совету состоит в том, что имена переменных, хранящих несколько значений (имеются в виду коллекции), должны быть во множественном числе. Различие между одним значением и множеством значений важно для читателей.

Если область видимости, время жизни и тип адекватно представляют переменную в своих сферах, то имя переменной нужно использовать для указания на роль переменной в вычислениях. Уменьшая количество передаваемой информации до минимума, вы можете выбирать простые и хорошо читаемые имена.

Локальная переменная

Локальные переменные доступны только от места их определения до конца их области видимости. Следуя принципу, согласно которому информация должна распространяться настолько мало, насколько это возможно, определяйте локальные переменные прямо перед их использованием на самом внутреннем уровне вложения.

Есть несколько общих ролей для локальных переменных.

- Коллектор. Переменная, которая собирает информацию для последующего использования. Часто содержимое коллектора возвращается как результат функции. Когда вы возвращаете коллектор, назовите его `result` или `results`.
- Счетчик. Специальный коллектор, который накапливает число каких-то других объектов.

Пояснение: если у вас встречается сложное выражение, то присвоение частей выражения переменным может помочь читателю преодолеть сложное место.

```
int top= ...;
int left= ...
int height= ...;
int bottom= ...;
return new Rectangle(top, left, height, width);
```

Хотя для вычислений это и не требуется, поясняющая локальная переменная помогает записать то, что в противном случае было бы длинным и сложным выражением.

Поясняющие переменные являются шагом к вспомогательным методам. Выражение становится телом метода, имя локальной переменной подсказывает имя для метода. Иногда эти упрощения вводятся для упрощения вызываемого метода. Иногда они помогают избежать повторения общих выражений.

Повторное использование: когда значение выражения меняется, но вы бы хотели использовать то же значение более одного раза, сохраните его в локальной переменной. Например, если вам нужно применить ту же временн*ую* отметку к нескольким объектам, то вы не можете получать новое время для каждого из объектов.

```
for (Clock each: getClocks())
   each.setTime(System.currentTimeMillis());
```

Вместо этого локальная переменная будет хранить время для нужных вам целей.

```
long now= System.currentTimeMillis();
for (Clock each: getClocks())
   each.setTime(now);
```

- Элемент. Последнее общее применение для локальных переменных — это хранение элементов коллекции, по которой идет итерация. В последнем примере `each` — это понятное, простое имя для локальной переменной элемента. Если я захочу узнать, по каким элементам идет итерация, я могу взглянуть на предшествующий цикл `for`.

Для вложенных циклов добавьте имя коллекции к имени локальной переменной элемента, для того чтобы подчеркнуть принадлежность.

```
broadcast() {
   for (Source eachSender: getSenders())
     for (Destination eachReceiver: getReceivers())
       ...;
}
```

Поле

Область видимости и время жизни поля то же самое, что и у объекта, к которому он присоединен. Поскольку поля принадлежат объекту как целому, то определяйте поля вместе, либо в начале, либо в конце класса. В начале определения дают читателю важный контекст для использования в процессе чтения остального кода. Оставить описания на последний момент — это все равно что послать сообщение “Поведение это все; данные — всего лишь детали реализации”. Хотя я философски согласен с выражением, что в объектном программировании логика более важна, чем данные, мне по-прежнему нравится видеть любые определения в самом начале.

Одна из ваших возможностей — определить поле как `final`. Это сообщает читателям, что значение поля не будет изменяться после того, как конструктор закончит выполнение. Хотя я держу в уме, какие из моих полей "окончательные", а какие нет, я не декларирую это явно. Для меня дополнительная ясность не стоит дополнительной сложности, но если я пишу код, который будет меняться многими людьми на протяжении длительного времени, то, кажется, имеет смысл сделать явным размежевание между полями, которые будут сохранять значения, и теми, значения которых будут изменяться произвольно.

Список возможных назначений для полей не настолько исчерпывающий, как для локальных переменных. Однако ниже приведено несколько распространенных ролей.

- Вспомогательное. Вспомогательное поле хранит ссылку на объекты, используемые многими методами объекта. Если объект передается как параметр многим методам, то подумайте о том, чтобы заменить параметр вспомогательным полем, которое можно установить в конструкторе.
- Флаг. Поле логического типа, говорящее: "Этот объект может действовать двумя различными образами". Если есть метод записи для такого флага, то он дополнительно говорит: "... и вы можете изменить поведение объекта во время его существования". Поле типа "флаг" хорошо лишь в случае, если оно используется только в нескольких проверках. Если код, который принимает решения на основе флага, все время дублируется, то рассмотрите вариант с полем типа "стратегия".
- Стратегия. Если вы хотите выразить мысль, что существуют различные способы произвести какие-то вычисления в объекте, сохраните в поле объект, который осуществляет только изменяемую часть вычислений. Если эти изменения в поведении не изменяются в процессе жизненного цикла объекта, установите поле стратегии в конструкторе. В противном случае предоставьте методы для манипуляции им.
- Состояние. Поля состояния похожи на поля стратегии в том смысле, что им делегируется часть поведения. Однако поля состояния при переключении сами устанавливают следующее состояние. Поля стратегии изменяются, если вообще изменяются, другими объектами. Реализованные таким образом машины состояний сложно прочитать, поскольку состояния и переходы выражены в разных местах. Однако для простых машин состояний этого будет достаточно.
- Компоненты. Эти поля хранят объекты или данные, присвоенные объектами.

Параметры

Кроме нечастных переменных (полей или статических полей), параметры являются единственным способом связать состояние одного объекта с состоянием другого. Поскольку нечастные переменные вводят строгую привязку между классами

и поскольку эта привязка имеет тенденцию разрастаться со временем, то параметры имеют преимущество во всех случаях, когда возможно использование как статических полей, так и параметров.

Привязка, которую вводят параметры, слабее, чем привязка на основе постоянных связей от одного объекта к другому. Например, вычисления внутри древовидной структуры требуют обращения к родительскому узлу. Вместо того чтобы иметь постоянную ссылку на родительский узел (рис. 6.3), передача параметра этим методам требует более слабой привязки между узлами. Без перманентной ссылки на родителя возможно, например, иметь поддерево как часть нескольких деревьев.

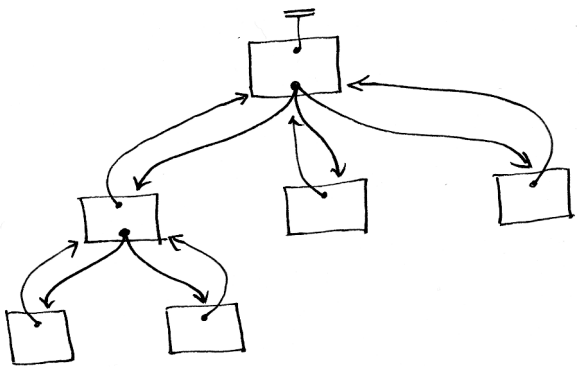


Рис. 6.3. Сильно связанная древовидная структура с указателями на родителей

Если во многих сообщениях от одного объекта к другому требуется тот же самый параметр, то, может быть, будет лучше навсегда присоединить параметр к вызываемому объекту. Параметры являются тонкими нитями, связывающими объекты вместе, но, подобно Гулливеру и лилипутам, много тонких нитей могут сделать объект неспособным к изменениям.

На рис. 6.4 показан простой параметр.

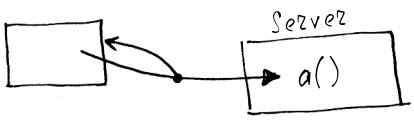


Рис. 6.4. Отдельный параметр вводит небольшую привязку

Эта схема иллюстрируется кодом:

```
Server s= new Server();
s.a(this);
```

Повторяя тот же параметр пять раз, мы значительно увеличим взаимную привязку.

```
Server s= new Server();
s.a(this);
s.b(this);
s.c(this);
s.d(this);
s.e(this);
```

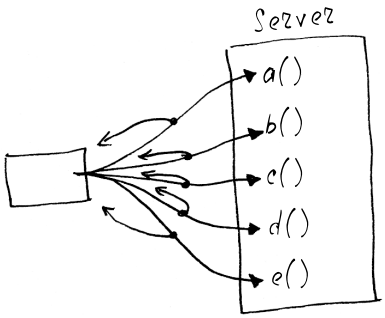


Рис. 6.5. Повторение параметров увеличивает привязку

В этом случае два объекта смогут лучше работать независимо, если параметры заменить ссылкой.

```
Server s= new Server(this);
s.a();
s.b();
s.c();
s.d();
s.e();
```

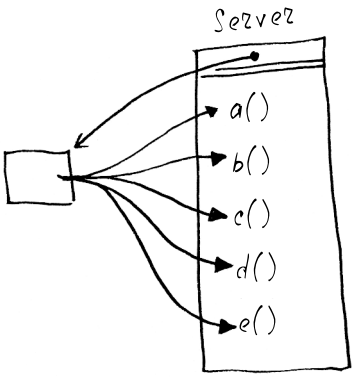


Рис. 6.6. Ссылка уменьшает связанность

Собирающий параметр

Вычисления, которые собирают результаты многих вызовов методов, требуют некоторого способа собирать эти результаты. Один из способов — вернуть результат изо всех методов. Это срабатывает, если значение простое, такое, как целое число.

```
Node
int size() {
  int result= 1;
  for (Node each: getChildren())
    result+= each.size();
  return result;
}
```

Когда объединение результатов выглядит более сложно, чем простое сложение, то более прямолинейно будет передать параметр, который будет собирать результаты. Например, собирающий параметр помогает при линеаризации дерева.

```
Node
asList() {
  List results= new ArrayList();
  addTo(results);
  return results;
}
addTo(List elements) {
  elements.add(getValue());
  for (Node each: getChildren())
    each.addTo(elements);
}
```

Некоторые другие примеры более сложного собирающего параметра: `GraphicsContext`, который передается по дереву элементов управления, или `TestResult`, который передается по дереву тестов в JUnit.

Опциональный параметр

Некоторые методы могут принимать или получать параметр по умолчанию, если он явно не указан. В таких случаях расположите обязательные параметры в начале списка параметров и добавьте опциональные в конце. Это делает общими столько параметров, сколько это возможно, и опциональные параметры остаются как альтернатива в конце.

Конструкторы `ServerSocket` демонстрируют опциональные параметры. Простейший конструктор не требует аргументов, но есть также и версия с опциональным номером порта, и другая версия, с опциональным номером порта и опциональной длиной очереди.

```
public ServerSocket()
public ServerSocket(int port)
public ServerSocket(int port, int backlog)
```

Языки с именованными параметрами могут выражать опциональные параметры более естественно. Поскольку Java располагает только позиционными параметрами, то факт, что параметр является опциональным, можно сообщить только по соглашению. Иногда это называют “телескопированием” шаблона параметров, чтобы придать физическую аналогию для создания коллекций параметров одной поверх другой.

Аргументы-переменные

Некоторые методы могут принимать любое число параметров заданного типа. Простое решение для этого случая — всегда передавать коллекцию как параметр. Однако вызывающая сторона таких методов часто засоряется из-за наличия временных коллекций.

```
Collection<String> keys= new ArrayList<String>();
keys.add(key1);
keys.add(key2);
object.index(keys);
```

Эта проблема является достаточно общей, так что Java предоставляет механизм для передачи методу переменного числа аргументов. Определяя показанный выше метод как `method(Class... classes)`, клиент может вызывать метод с любым числом аргументов.

```
object.index(key1, key2);
```

Аргументы-переменные должны быть последним параметром. Если метод имеет аргумент-переменную и опциональные аргументы, как описано выше, то опциональным аргументам приходится располагаться перед аргументами-переменными.

Объект-параметр

Если группа параметров передается вместе многим методам, то рассмотрите возможность ввести дополнительный объект-параметр. Его поля будут описывать то, что раньше было множеством параметров, и теперь достаточно будет передавать один этот объект. Когда вы поменяете длинные списки параметров на объект-параметр, то еще раз просмотрите код: возможно, некоторые фрагменты используют только поля нового объекта, так что лучше их выделить в методы нового объекта.

Например, в графических библиотеках Java широко распространено представление прямоугольника как независимых параметров: x, y, width и height. Иногда эти четыре параметра передаются через множество уровней вызовов методов, в результате чего код становится более длинным и сложным для чтения, чем мог бы быть.

```
setOuterBounds(x, y, width, height);
setInnerBounds(x + 2, y + 2, width - 4, height - 4);
```

Явно выделенный, как объект, прямоугольник, лучше объясняет код.

```
setOuterBounds(bounds);
setInnerBounds(bounds.expand(-2));
```

Введение объекта-параметра укорачивает код, объясняет его назначение и дает приют для алгоритмов растяжения и сжатия прямоугольника, которые в противном

случае должны были бы повторяться каждый раз, когда в них возникала необходимость (с частыми ошибками, когда программист забывает, что фактор растяжения должен быть продублирован для высоты и ширины). Многие мощные объекты начинали как объекты-параметры.

В то время как первичная мотивация для введения объектов-параметров была связана с улучшением читабельности, объекты-параметры могут стать важными пристанищами для логики. Факт, что данные появляются вместе в нескольких списках параметров, является кричащей подсказкой, что они сильно связаны. Класс, содержащий их фиксированный список, является явной формой определения "Этот набор данных сильно связан".

Аргумент, наиболее часто звучащий против объектов-параметров, — это производительность, поскольку размещение всех этих объектов занимает время. Во многих случаях это на практике не является проблемой. Если размещение объектов является узким местом, то объект-параметр снова может быть развернут, превращен в явный список параметров, если это необходимо. Лучше оптимизировать читабельный, взвешенный и протестированный код. Объекты-параметры могут внести в это свой вклад.

Константа

Иногда у вас есть значения, которые нужны в нескольких местах вашей программы, но которые не изменяются. Если эти величины известны на этапе компиляции, то сохраните эти значения в переменной, описанной как `static final`, и ссылайтесь на такую переменную в своей программе. Общим приемом является давать константам имена, состоящие целиком из символов верхнего регистра, чтобы подчеркнуть, что это не обычные переменные.

Важный момент в использовании переменных заключается в том, что, используя их, вы исключаете целый класс ошибок. Если у вас есть число 5, встроенное в ваш код, и затем вы меняете 5 на 6, то тут очень просто допустить промах. Если 5 имеет два различных значения, например "нарисовать границу объекта" и "передаваемый пакет является пакетом подтверждения", то изменение такой константы еще вероятнее вызовет ошибку. Самая сильная причина для использования констант в том, что вы можете взаимодействовать через имена констант, когда хотите подать сигнал, что вы имеете дело со значением. Читатель сможет понять `Color.WHITE` значительно лучше, чем `0xFFFFFF`. Если кодирование цвета изменится, то не будет необходимости менять саму константу, представляющую цвет.

Обычное использование констант — разъяснение различий в сообщениях интерфейса. Например, для центрирования текста вы можете вызвать `setJustification (Justification.CENTERED)`. Одно из преимуществ этого стиля API состоит в том, что вы можете добавить новые варианты существующих методов, добавляя новые константы без перестройки реализации. Однако эти сообщения не так хорошо чи-

таются, как отдельные методы для каждого варианта. В данном случае метод мог бы называться `justifyCentered()`. Интерфейс, опирающийся на вызовы методов, которые используют литеральные константы в качестве аргументов, может быть улучшен, если вы создадите по одному методу для каждого значения константы.

Имя, указывающее на роль

Как вы приходите к решению, как назвать переменную? Многие конфликтующие ограничения проистекают из этого вопроса. Я хочу полностью сообщать мои намерения через имя, что часто подразумевает длинное имя. Мне нравятся короткие имена, чтобы упростить форматирование кода. Имена должны быть прочитаны множество раз, по сравнению с количеством раз, когда они были введены, так что имена должны быть оптимизированы для чтения, а не для набора. Должны быть выражены и данные, хранящиеся в переменной, и роль, которую эти данные играют в вычислениях.

Есть несколько частей информации, которые мне нужны, когда я пытаюсь понять переменную. Каково ее назначение в вычислениях? Как используется объект, на который ссылается переменная? Каковы область видимости и время жизни переменной? Насколько широко ссылаются на эту переменную?

Многие схемы именования переменных включают тип переменной как часть имени. Мои имена так не делают. Какой смысл раз за разом сообщать компилятору тип переменной, если я собираюсь еще раз вернуться к этому и снова встроить ту же информацию в имя переменной? Я вижу полезным встраивание информации о типе в имена переменных в языках, которые не очень заботятся об ошибках типизации, таких как C. Java предоставляет достаточную поддержку для избежания таких ошибок.

Если я хочу узнать тип одной из моих переменных, моя оболочка IDE дает мне быструю информацию об этом. Использование коротких, насыщенных методов также дает мне быстрые ссылки на часто используемые переменные, локальные переменные и параметры.

Другая грань переменных, которую читатель должен понимать, — это область видимости переменной. Некоторые методы именования переменных включают область видимости как префикс имени, так что `fCount` будет полем, а `lCount` — локальной переменной. Опять же, конструируя достаточно короткие методы, я редко путаюсь в области видимости переменных. Если я не вижу определения переменной прямо в этом методе, то наиболее вероятно, что это — поле (я использую другие методы, чтобы исключить большинство статических полей).

Все это оставляет роль переменной как преимущественную информацию, которую я хочу сообщить через имя переменной, что в основном приводит к коротким, понятным именам. Если я испытываю трудности при подборе имени, то в основном из-за того, что не очень хорошо понимаю происходящие вычисления.

Есть несколько имен, которые часто встречаются в моем коде.

- `result`. Хранит объект, который будет возвращен из функции.
- `each`. Хранит индивидуальный элемент из коллекции во время итерации (хотя я становлюсь приверженцем использования единственного числа от имени коллекции, например, `(Node child: getChildren())`).
- `count`. Хранит счетчик.

Если у меня есть несколько переменных, которым я хочу дать одно и то же имя, то я его уточняю: `eachX` и `eachY` или `rowCount` и `columnCount`.

Я иногда склоняюсь к тому, чтобы сокращать слова в именах переменных. Это оптимизирует набор ценой более сложного прочтения. Поскольку переменные читаются гораздо чаще, чем пишутся, то это ложная экономия. Иногда я поддаюсь искушению использовать несколько слов для имени переменной, что делает переменную слишком длинной для комфортного набора. Когда это происходит, я смотрю на окружающий контекст. Почему я использую так много слов для того, чтобы выделить роль этой переменной среди ролей других переменных? Часто это приводит к тому, что я упрощаю общую схему, что дает мне возможность снова использовать короткие имена переменных в согласии с моими принципами.

Подводя итог: я сообщаю миру роль, которую играет переменная, через ее имя. Все остальные важные сведения о переменной, такие как ее время жизни, область видимости и тип, в основном могут быть легко переданы через контекст.

Определение типа

Одна из возможностей Java и других пессимистически типизированных языков — это необходимость определять типы переменных. Поскольку вам нужно определить тип, то вы также можете использовать определения типа как возможность для взаимодействия. Выберите определение типа, которое сообщает, как переменная будет использоваться, а не то, как она будет реализована.

`List<Person> members= new ArrayList<Person>()` говорит мне, что `members` будет использоваться как `List`. Я ожидаю увидеть вызов операций, таких как `get()` и `set()`, поскольку то, что отличает `List` от `Collection`, — это индексный доступ к данным.

Когда я впервые исследовал этот шаблон, я записал его догматически. Затем я попробовал использовать слабое правило, согласно которому все переменные должны быть определены настолько общими, насколько это возможно. Я обнаружил, что дополнительное удобство для обобщения всех типов не стоит того. Иногда переменная может быть типа `List`. Затем я могу передать ее в метод, где до того использовался только протокол `Collection`. Несовместимость между определениями была большей проблемой для читателей, чем отсутствие точности при определении переменой как `List` там, где она использовалась. Теперь я могу сказать это более мягко. Полезно определять переменные и методы с помощью общих типов, где это возмож-

но. Небольшая потеря точности и общности для удобства сопровождения является допустимой жертвой.

Главный момент при обобщении определения типов заключается в том, что вы оставляете возможность для дальнейшей модификации конкретного класса в процессе изменений. Если я определяю переменную как `ArrayList`, то не могу запросто изменить ее впоследствии на `HashSet`, как я бы мог, если бы определил ее как `Collection`. В основном, чем больше ваше решение предполагает, тем меньше у вас остается гибкости для дальнейших изменений. Для поддержки гибкости допускайте минимум информации, насколько это возможно, чтобы по возможности распространять общность. В предложении "`members` содержит `ArrayList`" больше информации, чем в выражении "`members` содержит `Collection`".

Фокусирование на взаимодействии является хорошей эвристикой для управления гибкостью. Определения типов являются примером этого. Когда я говорю, что переменная содержит `Collection`, я говорю об этом вполне точно. Хорошее взаимодействие обеспечивает лучшую гибкость.

Инициализация

Перед тем как вы сможете программировать, вам нужно знать, на что вы можете рассчитывать. Возможность сделать хорошие предположения помогает вам сфокусироваться на том, что именно нужно изучить. Одним из пунктов, который может оказаться полезным для ваших предположений, является состояние переменных. Инициализация — это процесс перевода переменных в заданное состояние перед тем, как их использовать.

Есть несколько вопросов при инициализации переменных. Один из них — это желание сделать инициализацию декларативной, насколько это возможно. Если инициализация и определение следуют вместе, то существует одно место, куда следует отправиться для ответов на вопросы о переменной. Другой вопрос — производительность. Переменные, которые дорого инициализировать, возможно, должны быть инициализированы через некоторое время после начала своего существования. Например, в Eclipse классы загружаются так поздно, как это возможно, чтобы поддерживать быструю скорость загрузки.

Ранняя инициализация

Одним из стилей инициализации является инициализация переменной настолько быстро, насколько она создается, — когда она определена или когда создается объект, в котором находится переменная (определение или конструктор). Одно из преимуществ ранней инициализации в том, что вы можете быть уверены, что переменная инициализирована до того, как она будет использована.

Инициализируйте переменные при определении, если это возможно. Это делает декларированный и актуальный типы близкими читателю.

```
class Library {
  List<Person> members= new ArrayList<Person>();
...
}
```

Инициализируйте поля в конструкторе, если они не могут быть инициализированы при определении.

```
class Point {
  int x, y;
  Point(int x, int y) {
    this.x= x;
    this.y= y;
  }
}
```

Есть определенная симметрия при инициализации всех полей объекта в одном месте, то ли в определении, то ли в конструкторе. Однако смешение двух стилей, кажется, также не вызывает никакой неразберихи, если объект остается в рамках допустимого размера.

Отложенная инициализация

Ранняя инициализация работает хорошо, пока вы не задумываетесь о цене вычисления значений переменных, которые появляются на свет. Когда вычисления дороги, вы должны отложить уплату этой цены (возможно, потому, что переменная может никогда не использоваться), создать метод чтения и инициализировать поле в момент вызова этого метода.

```
Library.Collection<Person> getMembers() {
  if (members == null)
    members= new ArrayList<Person>();
  return members;
}
```

Отложенная инициализация используется как самый общий прием. Ограничение в вычислительной мощности часто было причиной ее применения. Отложенная инициализация важна, когда вычислительная мощность является ограниченным ресурсом. Ограниченная в ресурсах среда, такая как Eclipse, где запуск должен быть быстрым, использует отложенную инициализацию, чтобы не загружать расширения до времени, когда они вскоре будут использованы.

Поле, инициализируемое с задержкой, сложнее понять, чем инициализируемое сразу. Читатель должен просмотреть как минимум два места, перед тем как поймет тип реализации поля. При кодировании вы сберегаете информацию для будущих читателей. К счастью, возникает только несколько часто встречающихся вопросов, так что несколько приемов достаточно для ответа на большинство из них. Отложенная инициализация говорит: “Здесь важна производительность”.

Выводы

Шаблоны состояния рассказывают о том, как взаимодействуют решения о представлении состояния программы. Следующая глава показывает обратную сторону медали: как взаимодействовать через решения передачи потока управления.

Глава 7

Поведение

Джон фон Нейман способствовал созданию одной из основных метафор программирования — последовательности инструкций, исполняющихся одна за другой. Эта метафора пронизывает большинство языков программирования, в том числе Java. Тема этой главы — описание поведения программы. Вам встретятся шаблоны, приведенные ниже.

- Управляющая логика. Описывает вычисления как последовательность шагов.
- Главный алгоритм. Выражает основную управляющую логику.
- Сообщение. Выражает управляющую логику посылкой сообщения.
- Выбор сообщения. Меняет имплементоров сообщения в ситуации выбора.
- Двойная доставка. Варьирует имплементоры сообщения вдоль двух осей, что выражает каскадный выбор.
- Декомпозиционное сообщение. Разделяет сложные вычисления на ряд взаимосвязанных частей.
- Обратное сообщение. Делает управляющую логику симметричной, посылая серию сообщений одному и тому же получателю.
- Приглашающее сообщение. Указывает на потенциальную вариацию, посылая сообщение, которое может быть имплементировано разными способами.
- Поясняющее сообщение. Уточняет назначение части логики.
- Алгоритм исключений. Выражает поведение в нестандартных ситуациях без вмешательств в основной алгоритм.
- Сторожевой пункт. Ранний возврат из логики исключения.
- Исключение. Описывает глобальный алгоритм исключений.
- Проверяемое исключение. Гарантирует, что заранее объявленные исключения будут перехвачены.
- Распространение исключений. Преобразует исключения в форму, необходимую для перехватчика.

Управляющая логика

Почему в программах вообще присутствует управляющая логика? Некоторые языки, такие как Prolog, вовсе не имеют этого понятия. В таких языках части логики плавают, словно в супе, ожидая, пока не активизируются определенные условия.

Java принадлежит к семейству языков, в которых последовательность исполнения является фундаментальным организующим принципом. Смежные строчки исполняются одна за другой. Условия заставляют код срабатывать только при определенных обстоятельствах. Циклы выполняют часть логики по многу раз. Сообщения активизируют несколько процедур. Исключения заставляют пройтись по стеку.

Все эти механизмы в сумме являются отличным средством выражения вычислений. Как автор/программист вы решаете, как реализовать свой алгоритм — как один поток вычислений, множество альтернативных потоков, каждый из которых одинаково важен, или какой-то комбинацией. Для начала вы группируете части логики в некую доступную для случайного читателя абстракцию, оставляя также различные детали для тех, кому нужно понять программу более глубоко. Некоторые группы кода представлены методами в классе, а некоторые обращаются к другому объекту.

Главный алгоритм

Программисты обычно помнят главный алгоритм своей программы. Обработка начинается здесь, кончается там. Вычисления идут своим путем, хотя могут появляться отклонения. Используйте нужный язык программирования, чтобы выразить алгоритм.

Некоторые программы, в частности те, которые спроектированы для работы во враждебном окружении, не имеют очевидного главного алгоритма. Однако таких меньшинство. Используя средства какого-либо языка программирования для описания редко исполняющихся, меняющихся самих по себе фактов, вы затеняете самую значимую часть программы, которую будут читать, понимать и часто менять. Это не значит, что исключительные условия несущественны, но ясное выражение главного алгоритма программы имеет большую ценность.

Для необычных условий или ошибок используйте исключения и сторожевые пункты.

Сообщение

Сообщение — одно из главных средств выражения логики в Java. Процедурные языки используют процедуры в качестве механизма сокрытия информации.

```
compute() {
  input();
  process();
  output();
}
```

Такая часть кода говорит: “Для понимания этих вычислений все, что вы должны знать, — это то, что они состоят из трех шагов, детали которых в данный момент несущественны”. Одна из прелестей объектного программирования — возможность выразить той же процедурой нечто большее. Для каждого метода потенциально существует целый набор одинаково структурированных вычислений, детали которых отличаются. И, как дополнительный бонус, отпадает необходимость четко описывать все эти детали будущих вариаций, если вы пишете инвариантную часть.

Использование сообщений как фундаментального механизма управляющей логики дает знание того, что изменение является основным состоянием программ. Каждое сообщение — потенциальная точка, в которой получатель сообщения может измениться, оставляя отправителя в прежнем состоянии. Вместо того чтобы сказать: “За этим скрывается кое-что, детали чего не важны”, основанный на сообщениях алгоритм скажет: “В этой точке случается что-то интересное касательно ввода. Детали могут варьироваться”. Мудрое использование этой гибкости, четкое выражение логики, где только возможно, и отнесение на будущее деталей — важное умение, если вы хотите писать эффективно взаимодействующие программы.

Выбор сообщения

Иногда сообщения отсылаются с целью выбора имплементации, как в операторе `case` процедурных языков. Например, если необходимо отобразить графику одним из нескольких способов, можно послать полиморфное сообщение, указывающее, что выбор будет иметь место во время исполнения.

```
public void displayShape(Shape subject, Brush brush) {
  brush.display(subject);
}
```

Сообщение `display()` выбирает реализацию, основываясь на типе кисти. Поэтому можно свободно реализовать любое количество кистей, например `ScreenBrush`, `PostscriptBrush` и т.д.

Свободное применение сообщений приводит к появлению кода с несколькими определенными условиями. Каждое выбирающее сообщение — потенциальная точка дальнейшего расширения. Каждое точное условие — другая точка, требующая определенной модификации для изменения поведения целой программы.

Для чтения кода, использующего много выбирающих сообщений, нужна некоторая сноровка. Читатель должен просмотреть несколько классов, прежде чем вникнет

в детали прохождения каждого пути вычислений. Как автор вы можете облегчить понимание, передавая свой замысел в именах методов. Остерегайтесь также применять сообщения слишком массированно. Если существует только несколько вариантов и больше не предвидится, не используйте сообщения ради предоставления возможности несуществующих вариаций.

Двойная доставка

Выбирающие сообщения хороши для выражения одного измерения вариабельности. В примере из "Выбор сообщения" это измерение было типом посредника, который рисовал форму. Если необходимо выразить два независимых измерения выбора, то можно делать это с помощью каскада выбирающих сообщений.

Допустим, мне нужно выразить, что овал Postscript обрабатывается по-другому, нежели прямоугольник. Для начала следует решить, куда я хочу поместить вычисления. Основная их часть как будто больше всего подходит объекту `Brush`, так что сначала нужно послать выбирающее сообщение в `Shape`, а потом в `Brush`.

```
displayShape(Shape subject, Brush brush) {
  shape.displayWith(brush);
}
```

Теперь каждый объект `Shape` имеет возможность по-разному имплементировать метод `displayWith()`. Вместо детализации процесса они добавляют свой тип к сообщению и передают управление в `Brush`.

```
Oval.displayWith(Brush brush) {
  brush.displayOval(this);
}
Rectangle.displayWith(Brush brush) {
  brush.displayRectangle(this);
}
```

Разные типы кистей теперь получают информацию, необходимую для того, чтобы они работали.

```
PostscriptBrush.displayRectangle(Rectangle subject) {
  writer print(subject.left() +" " +...+ " rect);
}
```

Двойная доставка вводит дубликацию, что негативно сказывается на гибкости. Имена типов получателей первого выбирающего сообщения разбрасываются по методам получателя второго сообщения. В данном примере это означает, что для добавления нового объекта `Shape` нужно будет вставить методы во все типы `Brush`. Если одно измерение более подвержено изменениям, чем другое, сделайте его получателем второго выбирающего сообщения.

Компьютерный исследователь во мне хочет обобщить ситуацию до тройной, четверной, пятерной доставки. Однако я лишь однажды пытался применить тройную доставку, и это было ненадолго — всегда находились другие пути для выражения многомерной логики.

Декомпозиционное сообщение

Иногда в сложных алгоритмах, состоящих из множества шагов, можно сгруппировать взаимосвязанные шаги и посылать сообщения, чтобы вызвать их. Внутренняя цель таких сообщений не в обеспечении передачи каких-то особенностей или чего-то столь же заумного. Это всего лишь старомодная функциональная декомпозиция. Сообщение используется просто для того, чтобы передать последовательность шагов процедуры.

Декомпозиционные сообщения нуждаются в описательном наименовании. Большинство читателей должны извлечь суть предназначения процедуры уже из имени сообщения. Только те, кому необходимо вникать в детали имплементации, могут прочитать код, вызываемый сообщением.

Трудности с наименованием декомпозиционных сообщений подсказывают нам, что это не лучший шаблон. Другой недостаток — длинные списки параметров. Если проявляются эти симптомы, я ввожу метод, вызываемый декомпозиционным сообщением, в основной текст и применяю другой шаблон, такой как объект-метод, чтобы успешно передать структуру программы.

Обратное сообщение

Симметрия улучшает читабельность кода. К примеру, возьмем метод

```
void compute() {
  input();
  helper.process(this);
  output();
}
```

Этот метод состоит из трех других, но ему недостает симметрии. Читабельность улучшится, если ввести вспомогательный метод, который выявит латентную симметрию. Теперь, читая `compute()`, не нужно тревожиться об отправителе — он входит в `this`.

```
void process(Helper helper) {
  helper.process(this);
}
void compute() {
  input();
```

```
  process(helper);
  output();
}
```

Таким образом, читатель может понять структуру метода `compute()`, просмотрев единственный класс.

Иногда вспомогательный метод, вызываемый обратным сообщением, становится важен сам по себе. Но в некоторых случаях их чрезмерное использование затеняет необходимость в перемещении функциональности. Посмотрим на код

```
void input(Helper helper) {
  helper.input(this);
}
void output(Helper helper) {
  helper.output(this);
}
```

То же самое можно лучше структурировать, перенеся весь метод `compute()` в класс `Helper`.

```
compute() {
  new Helper(this).compute();
}
Helper.compute() {
  input();
  process();
  output();
}
```

Порой я глупо себя чувствую, вводя методы только лишь для того, чтобы удовлетворить "эстетические" потребности наподобие симметрии. На самом деле эстетика заходит глубже. Она задействует большую часть мозга, чем это необходимо для строго линейного логического хода мыслей. Если вы воспитаете чувство эстетики кода, ваши впечатления станут приносить важную информацию о качестве кода. Эти ощущения, всплывающие из-под поверхности символического мышления, могут быть столь же ценными, как и в точности определенные и выверенные шаблоны.

Приглашающее сообщение

Иногда при написании кода возникает предположение, что кто-то захочет его расширить, введя субкласс. Чтобы обозначить возможность будущей переделки, используйте соответствующим образом названное сообщение — оно будет приглашать программистов к переработке кода для собственных целей.

Если для логики имеется имплементация по умолчанию, реализуйте ее сообщением. Если нет — объявляйте абстрактный метод, чтобы подчеркнуть приглашение.

Поясняющее сообщение

Различие в замысле и реализации всегда играло существенную роль в разработке ПО. Оно позволяет понять вычисления сначала в целом, а потом, если необходимо, в деталях. Можно использовать сообщения, чтобы обозначить это различие — посылая первым сообщение, содержащее в названии суть проблемы, которое впоследствии сменится сообщением, информирующем о способе решения.

Первый пример такого поведения я увидел в Smalltalk. В транслитерованном виде метод, обративший на себя внимание, выглядел так, как показано ниже.

```
highlight(Rectangle area) {
  reverse(area);
}
```

Я подумал: "Зачем это? Почему не вызвать `reverse()` напрямую, минуя метод `highlight()`?" После некоторого раздумья я осознал, что, хотя `highlight()` не преследует вычислительных целей, он служит для передачи замысла. Вызываемый код был написан в терминах решаемой проблемы, а именно выделения (`highlighting`) области экрана.

Предположим, что сообщение вводится для пояснения единственной строчки кода. Когда я встречаю

```
flags|= LOADED_BIT; // Установить бит загрузки
```

мне кажется, что лучше было бы написать

```
setLoadedFlag();
```

Хотя реализация `setLoadedFlag()` и является тривиальной, можно использовать этот метод для пояснения.

```
void setLoadedFlag() {
  flags|= LOADED_BIT;
}
```

Иногда вспомогательный метод, используемый объясняющими сообщениями, становится ценным сам по себе, как точка возможного расширения. Поймать удачу легко, если вы можете. Однако главная цель вызова поясняющих сообщений — это более четкое выражение замысла в коде.

Алгоритм исключений

Все программы имеют основной алгоритм, но они также могут иметь и один или более алгоритмов исключений. Эти пути вычислений менее важны для коммуникации, так как реже исполняются и меняются, а концептуально менее важны, чем главный

алгоритм. Выражайте его как можно яснее, так же поступайте и с исключениями, но не в ущерб главному алгоритму. Существуют два способа выразить алгоритм исключений — сторожевые пункты и собственно исключения.

Программы легче читать, если строчки исполняются одна за другой. Читатели могут использовать привычные навыки чтения прозы, чтобы понять замысел программы. Однако программы часто имеют множество путей. Одинаковое выражение всех путей приведет к созданию клубка змей, в котором флаги устанавливаются *здесь*, используются *там*, а возвращенные значения имеют особый смысл. Ответ на простой вопрос: "Какие строчки сейчас выполняются?" — становится комбинированным упражнением из области археологии и логики. Возьмите главный алгоритм, ясно опишите его, используйте исключения, чтобы выразить все остальное.

Сторожевой пункт

Все программы имеют главный алгоритм вычислений, но иногда возникают ситуации, требующие отклонений от алгоритма. Сторожевой пункт используется для описания простых нестандартных ситуаций с чисто локальными последствиями. Сравните

```
void initialize() {
  if (!isInitialized()) {
...
  }
}
```

и

```
void initialize() {
  if (isInitialized())
    return;
...
}
```

Во время чтения первой версии я отмечаю, что нужно будет взглянуть на пункт `else`. Я мысленно помещаю условие в стек. Это отвлекает внимание, пока я читаю пункт `then`. Первые две строчки второй версии сразу же ставят на заметку простой факт: получатель не был инициализирован.

Структура `if-then-else` выражает альтернативные, одинаково важные пути исполнения программы. Сторожевой пункт применим в несколько другой ситуации — когда одна ветвь вычислений превалирует над другой. В приведенном примере алгоритм, выполняющийся после инициализации объекта, более важен. Кроме того, легко заметить простой факт: даже при многократном вызове метода объект будет инициализирован только раз.

Возвращаясь в "средневековье" программирования, вспомним старую заповедь: процедура должны иметь одну точку входа и одну точку выхода. Так делалось, что-

бы предотвратить путаницу, возникающую при множественных переходах внутри процедуры. Этот принцип применим к языкам, подобным FORTRAN и Assembler, с большим количеством глобальных переменных, когда даже понимание структуры программы является тяжелой работой. В Java, с ее маленькими методами и преимущественно локальными данными, это чрезмерно консервативно. Данная часть программного фольклора, бездумно применяемая, мешает использованию сторожевых пунктов.

Они применяются, в частности, в множественных условиях.

```
void compute() {
  Server server= getServer();
  if (server != null) {
    Client client= server.getClient();
    if (client != null) {
      Request current= client.getRequest();
      if (current != null)
        processRequest(current);
    }
  }
}
```

Вложенные ветви условий приводят к дефектам. Та же самая версия кода, использующая сторожевые пункты, дает предпосылки для обработки запросов без сложной управляющей структуры.

```
void compute() {
  Server server= getServer();
  if (server == null)
    return;
  Client client= server.getClient();
  if (client == null)
    return;
  Request current= client.getRequest();
  if (current == null)
    return;
  processRequest(current);
}
```

Другим примером сторожевого пункта является оператор `continue` в циклах. Он говорит: “Пропустите этот элемент. Перемещайтесь к следующему”.

```
while (line = reader.readline()) {
  if (line.startsWith('#') || line.isEmpty())
     continue;
  // Здесь идет нормальная обработка
}
```

Кстати, этот код проводит четкую границу между нормальным исполнением цикла и логикой исключения.

Исключения

Исключения полезны для выражения тех переходов алгоритма, которые охватывают переходы между вызовами функций. Если вы понимаете, что проблема произошла несколькими уровнями выше по стеку — диск переполнен или потеряно соединение с сетью, — вы сможете корректно обработать этот факт только многими уровнями ниже в стеке вызова. Формирование исключения сразу же после ошибки и перехват его в точке, где оно может быть обработано, гораздо лучше, чем загромождение всего промежуточного кода точными проверками на все возможные ошибочные ситуации, из которых ни одна не будет обработана.

Исключения недешевы, они делают конструкцию частично убыточной. Тот факт, что вызываемый метод формирует исключение, влияет на имплементацию и дизайн всевозможных вызывающих методов, пока не будет достигнут перехватчик исключения. Это делает сложным отслеживание алгоритма управления, так как последовательные инструкции могут находиться в разных методах, объектах и пакетах. Код, который может быть написан с использованием условий и сообщений, но реализуется исключениями, чертовски сложно читать, потому как вы все время стараетесь понять, что же еще происходит в данной точке, кроме исполнения простой управляющей структуры. В общем, выражайте алгоритмы последовательностями, сообщениями, циклами и условиями (в таком порядке), где только возможно. Использование исключений без необходимости может запутать простой главный алгоритм.

Проверяемые исключения

В применении исключений есть некоторый риск — например, если исключение формируется, но никто его не перехватывает. Все, что происходит, — это аварийное завершение программы. Но нам нужно контролировать такие случаи, чтобы вывести диагностическую информацию и рассказать пользователю, что произошло.

Сформированные, но не перехваченные исключения — еще больший риск, чем если бы код, формирующий исключения, и перехватчик исключений были написаны разными людьми. Любые упущенные взаимодействия ведут к внезапному, невежливому по отношению к пользователю, выходу из программы.

Во избежание подобных ситуаций в Java введены проверяемые исключения. Они однозначно определяются программистом и проверяются компилятором. Код, в котором потенциально может появиться исключение, должен либо перехватить его, либо пропустить.

Проверяемые исключения очень дорого стоят. Во-первых, в стоимость входят объявления сами по себе. Это запросто прибавляет до 50% к длине метода и приводит к необходимости читать и понимать уровни между генератором исключения и перехватчиком. Во-вторых, проверяемые исключения затрудняют модификацию кода, хотя современные среды разработки делают рефакторинг менее обременительным.

Распространение исключений

Исключения возникают на разных уровнях абстракции. Перехват и обработка низкоуровневого исключения могут озадачить, если этого не ожидаешь. Когда вебсервер выводит мне на страницу ошибки, озаглавленную “NullPointerException”, содержимое стека — я вряд ли смогу как-то конструктивно использовать эту информацию. Вместо нее мне бы хотелось увидеть что-то похожее на “Программист не предусмотрел сценарий, представленный вами”. Я не возражаю, чтобы страница также предоставляла информацию об указателе, которую я потом послал бы программисту для диагностики проблемы, но детальный вывод ошибки без объяснения не очень полезен.

Низкоуровневые исключения часто содержат ценную информацию для определения дефекта. Инкапсулируйте низкоуровневые исключения в высокоуровневые и формируйте вывод, например в журнал программы, так, чтобы информации было достаточно для локализации ошибки.

Выводы

Управляющий алгоритм в объектных программах основан на исполнении методов. В следующей главе описывается выражение концепций вычислений в методах.

Глава 8

Методы

Логика делится на методы, не смешанные вместе в один большой ком. Почему? Какие проблемы могут быть решены введением очередного фрагмента? В чем вообще причина появления методов? Программу можно организовать как одну огромную процедуру с управляющими переходами на каждый случай — по крайней мере, концептуально это возможно. Хотя ранние программы так и были структурированы (со случайными новыми перестановками), "большой ком логики" страдал от проблем. Наиболее серьезной была сложность его прочтения. В одной гигантской процедуре трудно отличить важные части от менее важных, понять часть программы сейчас и оставить какие-то детали "на потом". Сложно выяснить, что существенно для вызывающего кода и что нужно знать тому, кто должен модифицировать эту функциональность. Вторая проблема заключается в том, что большинство задач программирования не уникальны. Вместо того чтобы реализовать каждый раз все с нуля, более удобно (и продуктивно) апеллировать к предыдущему решению. Гигантская процедура не предоставляет удобных способов ссылаться на отдельные части для повторного использования.

Разделение программы на методы дает возможность сказать: "Эти частицы логики не связаны тесно между собой". Разделение методов на классы и классы в пакетах продвигает взаимодействие дальше. Расположение этого кода находится в этом методе и того кода — в том, сигнализирует читателям, что данные части кода не связаны непосредственно. Их можно читать и понимать по отдельности. Более того, наименование методов дает шанс передать читателю назначение этой части вычислений независимо от ее реализации. Читатели зачастую могут узнать необходимую информацию, только лишь читая имена методов.

Методы также ловко решают проблему повторного использования. Когда вы пишете новую процедуру и вам необходима уже существующая в виде метода часть логики, ее можно вызвать.

Разделение больших вычислений на методы концептуально просто: располагайте связанные между собой части вместе, а не связанные — раздельно. На практике, впрочем, вы потратите время, энергию и определенный творческий потенциал, раздумывая сначала, что идет вместе и что нет, а затем определяя, как это лучше разделить. То, что является хорошим решением на данный момент, может не работать в будущем, когда вы поменяете логику системы. Разделение должно быть тем, что уменьшит в целом вашу загруженность работой. Знание того, какое разделение лучше работа-

ет, приходит с опытом. Здесь я приведу несколько подсказок из собственного опыта. Общими принципами разделения программы на методы являются размер, назначение и имя методов. Если вы сделаете слишком много маленьких методов, читателям будет тяжело следовать за таким фрагментированным выражением идеи. Слишком малое количество методов приводит к дублированию и сопутствующей потере гибкости. Существует множество типичных заданий в программировании, и создание нового метода — это обычный шаг к завершению многих из этих заданий. Методы, решающие одну из повторяющихся проблем, обычно легко называть. Именование методов, решающих уникальные проблемы, сложнее, но и важнее для читателей.

Ниже приведены шаблоны, относящиеся к методам.

- Составной метод. Компонует методы из вызовов других методов.
- Передающее замысел имя. Дает наименование методам по их предназначению.
- Область видимости метода. Делает метод как можно более частным.
- Объект-метод. Превращает сложные методы в объекты.
- Переопределенный метод. Переопределяет метод для выражения специализации.
- Перегруженный метод. Предоставляет альтернативный интерфейс к тем же вычислениям.
- Возвращаемый тип. Объявляет наиболее общий возвращаемый тип.
- Комментарий к методу. Комментирует методы для передачи информации, которую сложно извлечь из кода.
- Вспомогательный метод. Создает небольшие частные методы для более лаконичного выражения вычислений.
- Метод вывода отладки. Использует `toString()` для вывода полезной отладочной информации.
- Преобразование. Ясно выражает приведение одного типа объекта к другому.
- Метод преобразования. Использует метод на исходном объекте для простых приведений типа, возвращая преобразованный объект.
- Конструктор преобразования. Для большинства преобразований типа использует метод на классе преобразуемого объекта, принимая этот объект в качестве параметра.
- Создание. Ясно выражает создание объекта.
- Завершенный конструктор. Написание конструкторов, возвращающих полностью сформированные объекты.
- Метод-фабрика. Выражает более сложные операции создания с помощью статических методов класса вместо конструктора.

- Внутренняя фабрика. Если создание объекта требует пояснения или возможной переработки, заключает его во вспомогательный метод.
- Метод доступа к коллекциям. Предоставляет метод ограниченного доступа к коллекциям.
- Метод установки булевого значения. Предоставляет два метода для установки булевых значений (по одному на каждое состояние), если того требует взаимодействие.
- Метод запроса. Возвращает булевы значения из методов, именуемых как `asXXX`.
- Метод эквивалентности. Определяет `equals()` и `hashCode()`.
- Возвращающий метод. Изредка используется для одностороннего доступа к полям с помощью метода, возвращающего поле.
- Устанавливающий метод. Применяется еще реже; предоставляет возможность устанавливать поля.
- Безопасное копирование. Избегает ошибок наложения, копируя объекты, передаваемые в методы доступа или из них.

Составной метод

Данный метод компонует методы из вызовов других методов, которые находятся приблизительно на одном уровне абстракции.

Один из признаков плохо скомпонованного метода — это смесь различных уровней абстракции.

```
void compute() {
  input();
  flags|= 0x0080;
  output();
}
```

Подобный код раздражает читателя. Программу легче понять, когда она течет плавно, а внезапные сдвиги уровня абстракции нарушают течение. Чем заправляет тот бит в середине? Я спросил это у себя, когда читал вышеприведенный метод. Что он означает?

Одно возражение против использования множества маленьких методов — это снижение производительности за счет их вызовов. Когда я писал это, я сделал маленькую программку, сравнивающую миллион итераций цикла с миллионом сообщений. Преимущество было в среднем около 20–30% — недостаточно, чтобы повлиять на производительность большинства программ. В комбинации быстрые CPU и сильно локализованная природа узких мест программы делают производительность кода

хорошей до тех пор, пока вы не соберете статистику с использованием реальных наборов данных. Насколько длинным должен быть метод? Некоторые люди рекомендуют числовые ограничения — например, метод должен умещаться на страницу или 5–15 строк. Хотя читабельный код в основном удовлетворяет этим требованиям, но возникает вопрос: почему? Почему звенья логики работают наилучшим образом, когда они настолько велики?

Читатели кода вынуждены решать несколько проблем, которые по-разному влияют на размер методов. Когда охватываешь общую структуру, лучше видеть побольше кода. Свободное место в методе дает ключи к структуре и сложности кода. Есть ли там условия и циклы? Насколько глубоко вложены управляющие структуры? Как много работы требуется, чтобы выполнить задание, скрытое за именем метода? Один и тот же большой метод, который помог мне сориентироваться, становится препятствием, когда я пытаюсь понять код в деталях. Я могу эффективно держать в голове только одну значимую деталь в один момент времени, а метод в тысячу строк содержит более одной такой детали. Чтобы понять детали, я хочу собрать тесно связанные и разделить независимые.

Одновременная поддержка навигации и систематизации — это путь автора, который разделяет логику на методы. Я нахожу, что мой код более читабелен, когда он разбит на относительно короткие методы (по крайней мере, по стандартам C). Уловка подходит, когда у меня есть относительно независимые наборы деталей, которые могут быть разнесены в соответствующие методы. Иногда деталей слишком много для понимания, но их нелегко разделить. В таком случае я создаю объект-метод, чтобы дать всем деталям место для организации.

Другая проблема в выборе размера метода — специализация. Методы правильного размера могут быть переопределены целиком, без необходимости копировать код в субкласс и редактировать и без переопределения двух методов для одного концептуального изменения. Составные методы основаны на фактах, а не на спекуляции. Сделайте код работающим, а потом решайте, как он будет структурирован. Если вы потратили много времени на систематизацию заранее, вам всего лишь нужно будет отменить всю работу и переделать ее, когда вы узнаете что-то о реализации. Когда все детали логики разложены передо мной, гораздо легче разумно скомпоновать методы. Иногда мне кажется, что я знаю, как должны быть скомпонованы методы, но, когда я разделяю логику, оказывается, что результат сложно читать. В таких случаях я нахожу полезным написать все методы по строчкам, пока снова не получится один гигантский метод, который я снова разбиваю на части, учитывая недавний опыт.

Имя, передающее замысел

Методы должны именоваться для целей, которые подразумеваются при возможном вызове метода. Есть и другая информация, которую вы можете захотеть вложить в имя, — например, стратегия реализации. В любом случае передавайте замысел

в имени, а остальную информацию о методе — другими способами. Стратегия реализации — это дополнительная информация, наиболее часто включаемая в имена методов. Например:

```
Customer.linearCustomerSearch(String id)
```

Самой лучшей заменой было бы

```
Customer.find(String id)
```

так как это больше рассказывает о методе. Однако ваша задача как автора кода заключается не в том, чтобы просто выложить о программе все, что возможно, и так быстро, как возможно. Иногда нужны ограничения. Если стратегия реализации не существенна для пользователей, оставьте ее вне имени. Любопытствующий может посмотреть на тело метода, чтобы увидеть, как это реализовано. Даже если `Customer` предлагает одновременно линейный и хешированный поиск, лучше будет обозначить различия в именах, имея в виду точку зрения пользователя.

```
Customer.find(String id)
Customer.fastFind(String id)
```

(В действительности, вероятно, в этом случае будет лучше предложить только один `find()`, удовлетворяющий всех пользователей, но это другая история.) Реализация быстрой версии `find()` с помощью ассоциативного массива или дерева на самом деле не важна для пользователей метода.

Думайте об именах методов с точки зрения внешнего вида вызывающего кода. Это место, где читатели ожидают впервые встретить имя. Почему был вызван этот метод, а не другой? На этот вопрос может удовлетворительно ответить имя метода. Вызывающий метод должен рассказывать историю. Именуйте методы так, чтобы названия помогали рассказывать.

Если вы реализуете методы по аналогии с существующим интерфейсом, давайте им имена, используемые в интерфейсе. Если у вас есть особый вид итератора, назовите методы `hasNext()` и `next()`, даже если вы формально не реализуете интерфейс `Iterator`. Если методы являются только разновидностью одной функциональности, сначала подумайте, правильная ли метафора используется, а потом выражайте различия в виде префиксов к имени метода.

Область видимости метода

Каждый из уровней видимости — публичный, пакетный, защищенный и частный — что-то говорит о внутреннем смысле метода.

Два больших конфликтующих ограничения на область видимости метода — это необходимость донести какую-то функциональность внешнему пользователю и в то же время обеспечить будущую гибкость. Чем больше методов открывается для широ-

кого использования, тем сложнее поменять интерфейс к нужному объекту. При разработке JUnit Эрик Гамма (Erich Gamma) и я часто имели разногласия по поводу видимости методов. Программист Smalltalk во мне говорил, что создание видимых методов потенциально ценно для клиентов. Опыт Эрика в Eclipse научил его ценить гибкость, которая получается, когда раскрываешь как можно меньше. Я медленно продвигаюсь к его точке зрения. При выборе видимости у вас есть две цены. Одна из них — цена будущей гибкости. Очень узкий интерфейс облегчает будущие изменения. Вторая цена — это затраты при вызове объекта. Слишком тесный интерфейс заставляет всех клиентов делать больше работы, чем необходимо при использовании вашего объекта. Соблюдение баланса в этом вопросе — отправная точка к хорошим решениям относительно видимости.

Моя общая стратегия заключается в ограничении видимости, насколько это возможно. Если бы это было все, что требуется, то видимость методов мог бы регулировать и автоматический инструмент. По-настоящему сложная проблема начинается, когда какой-то код вызывает методы вне вашего прямого контроля. В таком случае вам приходится оперировать догадками, чтобы решить, какие методы будут публичными или защищенными, поручая вам поддерживать их или выделять солидные средства на изменение.

- Публичный. Когда метод объявляется публичным, вы выражаете свою уверенность, что он будет полезен и вне пакета, в котором он объявлен. Публичный метод означает, что вы берете под свою ответственность его поддержку, либо оставляя его неизменным, либо исправляя весь вызывающий код, либо, по крайней мере, уведомляя программистов об изменениях.

  ```
  public Object next();
  ```

  Это объявление говорит о том, что отныне и в обозримом будущем `next()` будет доступен клиентам.

- Пакетный. Пакетная видимость означает, что данный метод полезен для объектов внутри пакета, но доступ к нему извне нежелателен. Это вид добавочного определения: посторонние объекты тоже нуждаются в этом методе, но не все, только мои. Рассматривайте пакетную видимость как предположение о том, что функциональность будет перемещаться, так что метод станет менее видимым, либо о том, что метод может оказаться более полезным, чем предполагалось, но будет слишком дорого сделать его публичным.

- Защищенный. Этот род видимости хорош, только если код будет повторно использован в субклассах. Хотя он кажется более ограниченным, чем пакетный, защищенная и пакетная видимости в действительности ортогональны, так как субклассы вне пакета могут видеть и вызывать защищенные методы.

- Частный. Частные методы совершенны с точки зрения будущей гибкости, потому что всегда можно гарантировать, что вы сможете найти и изменить все вызывающие методы независимо от того, используют ли, расширяют ваш код

или нет. Делая метод частным, вы говорите, что ценность этого метода для внешних пользователей не настолько велика, чтобы сделать его более широко доступным.

Раскрывайте методы постепенно, начав с наиболее ограниченной видимости, которая будет работать. Если метод больше не нуждается в видимости, уменьшите ее. Это возможно, только когда у вас есть доступ ко всему вызывающему коду, и можно быть уверенным, что вы не помешаете клиентам, убрав из поля зрения метод, на который они рассчитывали. Я часто замечал, что методы, которые изначально кажутся частными, становятся ценными членами интерфейса, когда объект начинает использоваться другим образом.

Объявление финальных методов схоже с выбором видимости. Это значит, что вы не подразумевали использование этого метода людьми и не разрешаете его менять. Если инвариантность, поддерживаемая методом, достаточно сложна и тонка, этот уровень самозащиты может быть оправдан. Вы платите за уверенность в том, что никто не разрушит объект, устраняя вероятность его потенциально прибыльного переопределения и заставляя вместо этого кого-то делать ту же работу другим способом. Я не использую `final` и бываю разочарован, встретив финальные методы, когда у меня есть законное основание для их переопределения.

Объявление статического (`static`) метода делает его видимым, даже если вызывающая сторона не имеет доступа к экземпляру класса (т.е. видимость изменяется, хотя используются другие ключевые слова). Статические методы ограничены тем, что они не могут ссылаться на какой-либо экземпляр состояния, и поэтому не являются хорошими репозитариями для сложной логики. Статические методы могут быть унаследованы, но, однажды переопределенный, метод суперкласса не может быть вызван. Один из путей использования статических методов — замещение конструкторов.

Объект-метод

Это один из моих любимых шаблонов, возможно потому, что я использую его очень нечасто, но результаты впечатляют. Создание объекта-метода может помочь вам превратить запутанный клубок кода, упакованный в невозможный метод, в читабельный, ясный код, постепенно раскрывающий детали. Я применяю данный шаблон, когда у меня есть какой-то работающий код, и чем сложнее метод, тем лучше. Чтобы создать объект-метод, посмотрите на длинный метод с множеством параметров и временными переменными. Попытки извлечь любую часть метода приводят к созданию трудноименуемых субметодов с длинными списками параметров. Ниже описаны шаги к созданию объекта-метода (такой рефакторинг еще не поддерживается автоматически, если я пишу это).

1. Создайте класс по имени метода. Например, `complexCalculation()` станет `ComplexCalculator`.

Для каждого параметра, локальной переменной и поля, используемых в методе, создайте по полю в новом классе. Присвойте этим полям те же имена, что и в оригинальном методе (позже их можно будет исправить).

2. Создайте конструктор, который в качестве параметров будет принимать те же аргументы, что и в исходном методе, и использовать такие же поля.
3. Скопируйте оригинальный метод в новый метод `calculate()` класса. Параметры, локальные переменные и поля старого метода станут ссылками на поля в новом объекте.
4. Замените тело исходного метода кодом, который создает экземпляр нового класса и вызывает `calculate()`. Например:

```
complexCalculation() {
  new ComplexCalculator().calculate();
}
```

5. Если в оригинальном методе устанавливаются какие-то поля, сделайте это после возврата из `calculate()`.

```
complexCalculation() {
  ComplexCalculator calculator= new ComplexCalculator();
  calculator.calculate();
  mean= calculator.mean;
  variance= calculator.variance;
}
```

Убедитесь, что переделанный код работает так же, как и старый. Теперь-то и начинается самое интересное. Код в новом классе легко переработать. Вы можете извлечь методы и никогда не передавать никаких параметров, потому что все данные, используемые методом, хранятся в полях. Зачастую, начав извлекать методы, вы поймете, что некоторые переменные могут быть перенесены из полей в локальные переменные. Также может присутствовать информация, которую возможно передавать в метод как параметр вместо хранения ее в поле. Как только вы начнете извлекать методы, может обнаружиться, что какие-то общие конструкции, которые было тяжело изолировать, становятся полезными вспомогательными методами с осмысленными именами.

Иногда оригинальный метод бывает уже выделен на тот момент, когда я собираюсь создать объект-метод. В этом случае запишите по порядку все субметоды, чтобы убедиться, что весь код находится в одном месте. Явный признак того, что вам нужно проделать эту процедуру перед созданием объекта-метода, — это необходимость вызывать методы в исходном объекте. Сделайте резервную копию, распишите их и начните снова.

Переопределенный метод

Одна из прелестей объектного программирования — в разнообразии путей, которые предоставляются для выражения отличий между схожими вычислениями. Переопределенные методы дают простой способ описать вариацию. Методы, объявленные абстрактными в суперклассе, — это ясное указание конкретизировать вычисление, но любой не финальный метод является кандидатом для выражения вариации существующих вычислений. Хорошо скомпонованные методы суперкласса дают множество зацепок, на которые вы можете повесить свой собственный код. Если код суперкласса помещен в небольшие, связанные звенья, то можно переопределять методы целиком.

Переопределение метода — это не или-или. Вы можете выполнить код субкласса и суперкласса, вызвав `super.method()`. Делайте это, только вызывая методы с одинаковыми именами. Если ваш субкласс однозначно собирается вызывать иногда свой код, а иногда — код суперкласса, его будет сложно изучать и легко случайно поломать. Если все же чувствуется необходимость вызывать методы суперкласса, вы можете улучшить код, перерабатывая управляющие структуры до тех пор, пока не отпадет нужда в прыжках туда-сюда между субклассом и суперклассом.

Слишком большие методы суперкласса создают дилемму: скопировать код вниз по иерархии в субкласс или отыскать другой путь выражения вариации? Проблема состоит в том, что кто-либо может прийти после и поменять скопированный код в суперклассе, нарушив программу без вашего ведома.

Перегруженный метод

Когда вы объявляете тот же метод с отличающимися типами параметров, вы говорите: “Это альтернативный формат параметров данного метода”. Примером является метод, принимающий имя файла в виде объекта `String`, или непосредственно открытый поток в виде `OutputStream`. Это предоставляет пользователям возможность использовать как простой интерфейс имени, так и сохранить гибкость на случай, если кто-либо захочет передать в параметре сформированный поток (например, в целях тестирования). Перегруженные методы освобождают вызывающую сторону от ответственности за преобразование параметров, если существует несколько законных путей их передачи.

Вариантом перегрузки является использование одного и того же имени метода, но с другим числом параметров. Проблему этого стиля составляет то, что читатели, которые захотят спросить: “Что случится, когда я вызову этот метод?”, должны будут прочитать не только имя метода, но и список параметров, прежде чем узнают достаточно, чтобы понять результаты вызова. Если перегрузка сложна, читателям необходимо будет уловить тонкие различия между заданными типами аргументов.

Все перегруженные методы должны служить одной цели, вариации допустимы только в типе параметров. Разные возвращаемые типы для различных перегруженных методов слишком затрудняют чтение кода. Лучше найти новое имя для нового замысла. Давайте разные имена разным вычислениям.

Возвращаемый тип

Тип возврата метода свидетельствует прежде всего о том, является ли метод процедурой, при работе которой возникают сторонние эффекты, или функцией, возвращающей определенный тип объекта. Магический возвращаемый тип `void` позволяет Java избегать ключевого слова для обозначения различия между процедурами и функциями.

Намереваясь писать функцию, выберите возвращаемый тип для выражения вашего замысла. Этот тип может быть специфическим, конкретным классом или одним из простейших типов. Однако методы должны быть настолько широко применимы, как это возможно, поэтому выбирайте наиболее абстрактный возвращаемый тип. Это сохраняет возможность поменять конкретный возвращаемый тип, если возникнет необходимость.

Обобщение возвращаемого типа также может быть способом скрыть детали реализации. Например, возвращая `Collection` вместо `List`, можно сообщить пользователям, что элементы списка не обязательно построены по порядку.

Возвращаемые типы — это полигон для изменений во время разворачивания программы. Можно начать с возвращения конкретного класса и позже обнаружить, что несколько относящихся друг к другу методов возвращают различные конкретные классы, каждый из которых реализует или разделяет общий интерфейс. Выражение этой схожести объявлением общего интерфейса (если необходимо) и возвращением нового интерфейса из всех методов поможет читателям понять сходство.

Комментарий к методу

Выражайте как можно больше информации через имена и структуру кода. Добавляйте комментарии, если информация не очевидна из кода. При необходимости добавьте javadoc-комментарии, чтобы объяснить назначение методов и классов.

Многие комментарии абсолютно излишни в коде, написанном для коммуникации. Стоимость их написания и поддержка их соответствия с кодом не увеличивают их ценности.

Комментарии к методам — неудобный уровень абстракции. Если два метода ограничивают друг друга (например, первый должен вызываться перед вторым), то где должен помещаться комментарий? Необходимо обновлять комментарии вместе с кодом, и не существует никакой обратной связи, если комментарий устаревает.

Автоматизированные тесты могут передать информацию, которая не подходит на роль комментария к методу. В примере, приведенном выше, я могу написать проверку, удостоверяющуюся в генерации соответствующего исключения, если методы вызываются в неправильном порядке (хотя я предпочел бы устранить или инкапсулировать это ограничение). Автоматические проверки имеют много преимуществ. Их написание является ценным упражнением по проектированию, особенно когда осуществляется до реализации. Если тесты запускаются, они совмещены с кодом. Автоматизированные инструменты рефакторинга могут помочь обновлять тесты.

Все это говорит о том, что коммуникация является превалирующей ценностью в этих шаблонах реализации. Если комментарий к методу является наилучшим посредником для коммуникации, пишите хороший комментарий.

Вспомогательный метод

Вспомогательные методы — это следствие составных методов. Если вы собираетесь разделить большие методы на несколько маленьких, вам нужны эти меньшие методы-"помощники". Их назначение — сделать крупномасштабные вычисления более читабельными, скрыть временные незначительные детали и обеспечить возможность выразить ваш замысел в имени метода. "Помощники" обычно объявляются частными, становясь защищенными, если класс дополняется субклассом. Можно создать частный вспомогательный метод только для того, чтобы обнаружить, что внешние пользователи тоже хотят вызывать его. Если метод полезен внутренне, то есть шанс, что он станет также полезен и внешне. Даже если ваши маленькие "помощники" никогда не станут "выпускниками", они все же останутся ценной точкой взаимодействия.

"Помощники" обычно короткие, но иногда могут быть слишком короткими. Вот и сегодня я убрал вспомогательный метод, который только возвращал конструктор класса. Я нашел, что

```
return testClass.getConstructor().newInstance();
```

коммуницирует лучше как

```
return getTestConstructor().newInstance();
```

Однако этот вспомогательный метод может быть оправдан, если субклассы переопределяют вычисления конструктора.

Устраняйте "помощников" (по крайней мере, временных), когда логика метода становится неясной. Распишите все вспомогательные методы, посмотрите на логику свежим взглядом и заново извлеките методы, которые имеют смысл.

Последнее назначение вспомогательных методов — устранение общих субвыражений. Если вы вызываете такой метод повсеместно в классе, где требуется определенное маленькое вычисление, то его легко будет поменять. Если те же одна–три

строчки дублируются по всему объекту, то вы не только теряете возможность взаимодействия через хорошо подобранное имя, но также получаете трудности с модификацией кода.

Метод вывода отладки

Есть много потенциальных причин представления объекта в виде строки. Вы можете захотеть вывести объект пользователю, сохранить для последующего восстановления или показать содержимое объекта программисту.

Интерфейс `Object` довольно узок — он содержит одиннадцать методов. Один из них, `toString()`, выводит объект-получатель в виде строки, но с какой целью? У такого соблазнительного действия есть несколько назначений. Но они вырабатываются сами по себе в каждом конкретном случае. То, что хотят знать об объекте трейдер, программист или база данных, различается.

Вложения в высококачественный механизм отладки подобны действию рычага. Чтобы обнаружить важную внутреннюю деталь объекта, может понадобиться полминуты щелчков мышью. Выведите ту же деталь в `toString()`, и та же информация будет доступна за секунду, в результате одного щелчка. Я предпочитаю постигать свою программу при отладке, вместо того, чтобы просматривать объекты в среде разработки. Для интенсивных отладочных сессий сфокусированность на поддержке может сберечь минуты или часы попыток.

Поскольку `toString()` — публичный метод, это объект нападения. Если объект не поддерживает необходимый протокол, то нужно будет проанализировать строку вывода, чтобы получить полезную информацию. Такой код хрупок, потому что `toString()` часто меняется. Лучшая политика для предотвращения такой опасности — приложить все усилия, чтобы быть уверенным, что все ваши объекты имеют протоколы, нужные клиентам.

Таким образом, переопределяйте `toString()`, когда вам нужно обеспечить дружественный к программисту вывод объекта. Выводите строки иного формата в виде других методов или отдельных классов.

Преобразование

Иногда у вас есть объект A и вам нужен объект B, чтобы перейти к дальнейшим вычислениям. Как вы представите это преобразование исходного объекта в конечный?

Смысл всех этих шаблонов, в том числе шаблонов преобразования, — ясно передать замысел программиста. Однако есть несколько технических факторов, влияющих на наиболее эффективный путь выражения приведений одного типа к другому. Один из них — это число необходимых приведений. Если единственный объект должен быть всего лишь преобразован в другой объект, то можно пойти простым

путем. Потенциально неограниченное число преобразований требует иного подхода. Следующая проблема заключается в зависимостях между классами. Не так уж плохо ввести новую зависимость только для того, чтобы иметь удобное выражение приведения.

Реализация приведения — тоже отдельная проблема. Иногда вам нужно создать полноценный объект нового типа, скопировав информацию из исходного объекта. Иногда вы можете реализовать интерфейс конечного объекта, не копируя информацию исходного. Как альтернатива преобразованию, бывает возможным также найти общий интерфейс для обоих объектов и написать соответствующий код.

Метод преобразования

Если вам нужно выразить преобразование между объектами схожего типа и существует ограниченный набор преобразований, представьте их в виде метода исходного объекта.

Допустим, вы хотите реализовать декартову и полярную координатные системы. Чтобы создать метод преобразования типов, реализуйте

```
class Polar {
  Cartesian asCartesian() {
...
  }
}
```

Заметьте, что тип, возвращаемый методом преобразования, содержит специфику класса конечного объекта. Точка преобразования — получение объекта по другому протоколу. Альтернативой может быть реализация `getX()` и `getY()` объекта `Polar` и объявление `Polar` и `Cartesian` как реализаторов `Point`, что позволит игнорировать преобразование.

Методы преобразования имеют преимущества, так как хорошо читаются. Они достаточно популярны (более сотни примеров в Eclipse, например). Однако для их создания вам нужно иметь доступ к модифицируемому протоколу исходного объекта. Эти методы также вводят зависимость между исходным и конечным объектом. Если такая зависимость еще не существует, не так уж плохо ввести одну ради удобства, предоставляемого методом преобразования. Наконец, такие методы становятся громоздкими, когда их число потенциально не ограничено. Класс с двадцатью различными `asThis()` и `asThat()` трудно читать. Вместо этого вы можете поменять клиенты так, что они смогут обработать исходный объект без преобразования.

Эти недостатки заставляют меня скупо использовать методы преобразования и только в тех ситуациях, когда преобразуются объекты сходного типа. В других случаях я использую конструктор преобразования, чтобы выразить его.

Конструктор преобразования

Конструктор преобразования берет исходный объект в качестве параметра и возвращает конечный объект. Этот шаблон полезен, когда исходный объект преобразуется в множество конечных, потому что преобразования не нагромождаются в исходном объекте.

Например, `File` поддерживает конструктор, который преобразует объект `String`, представляющий имя файла, в объект, пригодный для чтения, записи и удаления. Хотя может быть удобным иметь `String.asFile()`, число вариаций на эту тему огромно, поэтому лучше написать `File.(String name)`, `URL(String spec)` и `StringReadStream(String contents)`. Иначе `String` может иметь неограниченное количество методов преобразования.

Если вам нужна свобода в реализации преобразований и необходимо возвращать что-либо иное, нежели конкретный класс, конструктор может быть выражен как фабричный метод, возвращающий более общий тип (или помещен в другой класс, отличный от созданного методом).

Создание

Встарину (полвека назад) программы были большими едиными массами кода и данных. Управление могло передаваться из ниоткуда в никуда. Доступ к данным мог быть получен откуда угодно. Обработка информации, изначальное назначение компьютеров, совершалась (говоря относительно) молниеносно и совершенно точно. Потом же открылся неловкий факт: написанная программа столь же часто меняется, сколь и запускается. Все эти прыжки управления, и самомодифицирующийся код, и данные, доступные отовсюду, прекрасны для выполнения, но ужасны, если вы собираетесь позже изменить программу. И так началась длинная и прерывистая дорога поиска моделей вычисления, в которых изменение *здесь* не создает непредвиденной проблемы *там*.

Меньшие программы обычно легче модифицировать, чем большие. Одна ранняя стратегия, облегчающая изменения программ, состояла в разделении большого компьютера, исполняющего большую программу, на связку меньших компьютеров (объектов), исполняющих маленькие программы. Объекты служили будущим изменениям, обеспечивая событийную подложку, изменение которой имело небольшую стоимость для программы.

Это разделение существует для нужд человека: чтобы приспособить наши подверженные ошибкам, переменчивые, изобретательные, неподходящие компьютерам умы. Компьютер работает одинаково, является ли код одним большим бесформенным комом, или тщательно спроектированной сетью из сопровождающих друг друга объектов. Для читателя создание объекта означает: какие-то утверждения объединены, чтобы обеспечить некоторые вычисления, детали которых несущественны сейчас.

Использование создания объектов выразительно требует соблюдения баланса между необходимостью ясного и прямого описания и гибкостью. Следующие шаблоны реализации предоставляют техники для выражения вариаций на тему “сделай меня объектом”.

Завершенный конструктор

Объектам нужна определенная информация перед тем, как они смогут вычислять. Передавайте эти предварительные условия потенциальным пользователям, создавая конструкторы, которые возвращают объекты, готовые к вычислениям. Если есть множество путей настройки объекта, предоставьте множество конструкторов, каждый из которых будет возвращать правильно сформированный объект.

```
new Rectangle(0, 0, 50, 200);
```

Иногда создание объекта в конструкторе без аргументов и последующий вызов серии настраивающих методов делают код более гибким. Однако этот подход не рассказывает, какие комбинации параметров требуются для корректной работы объекта.

```
Rectangle box= new Rectangle();
box.setLeft(0);
box.setWidth(50);
box.setHeight(200);
box.setTop(0);
```

Могу ли я продолжать без какого-либо из этих параметров? Нельзя сказать ничего определенного, просто прочитав интерфейс. Если же я вижу конструктор с четырьмя аргументами, то я понимаю, что каждый из них необходим.

Конструкторы подключают клиентов к конкретному классу. В каких-то случаях вы можете быть удовлетворены кодом, вызывающим конструктор конкретного класса. Если же требуется более абстрактный код, введите фабричный метод. Даже если он уже существует, предоставьте еще и завершенный конструктор, чтобы любопытные читатели могли быстро понять, какие параметры нужны для создания объекта.

Когда реализуете завершенный конструктор, вынесите все конструкторы в один главный конструктор, который производит всю инициализацию. Это гарантирует, что все прочие конструкторы будут создавать объекты, удовлетворяющие всем необходимым инвариантам, и передадут их будущим модификаторам класса.

Метод-фабрика

Альтернативный способ представить создание объекта — статический метод класса. Такие методы имеют пару преимуществ перед конструкторами: они могут возвращать более абстрактный тип (субкласс или реализацию интерфейса) и могут что-то

нести в своем имени, не только имя класса. Однако они усложняют код, поэтому их следует использовать, только когда необходимо, не просто ради самих себя.

Приведенный к фабричному методу, пример с `Rectangle` может выглядеть приблизительно так, как показано ниже.

```
Rectangle.create(0, 0, 50, 200);
```

Если вы замышляете что-то более сложное, чем создание объекта, — например, запись объектов в кэш или создание субкласса, который будет определен во время выполнения, то метод-фабрика будет полезен. Однако, как читатель, я всегда бываю озадачен, когда вижу такой метод. Что еще там происходит, кроме создания объекта? Я не хочу тратить время своих читателей, поэтому, если все что случается в методе, — это красивое создание объекта, я выражаю его через конструктор. Если же в это время происходит что-то еще, я ввожу метод-фабрику, чтобы указать любопытным читателям на этот факт.

Вариант фабричного метода — сбор всех соотносящихся методов-фабрик вместе в виде методов экземпляра специального фабричного объекта. Это полезно, когда у вас есть несколько конкретных классов, которые изменяются в одно время. Например, для каждой операционной системы могут существовать разные фабричные объекты, используемые для различных системных вызовов.

Внутренняя фабрика

Что вы делаете, когда создание вспомогательного объекта вынесено в приватный метод, но слишком сложно, или его надо изменять в субклассах? Напишите метод, который создает и возвращает новый объект.

Внутренние фабрики часто используются при инициализации. Отправная точка возвращающего метода — “ленивая” инициализация переменной.

```
getX() {
  if (x == null)
    x= ...;
  return x;
}
```

Этого достаточно для одного метода. Если вычисления `x` сложны, результативней передать это внутренней фабрике.

```
getX() {
  if (x == null)
    x= computeX();
  return x;
}
```

Внутренние фабрики предполагают, что они будут переопределены в субклассах. Вычисление, использующее одни и те же алгоритмы на разных структурах данных, может быть выражено посредством внутренних фабрик. Вы можете также передать структуры данных в виде параметров вспомогательного объекта.

Метод доступа к коллекциям

Допустим, у вас есть объект, содержащий коллекцию. Как вы обеспечите к ней доступ? Простейшее решение — предоставить возвращающий метод для коллекции.

```
List<Book> getBooks() {
  return books;
}
```

Это дает клиентам максимум гибкости, но создает ряд проблем. Внутреннее состояние, которое зависит от содержимого коллекции, может быть обновлено без вашего ведома, если вы возвращаете коллекцию целиком. Предоставляя такой многоцелевой доступ, вы также упускаете возможность создания ясного, выразительного протокола для объектов.

Одна из альтернатив — упаковать коллекцию в другую, немодифицируемую коллекцию до того, как вы возвратите ее. В конце концов, упаковщик будет коллекцией только для компилятора. Если кто-либо попытается модифицировать коллекцию, будет сгенерировано исключение. Отладка такой ошибки, особенно при производстве кода, достаточно дорого стоит.

```
List<Book> getBooks() {
  return Collections.unmodifiableList(books);
}
```

Вместо этого я предлагаю методы, предоставляющие четко ограниченный доступ к информации внутри коллекции.

```
void addBook(Book arrival) {
  books.add(arrival);
}
int bookCount() {
  return books.size();
}
```

Если клиентам необходимо просматривать элементы коллекции в цикле, напишите метод, возвращающий итератор.

```
Iterator getBooks() {
  return books.iterator();
}
```

Это предотвращает клиентов от модификации коллекции, исключая только ту досадную операцию `remove()` в объекте `Iterator`. Если вам нужна гарантия, что клиенты не будут менять содержимое коллекции, возвращайте итератор, генерирующий исключение в случае удаления элемента. Однако нотификация во время исполнения рискованна и потенциально дорого стоит с точки зрения отладки.

```
Iterator<Book> getBooks() {
  final Iterator<Book> reader= books.iterator();
  return new Iterator<Book>() {

    public boolean hasNext() {
        return reader.hasNext();
   }
   public Book next() {
        return reader.next();
   }
   public void remove() {
        throw new UnsupportedOperationException();
   }
  };
}
```

Если вы обнаружили, что копируете большинство протоколов, относящихся к коллекции, похоже, что существует проблема в конструкции. Если объект сделал много работы для клиентов, не стоит предлагать столько доступа к его внутренним данным.

Метод установки булева значения

В каком порядке лучше устанавливать булевы состояния? Простейшее, неприкрытое решение состоит в подходе, приведенном ниже.

```
void setValid(boolean newState) {
...
}
```

Если клиентам нужна гибкость при доступе, то этот стиль хорош. Однако, когда все вызовы устанавливающего метода — это константы `true` или `false`, можно предложить более выразительный интерфейс, предоставив два метода, каждый для одного булева значения:

```
void valid() {...
void invalid() {...
```

Код, использующий этот интерфейс, лучше читается, и в нем легче понять, какое состояние устанавливается в данный момент. Но, если вы видите код, подобный

```
...
if (...boolean expression...)
  cache.valid();
else
  cache.invalid();
```

возвратитесь и используйте `setValidity(boolean)` вместо этого.

Метод запроса

Иногда объекту нужно принять решение, основываясь на состоянии другого объекта. Это не идеально, если другой объект может вынести решение сам для себя. Но, когда объект должен предлагать критерий решения как часть протокола, сделайте это с помощью метода с префиксом “is” или “have”.

Если один объект имеет много логики, зависящей от состояния другого объекта, это указывает на то, что она находится не на своем месте. Например, если метод выглядит как

```
if (widget.isVisible())
  widget.doSomething();
else
  widget.doSomethingElse();
```

то объекту `widget`, скорее всего, не хватает метода.

Попробуйте переместить логику и посмотрите, не лучше ли она будет читаться. Иногда такие перемещения конфликтуют с вашими предположениями относительно ответственности разных объектов за различные части вычислений. Верьте вашим глазам и действуйте на основании увиденного — обычно это улучшает конструкцию. Результат читается лучше и может быть более широко использован, чем жестко зафиксированная картина, выдаваемая прошлым опытом в аванс.

Метод эквивалентности

Если два объекта нужно проверить на эквивалентность, например, когда они используются в качестве ключей в ассоциативном массиве, но их тождество недоказуемо, реализуйте `equals()` и `hashCode()`.Так как два эквивалентных объекта имеют одинаковое значение хеша, вычисляйте его, используя только данные, задействованные в сравнении. Например, если вы пишете финансовое ПО, финансовые инструменты в нем могут иметь серийные номера. Это может привести к написанию метода, приведенного ниже.

```
Instrument
public boolean equals(Object other) {
```

```
  if (! other instanceof Instrument)
    return false;
  Instrument instrument= (Instrument) other;
  return getSerialNumber().equals(instrument.getSerialNumber());
}
```

Обратите внимание на сторожевой пункт в начале метода. Теоретически любые два объекта могут быть проверены на эквивалентность, так что ваш код должен быть подготовлен к этому заранее. Если вы узнаете, что межклассовые сравнения являются признаком программной ошибки, вы уберете сторожевой пункт и позволите генерироваться `ClassCastException`. Либо же формируйте `IllegalArgumentException` внутри сторожевого пункта. Так как серийные номера — единственное, что используется при сравнении, то они и будут данными, на основании которых будет вычисляться хеш.

```
Instrument
public int hashCode() {
  return getSerialNumber.hashCode();
}
```

Заметьте, что для небольших наборов данных 0 работает так же хорошо, как и хеш-код.

Весь вопрос эквивалентности казался более важным двадцать лет назад. Я могу вспомнить, как на детальную разработку схем эквивалентности уходила куча времени. Тогда была популярна карикатура, демонстрирующая двух обедающих мужчин в зале буфета. Один из них говорит официанту: “Я буду то же, что и он”, после чего официант хватает тарелку второго посетителя и швыряет ее к первому.

Сегодняшние `equals()` и `hashCode()` — рудиментарные остатки того дела с эквивалентностью. Если вы собираетесь использовать их, следуйте правилам. В противном случае могут возникнуть странные дефекты — например, вы помещаете объект в коллекцию и не можете получить его сразу же после этого.

Есть и другой повод для беспокойства. Являются ли два постоянных эквивалентных объекта одним и тем же объектом? Например, выделение `Instrument` в фабричном методе инициирует проблему

```
Instrument
static Instrument create(String serialNumber) {
  if (cache.containsKey(serialNumber))
    return cache.get(serialNumber);
  Instrument result= new Instrument(serialNumber);
  cache.put(serialNumber, result);
  return result;
}
```

Возвращающий метод

Один из способов обеспечить доступ к состоянию объекта — предоставить метод, возвращающий его состояние. По соглашению, в Java такие методы имеют префикс “get”. Например:

```
int getX() {
  return x;
}
```

Это соглашение — форма метаданных. Я недавно пытался именовать возвращающие методы просто по имени возвращаемой переменной, но вскоре прекратил эти попытки. Если читатели находят более легким читать “getX”, чем “x”, то, каким бы ни было мое персональное мнение, лучше делать так, как они ожидают.

Написание возвращающих методов не столь интересно, как решение об их видимости, или вопрос о том, писать ли их вообще. Следуя принципу объединения логики и данных, необходимость публичных и пакетных возвращающих методов являет собой подсказку, что логика должна быть где-то в другом месте. Вместо написания возвращающего метода попытайтесь переместить логику, использующую данные. Однако существует пара исключений относительно моего нежелания использовать видимые возвращающие методы. Одно из них проявляется, когда у меня есть набор алгоритмов, расположенных в отдельном объекте. Алгоритмы требуют доступ к данным и нуждаются в возвращающем методе для их получения. Другое исключение проявляется, если мне нужен публичный метод, и само собой случается, что он должен возвращать значение поля. Наконец, возвращающие методы, вызываемые инструментарием, часто должны быть публичными.

Внутренние возвращающие методы (частные или защищенные) полезны для воплощения “ленивой” инициализации или кэширования. Как и для всех дополнительных абстракций, этих доработок можно избегать, если на самом деле в них нет нужды.

Устанавливающий метод

Если вам нужен метод, устанавливающий значение поля, добавляйте к его имени префикс “set”. Например:

```
void setX(int newX) {
x= newX;
}
```

Я еще менее охотно делаю видимыми устанавливающие методы, чем возвращающие. Устанавливающие методы называются по способу реализации, а не по замыслу. Если полезная часть интерфейса лучше реализуется с помощью устанавливающего метода, это прекрасно, но имя методу должно даваться с точки зрения клиентского

кода. Понять проблему, решаемую клиентом, лучше, установив значение и предоставив метод, адресующийся напрямую к проблеме.

Использование устанавливающего метода как части интерфейса позволяет реализации просочиться наружу.

```
paragraph.setJustification(Paragraph.CENTERED);
```

Наименование интерфейса в соответствии с назначением метода помогает коду говорить:

```
paragraph.centered();
```

даже если реализация `centered()` является устанавливающим методом.

```
Paragraph:centered() {
  setJustification(CENTERED);
}
```

Устанавливающие методы, используемые внутренне (частные или защищенные), могут быть ценны, допустим, для обновления зависимой информации. Наш параграф, например, может требовать обновления изображения, когда выравнивание меняется. Это может быть реализовано в устанавливающем методе.

```
private void setJustification(...) {
...
  redisplay();
}
```

При таком использовании устанавливающий метод действует как простой движок ограничений, гарантируя, что если меняются *эти* данные, то *те* зависимые данные меняются соответственно (в данном случае это содержимое параграфа и информация, отображаемая на экране).

Устанавливающие методы делают код хрупким. Есть принцип — избегать дистанционных действий. Если объект A ссылается на внутреннее представление объекта B, то изменение в коде B также потребует изменения в коде A — не потому, что A изменился фундаментальным образом, но потому, что посылки, на которых основывался объект A, изменились. Логику и данные лучше совмещать. Возможно, данные следует присвоить A, или B должен предложить более осмысленный протокол. Как и с возвращающими методами, если у вас есть инструментарий, которому нужно вызывать устанавливающие методы, пометьте их “Только для инструментария” и сделайте публичными. Предоставьте читателям более модулированный и общительный интерфейс.

Безопасное копирование

При использовании возвращающих или устанавливающих методов возникают потенциальные проблемы с наложением, когда два объекта думают, что имеют эксклюзивный доступ к третьему объекту. Такие ситуации — симптом глубоких проблем в конструкции, таких как недостаток ясности при определении объектов, ответственных за данные. Однако некоторых дефектов можно избежать, делая копию объекта перед его возвращением или сохранением.

```
List<Book> getBooks() {
  List<Book> result= new ArrayList<Book>();
  result.addAll(books);
  return result;
}
```

В таком случае, вероятно, лучше предоставить методы доступа к коллекции. Однако, если требуется доступ к коллекции целиком, то это безопасный способ.

Устанавливающие методы могут быть также написаны с помощью безопасных копий.

```
void setBooks(List<Book> newBooks) {
  books= new ArrayList<Book>();
  books.addAll(newBooks);
}
```

Помнится, я просматривал одну банковскую систему, в которой было слишком много безопасных копий. Каждый метод доступа (возвращающий или устанавливающий) имел две версии, одну из них — безопасную. Чтобы устранить дефекты наложения, огромные структуры объектов копировались каждый раз при вызове безопасного метода. Система была слишком медленной, и клиенты склонялись к использованию небезопасных версий, в результате чего были проблемы с наложением. Стоящие за этим недостатки конструкции — а именно отсутствие достаточно осмысленного протокола — никогда не обсуждались.

Безопасное копирование — это исключительно полумера, используемая, чтобы защитить код от неконтролируемого внешнего доступа. Она не должна быть перманентной частью основной семантики реализации. Постоянные объекты и составные методы предоставляют более простой и коммуникативный интерфейс, менее предрасположенный к ошибкам.

Выводы

В данной главе были описаны шаблоны создания, использующие средства языка Java. В следующей главе речь пойдет о шаблонах применения классов коллекций.

Глава 9

Коллекции

Должен признаться, я не ожидал, что эта глава будет такой большой. Когда я начинал писать ее, то думал, что смогу обойтись API-документом — типами и операциями. Основная идея проста: коллекции отличают объекты в коллекции от тех, что не в коллекции. Что тут еще можно сказать?

Но я обнаружил, что коллекции — значительно более богатая тема, чем я даже подозревал, как по своей структуре, так и для выражения своих намерений. Концепция коллекций совмещает несколько различных метафор. Метафора, на которой делается ударение, изменяет то, как используются коллекции. Каждый из интерфейсов коллекций обыгрывает различные вариации на тему множества объектов. Каждая реализация также взаимодействует по-особому, в основном по отношению к производительности. Таким образом, создание коллекций — это значительная часть обучения умению передавать свои мысли в коде.

Поведение, подобное коллекциям, давно было реализовано путем использования ссылок в самих структурах данных: каждая страница документа может иметь ссылки на предыдущую и последующую страницы. В последнее время стиль программирования склонился к использованию отдельного объекта для коллекции, который ссылается на элементы. Это позволяет гибко разместить тот же элемент в нескольких различных коллекциях без модификации самого объекта.

Коллекции важны, поскольку они являются способом выразить одно из самых фундаментальных свойств вариации в программировании — вариацию числа. Вариация лежит в основе данных, выраженная в помещении данных в коллекцию. Подробные детали коллекции в значительной мере относятся к намерениям создателя кода по отношению к читателю.

Есть старое (по компьютерным меркам) выражение, что существует только три важных числа — “ноль”, “один” и “много” (это высказывание явно не принадлежит аналитику). Если отсутствие поля соответствует “нолю”, и присутствие — “одному”, то коллекция как раз относится к случаю “много”.

Коллекции обитают в странном мире — на половине пути от конструкций языка программирования к библиотекам. Коллекции настолько универсально полезны, а их использование настолько хорошо понятно, что, кажется, уже не за горами универсальный язык программирования, который позволит использовать выражения вроде `plural unique Book books;` вместо современного `Collection<Book> books = new HashSet<Book>();`. Поскольку коллекции являются первосортными

элементами языка, то тем более важно знать, как использовать вашу библиотеку коллекций для выражения общих идей самым прямолинейным образом.

Оставшаяся часть этой главы разделена на шесть частей: метафоры, на которых строятся коллекции; сущности, которые можно выразить с помощью использования коллекций; интерфейсы коллекций, и что они значат для читателя; реализации коллекций, и о чем они говорят; обзор функций, доступных для класса `Collections`, и, наконец, обсуждение расширения коллекций путем наследования.

Метафоры

Как уже отмечалось выше, коллекции являются смесью различных метафор. Первая — это переменная с несколькими значениями. Смысл такой переменной, которая ссылается на коллекцию, именно в том, что переменная ссылается на несколько объектов одновременно. С этой точки зрения коллекция перестает существовать как отдельный объект. Сущность коллекции нас больше не интересует — только объекты, на которые она ссылается. Как и со всеми другими переменными, можно присвоить значение многозначной переменной (добавить или удалить элементы), получить ее значение, или посылать переменной сообщения (с помощью цикла `for`).

Метафора многозначной переменной разбивается в Java, поскольку коллекции являются отдельным объектом со своим именем. Вторая метафора смешивает понятие коллекций и понятие объектов — т.е. коллекции являются, помимо прочего, объектами. Можно получить коллекцию, передать ее дальше, проверить ее на равенство и посылать ей сообщения. Коллекции могут быть разделены между объектами, хотя это создает возможность возникновения проблемы двойников. Поскольку коллекции являются набором взаимосвязанных интерфейсов и реализаций, они открыты для расширения — как путем расширения интерфейсов, так и новыми реализациями. Так что, так же как и коллекции “являются” многозначными переменными, они “являются” и объектами.

Комбинация двух метафор приводит к еще более странным эффектам. Поскольку коллекция реализована как объект, она может быть передана по цепочке, и таким образом получится эквивалент вызова по ссылке, когда вместо передачи в подпрограмму значения переменной передается сама переменная. Изменение значения переменной оказывает эффект и на вызывающую процедуру. Вызов по ссылке вышел из моды несколько десятилетий тому назад из-за возможности непреднамеренных побочных эффектов. Было сложно отладить программу, когда нет уверенности в том, в каком из всех возможных мест переменная может быть модифицирована. Существуют некоторые соглашения программирования относительно коллекций, которые помогают избежать ситуаций, когда сложно прочитать код и предсказать, когда коллекция может быть модифицирована.

Третья метафора, полезная для рассуждений о коллекциях, — это думать о них как о математических множествах. Коллекция — это набор объектов, так же как матема-

тическое множество — это набор элементов. Множество делит мир на вещи, которые содержатся в множестве, и объекты, которые в нем не содержатся. Две базовые операции над математическими множествами — это поиск их мощности (соответствует методу `size()` для коллекций) и проверка принадлежности (представлена методом `contains()`).

Математическая метафора только приближенно относится к коллекциям. Другие главные операции для множеств — объединение, пересечение, разность и дополнение — не имеют для коллекций прямых эквивалентов. Нет ли их потому, что эти операции изначально менее полезны, или они не используются, потому что недоступны, — это может вызвать интерес в качестве дискуссии.

Сущности

Коллекции используются для выражения нескольких ортогональных концепций в программах. В принципе можно выразить себя точно так, как мы этого хотим. Для коллекций это значит использование самого общего из доступных интерфейса при определении и самого специфического класса при реализации. Однако это не абсолютное правило. Я внимательно прошелся по JUnit и обобщил все определения переменных. Результат показал отсутствие однородности. Радость иметь один и тот же объект объявленным в одном месте как `Iterable`, в другом — как `Collection` и как `List` где-то еще, делало чтение кода более сложным без какой-либо уважительной причины. Было бы проще, если бы каждая переменная была определена как `List`.

Первая сущность, выражаемая коллекциями, — это их размер. Массивы (которые являются примитивными коллекциям) имеют фиксированный размер, задаваемый при создании массива. Большинство коллекций могут изменять размер после того, как были созданы.

Вторая сущность, выражаемая через коллекции, — это то, имеет ли значение порядок элементов. Существуют вычисления, в которых элементы влияют один на другой, или иногда внешние вычисления придают значение порядку вызовов для коллекции, которая сохраняет порядок элементов. Порядок может быть таким, в котором элементы были добавлены, или может быть представлен каким-то внешним влиянием, таким, как лексикографическое сравнение.

Следующая сущность, которой может быть выражена коллекция, — уникальность элементов. Есть вычисления, в которых наличие или отсутствие элемента существенно. Другие предполагают, что элемент может быть представлен в коллекции несколько раз, чтобы вычисление было правильным.

Как осуществляется доступ к элементам? Иногда достаточно пройтись по всем элементам, делая некоторые вычисления с одним из них в каждой итерации. Если линейный поиск достаточно быстрый, то базовый интерфейс `Collection` вполне подходит. Если коллекция увеличивается слишком быстро, может оказаться важным

быть в состоянии проверить наличие или получить доступ к элементу по ключу, что предполагает `Set` или `Map`. Время и занимаемое место могут быть оптимизированы с помощью продуманного выбора коллекции.

Производительность

Большинство программистов, как правило, не должны беспокоиться о производительности мелкомасштабных операций. Это хорошая новость по сравнению с былыми временами, когда настройка производительности была ежедневной заботой. Однако вычислительные ресурсы не бесконечны. Когда опыт подсказывает, что нужно улучшить производительность, и измерения показывают, какое место является узким, важно ясно выразить связанные с производительностью решения. Часто лучшая производительность приводит к снижению какого-то другого качества кода, как читабельность или гибкость. Важно заплатить, насколько это возможно, меньше за требуемую производительность.

Кодирование в интересах производительности может нарушить принцип локального влияния. Небольшое изменение в одной части программы может снизить производительность в другой. Если метод работает эффективно, только если передаваемая ему коллекция может быстро проверить вхождение объекта, тогда невинная замена `ArrayList` на `HashSet` во всей остальной программе может сделать метод недопустимо медленным. Удаленное влияние — это другой аргумент для внимательного кодирования, когда кодирование осуществляется в целях производительности.

Производительность связана с коллекциями, поскольку большинство коллекций может расти безгранично. Структура данных, содержащая символы, которые я набираю прямо сейчас, должна быть в состоянии принять миллионы символов. Я хочу добавлять миллионный символ точно так же быстро, как и первый.

Моя общая стратегия в отношении кодирования производительности заключается в следующем: сначала использовать самую простую возможную реализацию, а затем выбрать более специальный класс коллекции, когда это становится необходимым. Когда я принимаю решения, связанные с производительностью, я пытаюсь локализовать их настолько, насколько это возможно, даже если это требует некоторых изменений в проекте. Затем, когда производительность снова становится достаточно хорошей, я завершаю настройки.

Интерфейсы

Читатели основанного на коллекциях кода ищут ответы на различные вопросы, когда они смотрят на интерфейсы, которые декларируются для своих переменных, и на те реализации, которые вы выбираете для переменных. Определение интерфейса

рассказывает читателю о коллекции: имеет ли она определенный порядок, могут ли в ней дублироваться элементы, и можно ли каким-то образом получить элемент напрямую, через ключ, или для этого нужно выполнить итерацию.

Ниже описаны следующие интерфейсы.

- `Array`. Массивы являются самыми простыми и наименее гибкими коллекциями: фиксированный размер, простой синтаксис доступа и быстрое обращение.
- `Iterable`. Базовый интерфейс коллекций позволяет использовать коллекции для итерации, но больше ни для чего другого.
- `Collection`. Предоставляет добавление, удаление и проверку наличия элемента.
- `List`. Коллекция, элементы которой упорядочены и могут быть доступны по своей позиции в коллекции ("дайте мне третий элемент").
- `Set`. Коллекция без повторяющихся элементов.
- `SortedSet`. Упорядоченная коллекция без повторений.
- `Map`. Коллекция, элементы которой хранятся и извлекаются по ключу.

Array

Массивы — это самый простой интерфейс к коллекциям. К сожалению, они не располагают таким же протоколом, как и остальные коллекции, так что сложнее перейти от массивов к коллекциям, чем от одного типа коллекций к другому. В отличие от большинства коллекций размер массива зафиксирован с момента создания. Массивы также отличаются тем, что они встроены в язык, а не предоставляются библиотекой.

Массивы более эффективны по времени и занимаемой памяти, чем другие коллекции, для простых операций. Замеры времени, которые я проводил, чтобы утверждать это, указывают, что запросы к массиву `(elements[i])` более чем в десять раз быстрее, чем эквивалентная операция в `ArrayList-(elements.get(i))`. (Поскольку эти значения немного изменяются в различных операционных средах, то нужно произвести замеры самостоятельно, чтобы наверняка знать разницу в производительности.) Гибкость других классов коллекций делает их более важными в большинстве случаев, но массивы являются удобным приемом, чтобы взвинтить производительность, когда она нужна вам в небольшой части программы.

Iterable

Определить переменную как `Iterable` обозначает только то, что она может содержать множество значений. `Iterable` — это основа для конструкции цикла `loop` в Java 5. Каждый объект, описанный как `Iterable`, может быть использован в цикле `loop`. Это реализовано через неявный вызов метода `iterator()`.

Один из вопросов для взаимодействия при использовании коллекций — это собирается ли клиент модифицировать их. К сожалению, `Iterable` и его "помощник", `Iterator`, не предоставляют способа для декларативного указания факта, что коллекция не будет модифицирована. Как только у вас есть `Iterator`, можно вызвать метод `remove()`, который удаляет элемент из соответствующего `Iterable`. В то время как ваши `Iterable` находятся в безопасности от добавления элементов, они могут удалять элементы без нотификации объекта, который является владельцем коллекции.

Есть несколько путей для того, чтобы убедиться, что коллекция не модифицирована: создание оболочки из немодифицируемой коллекции, создание собственного итератора, который вызывает исключение, когда клиент пытается модифицировать коллекцию, или возврат безопасной копии.

Интерфейс `Iterable` прост. Он даже не позволяет вам измерять размер экземпляров. Все, что здесь можно сделать, — это перемещаться по элементам. Субинтерфейсы `Iterable` предоставляют более полезное поведение.

Collection

Коллекции происходят от `Iterable`, но добавляют методы для добавления, удаления, поиска и подсчета элементов. Определяя переменную или метод как `Collection`, остается много опций для класса реализации. Оставляя определение наиболее общего вида, мы сохраняем свободу позже изменять классы реализации без необходимости выискивать изменения по всему коду.

Коллекции отчасти похожи на математические записи множеств, исключая то, что операции, эквивалентные объединению, пересечению и разности (`addAll()`, `retainAll()` и `removeAll()`), модифицируют исходные данные, а не возвращают заново распределенную результирующую коллекцию.

List

`List` добавляет к `Collection` идею о том, что элементы располагаются в определенном порядке. Можно получить элемент, указав его позицию в коллекции. Постоянная последовательность важна, когда элементы коллекции взаимодействуют друг с другом. Например, очередь сообщений, которая должна быть обработана в порядке поступления, должна быть сохранена в структуре типа `List`.

Set

`Set` — это коллекция, которая не содержит повторений (элементов, которые будут возвращать истину при проверке `equal()` с другим элементом). Это близко соответствует математическому значению множества, хотя метафора тонка, поскольку добавление элемента в `Set` модифицирует коллекцию, вместо того, чтобы возвращать новую коллекцию, которая бы содержала новый элемент.

Set отбрасывает информацию, которую хранят большинство других коллекций: число раз, которое встречается в заданный элемент. Это не вызывает проблем в случаях, когда нас интересует присутствие или отсутствие элемента, но не число раз, сколько он появляется. Например, вас интересуют все авторы книг в библиотеке, но не интересует, сколько книг из них написал каждый. Мы просто хотим знать, кто они. Для такого списка Set является подходящим вариантом.

Элементы в Set не расположены в определенном порядке. Только потому, что мы выбираем их по очереди, еще не значит, что элементы появятся в таком же порядке и в следующий раз. Отсутствие упорядоченности не является ограничением в случаях, когда элементы не взаимодействуют один с другим.

Иногда вам нужно хранить дубликаты в коллекции, но удалять их для определенных операций. Создайте временное хранилище Set и передайте его в операцию.

```
printAuthors(new HashSet<Author>(getAuthors());
```

SortedSet

Упорядоченность и уникальность элементов коллекций не являются взаимно исключающими. Если нужно сохранить коллекцию по порядку, но при этом избежать дупликации, используйте SortedSet. Он хранит упорядоченные и уникальные элементы.

В отличие от упорядочивания в List, которое имеет отношение к порядку, в котором элементы были добавлены, или к явным индексам, переданным в add(int, Object), упорядочивание в SortedSet осуществляется через интерфейс Comparator. В отсутствие явного порядка используется "естественный" порядок элементов. Например, строки сортируются в лексикографическом порядке.

Для сравнения авторов, книги которых находятся в библиотеке, можно использовать SortedSet.

```
public Collection<String> getAlphabeticalAuthors() {
  SortedSet<String> results= new TreeSet<String>();
  for (Book each: getBooks())
    results.add(each.getAuthor());
  return results;
}
```

Этот пример использует сортировку строк по умолчанию. Если Book имеет автора, представленного объектом, то код для их сортировки может выглядеть так, как показано ниже.

```
public Collection<String> getAlphabeticalAuthors() {
  Comparator<Author> sorter= new Comparator<Author>() {
    public int compare(Author o1, Author o2) {
      if (o1.getLastName().equals(o2.getLastName()))
        return o1.getFirstName().compareTo(o2.getFirstName());
      return o1.getLastName().compareTo(o2.getLastName());
    }
  };
```

```
  SortedSet<String> results= new TreeSet<String>(sorter);
  for (Book each: getBooks())
    results.add(each.getAuthor());
  return results;
}
```

Map

Последний интерфейс к коллекциям — это Map, который является гибридным вариантом других интерфейсов. Map хранит значения по ключу, но, в отличие от List, ключ может быть любым объектом, а не только целым числом. Ключи Map уникальны, подобно Set, хотя значения могут содержать повторения. Элементы Map не находятся в определенном порядке, так же как и в Set.

Поскольку Map не совсем такой, как другие интерфейсы коллекций, он стоит обособленно, не наследуя ни один другой интерфейс. Map — это две взаимосвязанные коллекции: одна коллекция ключей и другая — связанная коллекция ключей. Нельзя просто запустить итератор на Map, поскольку не ясно, хотим ли мы просканировать ключи или значения, или и то и другое.

Map часто полезны для воплощения двух шаблонов реализации: внешнего и переменного состояний. Внешнее состояние подразумевает хранение данных специального назначения, относящихся к объекту, отдельно от самого объекта. Один из способов реализовать внешнее состояние — с помощью Map, ключи которого являются объектами, а значения — соответствующими данными. В переменном состоянии различные экземпляры того же класса хранят различные поля данных. Для реализации этого расположите в объекте Map, который отображает строки (представляющие имена виртуальных полей) в значения.

Реализация

Выбор классов реализации для коллекций делается в основном исходя из производительности. Как всегда в вопросах производительности, лучше для начала выбрать простую реализацию, а потом настраивать ее, исходя из практического опыта (рис. 9.1).

В этом разделе каждый интерфейс вводит альтернативную реализацию. Поскольку соображения производительности доминируют в выборе классов реализации, то каждый набор альтернатив сопровождается замерами производительности для важных операций. Приложение “Измерения производительности” содержит исходный код, использованный для замеров и сбора этих данных.

Фактически большинство коллекций реализовано как ArrayList, за которым с большим отрывом следует HashSet (~3400 ссылок на ArrayList в Eclipse+JDK, против ~800 ссылок на HashSet). Это быстрое решение для выбора, какой из этих классов подходит для ваших нужд. Однако для тех случаев, когда опыт подсказывает,

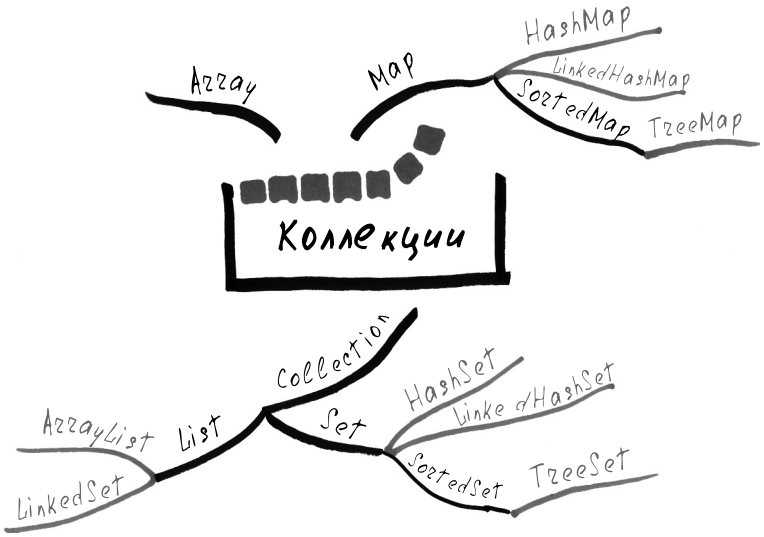


Рис. 9.1. Интерфейсы и классы коллекций

что производительность будет иметь значение, остальная часть этого раздела представляет в деталях альтернативные реализации.

Последний фактор при выборе класса реализации коллекции — это ожидаемый размер коллекции. Приведенные ниже данные показывают производительность коллекций при изменении их размеров от одного до ста тысяч элементов. Если ваша коллекция содержит один или два элемента, результат вашего выбора класса реализации может отличаться от того, который вы сделаете, если число элементов может масштабироваться до одного миллиона. В любом случае преимущества, получаемые от смены класса реализации, часто ограничены, и вам придется рассмотреть более масштабные алгоритмические изменения, если в дальнейшем потребуется улучшить производительность.

`Collection`

Класс, используемый по умолчанию при реализации `Collection` как `ArrayList`. Потенциальная проблема с производительностью `ArrayList` состоит в том, что операции, такие как `contains(Object)`, или другие операции, на которые мы рассчитываем, вроде `remove(Object)`, занимают время, пропорциональное размеру коллекции. Если профиль производительности показывает, что один из этих методов является узким местом, попробуйте заменить `ArrayList` на `HashSet`. Перед тем как это сделать, убедитесь в том, что алгоритм нечувствителен к игнорированию дублирующихся элементов. Если у вас есть данные, где гарантированно нет повторяющихся элементов, переход будет вообще незаметен. На рис. 9.2 сравнивается производительность `ArrayList` и `HashSet` (см. приложение А, чтобы ознакомиться с методологией замеров).

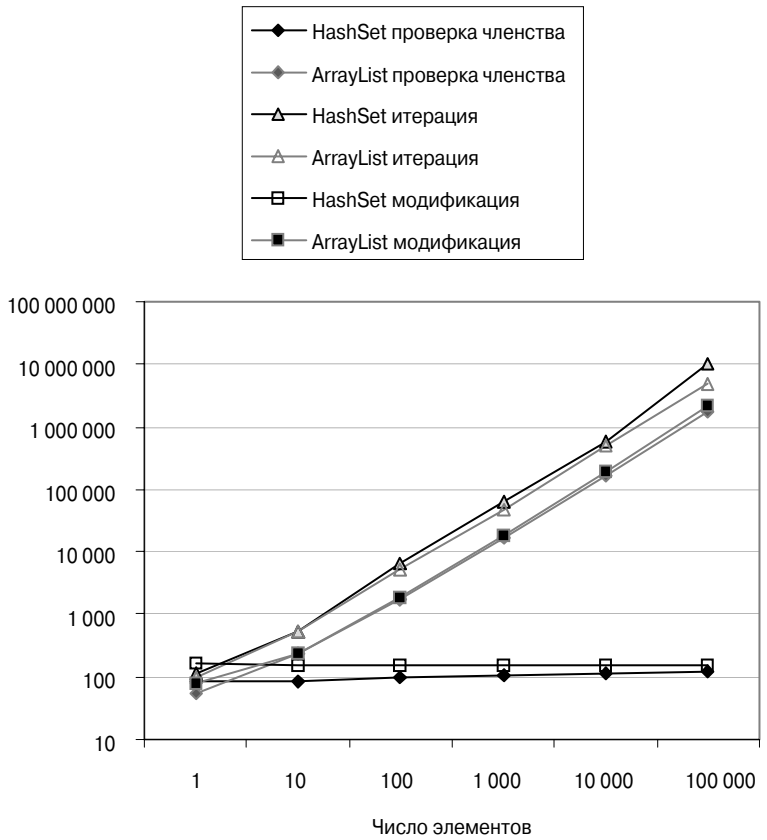


Рис. 9.2. Сравнение `ArrayList` и `HashSet` как реализаций `Collection`

`List`

К протоколу `Collection` `List` добавляет идею о том, что элементы располагаются в определенном порядке. Две часто используемые реализации `List` — `ArrayList` и `LinkedList`.

Профили производительности этих двух реализаций являются зеркальными отображениями друг друга. `ArrayList` обеспечивает быстрый доступ к элементам и медленно добавляет и удаляет элементы, в то время как `LinkedList` медленно получает доступ к элементам, но быстро их добавляет и удаляет (рис. 9.3). Если известно, что в вашем профиле доминирует вызов `add()` или `remove()`, то подумайте о том, чтобы перейти от `ArrayList` к `LinkedList`.

`Set`

Есть три реализации класса `Set`: `HashSet`, `LinkedHashSet` и `TreeSet` (который, по сути, реализует `SortedSet`). `HashSet` — самый быстрый, но его элементы не на-

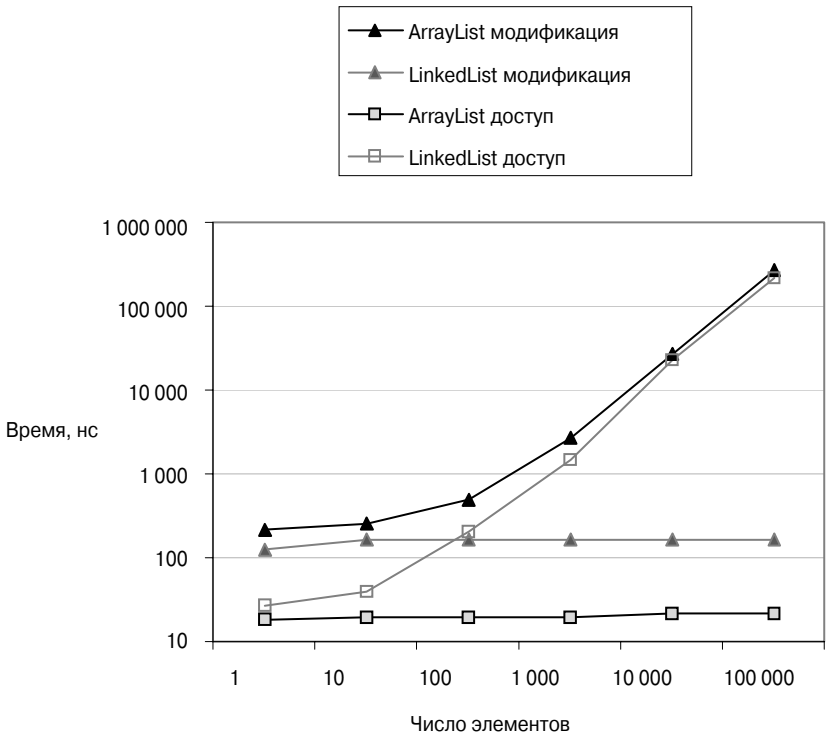


Рис. 9.3. Сравнение `ArrayList` и `LinkedList`

ходятся в каком-то определенном порядке. `LinkedSet` управляет элементами в порядке, в котором они были добавлены, но ценой дополнительных затрат 30% времени для добавления и удаления элементов (рис. 9.4). `TreeSet` хранит свои элементы отсортированными в терминах интерфейса `Comparator`, но это достигается ценой того, что добавление или удаление элемента, так же как и его выборка, занимает время, пропорциональное log n, где n — размер коллекции.

Выберите `LinkedHashSet`, если вам нужно, чтобы порядок элементов был постоянным. Внешние пользователи, например, могут пожелать получать элементы в одном и том же порядке каждый раз.

Map

Реализации `Map` следуют похожему шаблону реализаций для `Set`. `HashMap` — проще и быстрее. `LinkedMap` предохраняет порядок элементов, итерируя по элементам в том же порядке, в котором они были вставлены. `TreeMap` (в действительности реализующий `SortedMap`) проходит по элементам на основе порядка ключей, но это достигается ценой того, что вставка и проверка наличия занимают время, пропорциональное log n. На рис. 9.5 показаны оценки производительности для этих реализаций.

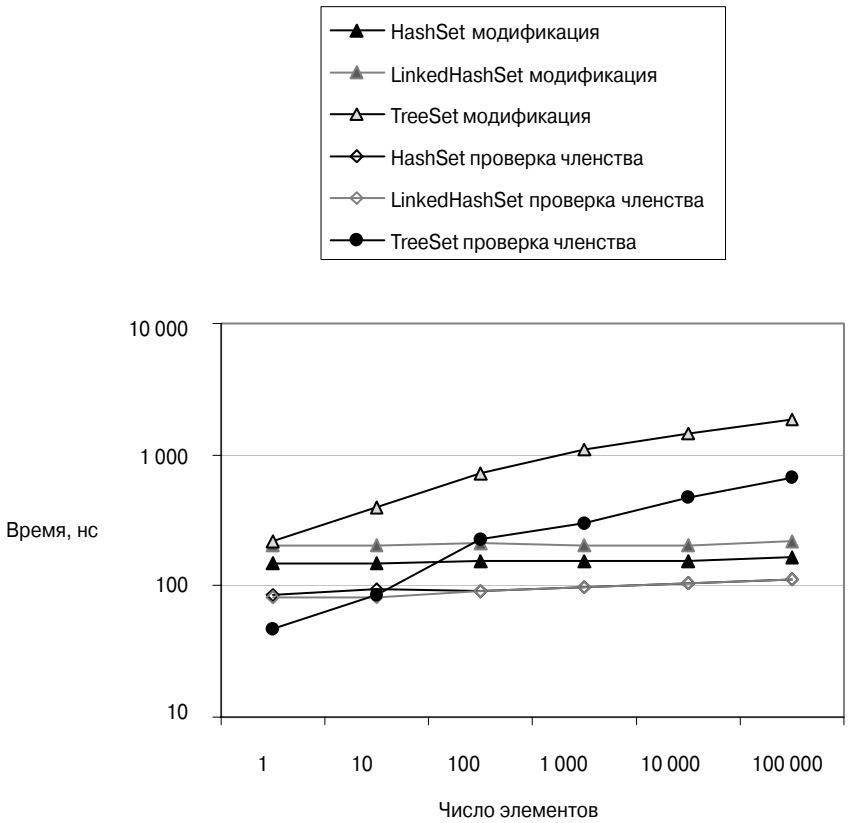


Рис. 9.4. Сравнение реализаций `Set`

Collections

Вспомогательный класс `Collections` — это библиотечный класс, который предоставляет функциональность коллекций, которые не вписываются ни в какой из готовых интерфейсов коллекций. Ниже приводится краткий обзор доступных возможностей.

Поиск

Операция `indexOf()` занимает время, пропорциональное размеру списка. Однако, если элементы отсортированы, двоичный поиск может найти нужный элемент за время $\log_2 n$. Вызовите `Collections.binarySearch(list, element)` для возврата индекса элемента в списке. Если элемента нет в списке, то будет возвращено отрицательное число. Если список не отсортирован, то результат непредсказуем.

Двоичный поиск улучшает производительность только для списков с постоянным временем выборки случайного элемента, таких как `ArrayList` (рис. 9.6).

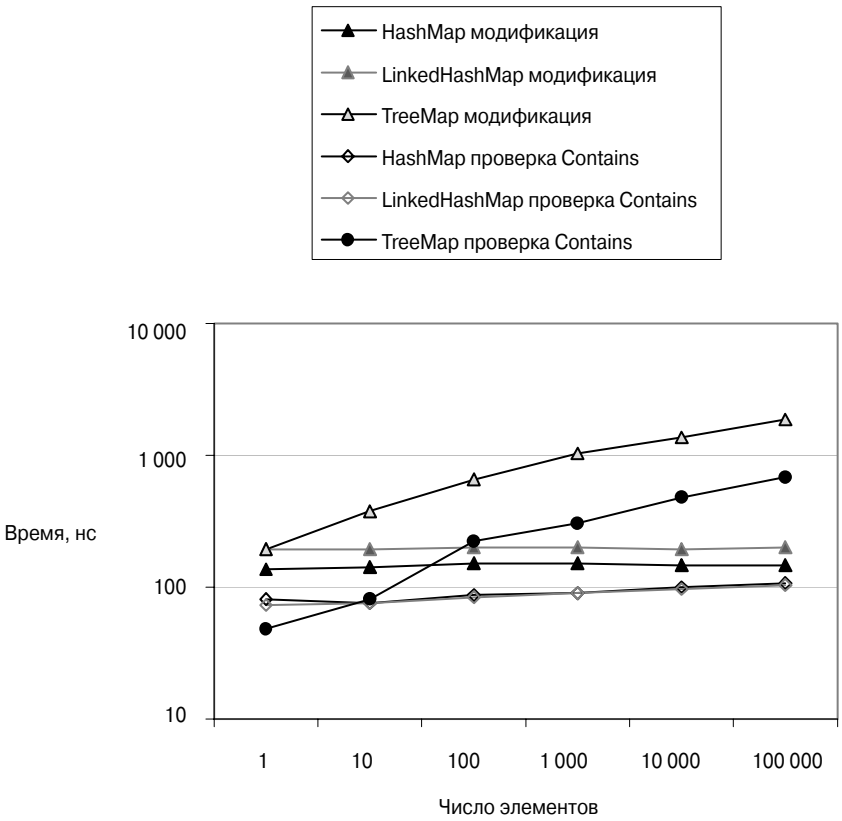


Рис. 9.5. Сравнение реализаций `Map`

Сортировка

`Collections` также предоставляет операции для изменения порядка элементов. `Reverse(list)` обращает порядок всех элементов в списке. `Shuffle(list)` располагает элементы в произвольном порядке. `Sort(list)` и `sort(list, comparator)` располагают элементы в возрастающем порядке. В отличие от двоичного поиска производительность сортировки остается такой же, как и для `ArrayList` и `LinkedList`, поскольку элементы сначала копируются в массив, массив сортируется, а затем элементы копируются обратно (выполните проверку времени сортировки из приложения A, чтобы проверить это).

Немодифицируемые коллекции

Как уже упоминалось выше при обсуждении `Iterable`, даже самые простые интерфейсы коллекций позволяют их модификацию. Если коллекция передается в неизвестный код, то можно быть уверенным в том, что она не будет модифицирована, если вы создадите для нее обертку из субкласса `Collections`, которая вызывает

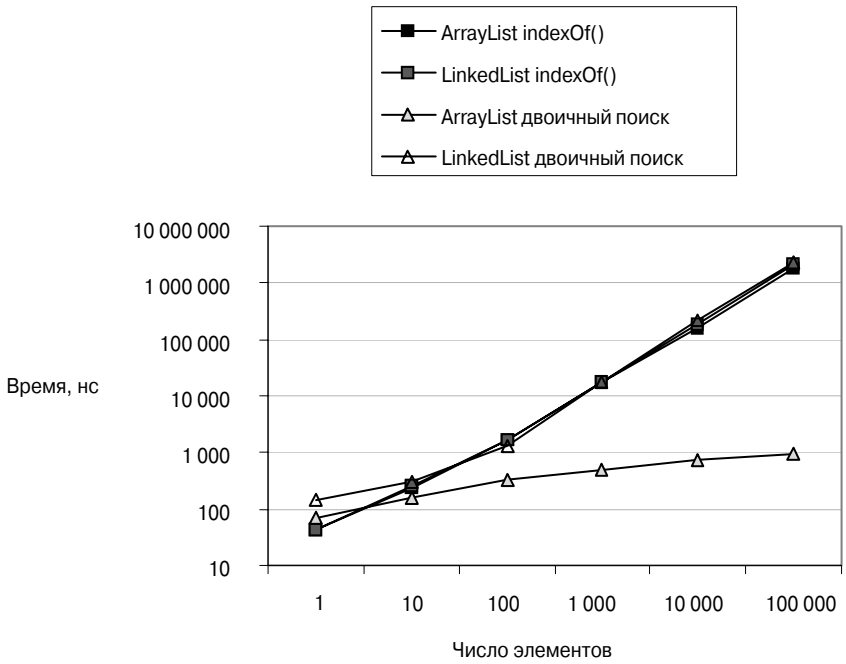


Рис. 9.6. Сравнение *indexOf()* и двоичного поиска

исключение времени выполнения, если клиент пытается модифицировать коллекцию. Есть варианты, которые работают с `Collection`, `List`, `Set` и `Map`.

```
@Test(expected=UnsupportedOperationException.class)
  public void unmodifiableCollectionsThrowExceptions() {
    List<String> l= new ArrayList<String>();
    l.add("a");
    Collection<String> unmodifiable= Collections.unmodifiableCollection(l);
    Iterator<String> all= unmodifiable.iterator();
    all.next();
    all.remove();
}
```

Коллекции из одного элемента

Если у вас есть один элемент и вам нужно передать его в интерфейсе, который ожидает коллекцию, то можно быстро конвертировать его, вызвав `Collections.singleton()`, который возвращает `Set`. Есть также варианты, которые конвертируют в `List` и `Map`. Все возвращаемые коллекции немодифицируемые.

```
@Test public void exampleOfSingletonCollections() {
  Set<String> longWay= new HashSet<String>();
    longWay.add("a");
```

```
    Set<String> shortWay= Collections.singleton("a");
    assertEquals(shortWay, longWay);
}
```

Пустые коллекции

Подобным образом, если вам нужно использовать интерфейс, который ожидает коллекцию, и известно, что в ней нет элементов, `Collections` может создать для вас немодифицируемую пустую коллекцию.

```
@Test public void exampleOfEmptyCollection() {
  assertTrue(Collections.emptyList().isEmpty());
}
```

Расширение коллекций

Я часто видел код, который расширял классы коллекций. Если `Library` содержит список книг, например, то этот класс можно реализовать как расширение `ArrayList`.

```
class Library extends ArrayList {...}
```

Это определение предоставляет реализации `add()` и `remove()`, итерацию и другие операции с коллекциями.

Есть несколько проблем с расширением классов коллекций для получения "коллекционного" поведения. Первая — многие из операций, предоставляемые коллекциями, не будут подходить для клиентов. Например, клиенты в основном не захотят выполнять операции `clear()` или `toArray()`. По крайней мере, такие метафоры очень запутаны и обескураживают. Что еще хуже, все эти операции не должны быть унаследованными, а реализованы заново как вызывающие исключение `UnsupportedOperationException`. Это не очень хорошая идея — унаследовать парочку полезных строк кода и потратить значительно больше строк, удаляя ненужную вам функциональность. Вторая проблема с наследованием от классов коллекций состоит в том, что этот подход исключает наследование, незаменимый ресурс. Чтобы выбрать пару строк из полезной реализации, приходится отказываться от использования наследования на каком-то более высоком уровне.

В такой ситуации лучше делегировать функциональность коллекции, чем наследовать от нее.

```
class Library {
  Collection<Book> books= new ArrayList<Book>();
...
}
```

При таком подходе можно выделить только те операции, которые существенны для вас, и дать им осмысленные имена. Мы оставляем за собой право легко исполь-

зовать наследование, чтобы разделять реализацию с другими классами модели. Если `Library` осуществляет доступ к книгам по нескольким ключам, то можно назвать операции соответственно

```
Book getBookByISBN(ISBN);
Book getBookByID(UniqueID);
```

Расширяйте коллекции, только если создается класс коллекции общего назначения — что-то, чему место в `java.util`. Во всех других случаях храните элементы во внутренней коллекции.

Выводы

В этой главе описывались шаблоны для использования классов коллекций относительно языка Java. Все предыдущие шаблоны были написаны с точки зрения разработки программ, когда простота и ясность взаимодействия снижали стоимость. Но также возможно изменить дизайн приложения в целом. В следующей главе описывается, как модифицировать эти шаблоны при построении инфраструктур, где сложность допустима, если она сохраняет способность и далее развивать инфраструктуру, без необходимости изменять весь код приложения.

Глава 10

Развитие инфраструктур

Предыдущие шаблоны реализации предполагали, что изменение кода достаточно дешево в сравнении с пониманием и передачей его замысла. Это относится к большей части моего опыта разработки. Однако разработка инфраструктур, в которых клиентский код не может быть изменен разработчиками инфраструктуры, предполагает другой подход. Изменение конструкции JUnit, например, в целом легко выполнить, но очень дорого применить на практике, потому что масса создателей инструментариев и тестов должны будут также поменять свой код. Несовместимые обновления так дорого стоят, что мы избегаем их, насколько это возможно.

Когда JUnit 4 был только выпущен, мы потратили примерно половину нашего проектного бюджета на уменьшение стоимости внедрения для наших клиентов. Мы старались убедиться в том, что тесты нового стиля будут работать со старыми инструментариями, а тесты старого стиля — с новыми. Также мы хотели быть уверенными, что сможем свободно вносить в дальнейшем изменения в JUnit, не нарушая клиентский код.

В данной главе вкратце описывается, как шаблоны реализации меняются при разработке инфраструктур. В ней говорится о проблемах разработки инфраструктур, о том, как уменьшить вред от несовместимых обновлений и проектировать инфраструктуры, чтобы избежать таких обновлений. Развитие инфраструктур и одновременная минимизация потерь для клиентов требуют дополнительной сложности в инфраструктуре, уменьшения областей видимости для клиентов и внимания при передаче информации о необходимых изменениях.

Изменение инфраструктур без изменения приложений

Фундаментальная дилемма в разработке и поддержке инфраструктур состоит в том, что они нуждаются в развитии, но при этом есть большая опасность нарушить существующий клиентский код. Идеальное обновление инфраструктуры добавляет новую функциональность без каких-либо изменений в существующей. Однако совместимые обновления не всегда возможны.

Поддержка обратной совместимости часто добавляет сложность в инфраструктуру. В некоторой точке стоимость поддержки идеальной совместимости перевешивает ценность для клиентов. Улучшение экономической стороны разработки инфраструктур зависит от уменьшения вероятности несовместимых обновлений и снижения

стоимости таких обновлений, если они становятся необходимыми. Хотя при традиционной разработке сведение сложности к минимуму — ценная стратегия, делающая код более понятным, в разработке инфраструктур часто финансово более выгодно добавить сложность, чтобы расширить возможности разработчика инфраструктуры к ее улучшению без нарушения клиентского кода.

Хотя совместимость остается важнейшей ценностью для разработки инфраструктур, простота также очень важна. Сложные инфраструктуры менее охотно используются, чем простые. Вводите добавочную сложность только лишь для соблюдения баланса между свободой будущей разработки и стоимостью для клиентов.

Основная линия предыдущих шаблонов реализации — это код, применимый как можно более широко, но в то же время простой для понимания. В разработке инфраструктур практическая применимость кода приносится в жертву будущей свободе изменения конструкции. Например, я стремлюсь объявлять поля защищенными в большей части своего кода, но при разработке инфраструктур я делаю их частными. В результате мои суперклассы труднее использовать клиентам, но это позволяет мне менять представление данных в инфраструктуре, не влияя на приложения клиентов. Инфраструктура со всеми защищенными полями более непосредственна в практике, но будет проявлять большую инерцию к дальнейшему развитию.

В результате появляются инфраструктуры, сложные в развитии, но простые в использовании, и узко применимые на практике, но достаточно легко эволюционирующие. Эти дополнительные ограничения на конструкцию делают разработку инфраструктур более рискованной и дорогой, чем разработку приложений. К счастью, варианты шаблонов реализации могут помочь вам создавать, внедрять и модифицировать инфраструктуры, которые одинаково полезны на практике и пригодны к изменениям.

Несовместимые обновления

Даже если обновление инфраструктуры может потенциально нарушить клиентский код, существуют способы уменьшить стоимость обновления для клиентов. Построение обновлений в виде ряда маленьких шагов предупреждает клиентов о том, что происходит, и предоставляет возможность определиться с инвестициями в модификацию кода. Например, обозначая код как устаревший, но оставляя его функционирующим, вы посылаете клиентам сообщение, что им следует перейти на новый API. Обозначение устаревшего кода — это пример более глобальной стратегии поддержки двух разных архитектур для решения одной проблемы. Параллельные архитектуры добавляют сложность, но уменьшают потери при обновлении.

В Java параллельные архитектуры демонстрируют классы коллекций. Старые классы `Vector` и `Enumerator` были сделаны совместимыми с новыми классами, основанными на `Collection`. Теперь (и в будущем, в случае Java) вы можете запустить код, использующий старые коллекции.

Пакеты могут предложить клиентам способ дискретного доступа к обновлениям. Представляя новые классы в новом пакете, вы можете дать то же имя старым классам. Например, если я могу обновить `org.junit.Assert` до `org.junit.newandimproved.Assert`, то клиентам нужно будет только поменять строку импорта, чтобы приступить к использованию нового класса. Изменение секции импорта менее рискованно, нежели модификация кода. Другая дискретная стратегия — менять либо API, либо реализацию, но не одновременно то и другое. Промежуточный релиз с новым интерфейсом к старому коду или старым интерфейсом к новому коду дает возможность всем — как разработчикам инфраструктуры, так и клиентам — время войти в курс перемен. Остается возможность решить какие-то технические проблемы, которые все же не так существенны.

Классы коллекций выносят на обсуждение другой вопрос относительно обновления инфраструктур: удаление устаревшей функциональности. Часть соглашения между провайдером инфраструктуры и клиентом оговаривает, как часто код клиента должен обновляться, чтобы работать с новым релизом инфраструктуры. Решение Sun состоит в том, что старый код будет работать вечно. А Eclipse, например, соглашается поддерживать совместимость только в релизах одной главной версии. Как провайдер инфраструктуры при выборе стратегии утилизации вы должны осторожно балансировать между необходимостью быстро развивать инфраструктуру и потребностью клиентов работать со стабильной платформой.

Eclipse предоставляет пример иного пути с целью уменьшения стоимости несовместимых обновлений: предлагает автоматизированный инструментарий для обновления клиентского кода. Таким образом, была уменьшена цена потенциально несовместимого обновления версии 2.x до 3.0, когда предполагалось, что большинство плагинов версии 2.x будут без изменений работать в версии 3.0, но также был предложен и инструментарий, делающий плагины 2.x полностью совместимыми. Он добавлял необходимые файлы и перемещал функциональность между файлами, так что старый код по-прежнему работал в новой версии. Комбинируя стратегии, Eclipse во многом поддерживает свою свободу к улучшению быстро развивающейся инфраструктуры, одновременно обеспечивая своим клиентам достаточно стабильную функциональность.

Вы можете уменьшить стоимость изменения кода, дав клиентам возможность переключиться на обновленную версию с помощью простой операции поиска/замены. Если меняется имя метода, будет дешевле для клиентов, если вы оставите аргументы в том же порядке. Наверное, когда-нибудь будет возможно передавать наборы рефакторинга вместе с обновлениями инфраструктур, но сейчас снижение стоимости обновлений может ограничить ваши конструкторские возможности.

Другой фактор при осуществлении несовместимых обновлений — это состав и прирост сообщества ваших клиентов. Если текущие клиенты согласны использовать новейшую функциональность, они будут стремиться предпринять что-то для обновления. Если оно позволяет вам значительно увеличить клиентскую базу, то вы можете подвергнуться обструкции со стороны уже имеющихся клиентов. Сегодняшние

жалобы от 400 потребителей не смотрятся значительными, если за шесть месяцев вы будете иметь 4000 счастливых клиентов. Заботьтесь о реальных потребителях, а не о фантастических, иначе можно остаться с обновленной инфраструктурой и вовсе без клиентов.

Здесь было показано, как организовать несовместимые обновления инфраструктур. Наиболее желательной ситуацией, безоговорочно, является обновление, вводящее новую функциональность, но не влияющее на существующий клиентский код. Остаток этой главы обсуждает шаблоны реализации для написания инфраструктур, которые вы можете обновить без вреда для клиентов.

Осуществление совместимого изменения

Для обновления инфраструктуры, поддерживающей совместимость, клиентский код должен ссылаться на как можно меньшее количество деталей инфраструктуры. Однако этот код все-таки должен ссылаться на некоторые из них, иначе исчезает причина для существования инфраструктуры вообще. В идеальном случае клиенты будут использовать только те детали, которые вы не хотите менять. Но, так как модификации непредсказуемы, нельзя сразу же решить, какие детали менять не захочется. Впрочем, здесь можно поиграть с равновесием: уменьшить количество видимых деталей и раскрыть те, которые наименее склонны к изменению, избавиться от полезной функциональности, сохраняя свободу внесения изменений в конструкцию.

В любом случае придется принять решение относительно вариантов совместимости, которые вы собираетесь предложить. Совместимо ли ваше обновление сверху вниз, и могут ли клиенты все еще вызывать старые методы и передавать старые объекты в инфраструктуру? Совместимо ли обновление снизу вверх, так что вы можете передать объекты нового стиля клиентам и они будут работать, как старые? Стиль или стили совместимости влияют на количество усилий, которые будут вложены в разработку и тестирование обновления. Наш последний релиз, JUnit, например, был гораздо более дорогим, потому что мы решили предложить как обратную, так и прямую совместимость. Пользователи сообщили о нескольких дефектах совместимости, которые мы не учли во время работы над проектом. Тем не менее я удовлетворен нашим решением. Мы имели огромную базу существующих тестов для поддержки и клиентов, которые в большинстве своем не торопились с обновлением. Однако поддержка одновременно прямой и обратной совместимости имела неожиданные последствия.

Большинство инфраструктур на Java представляются в виде объектов, которые создаются, используются и детализируются клиентами. Эта глава описывает, как представлять инфраструктуры таким образом, чтобы клиенты могли применять необходимую функциональность и в то же время разработчик мог продолжать развитие инфраструктуры. Достижение этого равновесия требует внимательного отношения к использованию и созданию объектов и к тому, как структурированы методы.

Библиотечный класс

Один из простых и должным образом "защищенных от будущего" стилей API — это библиотечный класс. Если вы можете представить всю функциональность в виде вызовов процедур с простыми параметрами, то клиенты будут изолированы от будущих перемен. Когда вы выпускаете новую версию библиотечного класса, остается только убедиться, что все существующие методы работают так же, как и раньше. Новая функциональность представлена новыми процедурами или новыми вариантами уже существующих.

Класс `Collections` является примером API, представленного в библиотечном классе. Клиенты используют его, вызывая статические методы, не создавая экземпляр класса. Новые версии классов коллекций добавляют новые статические методы, оставляя существующую функциональность неизменной.

Большая проблема с представлением API в виде библиотечного класса — это ограниченное число концепций и вариантов, которые могут быть легко выражены. Как только вариации функциональности и сочетаний вариаций умножаются, некоторые необходимые процедуры могут быть уничтожены. Кроме того, клиенты могут менять только данные, которые посылаются в инфраструктуру: они не могут встроить вариации в логику.

Объекты

Если вы собираетесь представить инфраструктуру в виде объектов, то перед вами встает сложная задача балансировки простоты и сложности, гибкости и специфичности, чтобы инфраструктура была такой же полезной и стабильной для клиентов, как и способной к эволюции. Уловка заключается в том, чтобы писать инфраструктуры таким образом (насколько это возможно), что клиенты будут ссылаться только на детали, нежелательные для изменений.

Ниже будут обсуждаться четыре проблемы представления инфраструктур как объектов.

- Стиль использования. Будут ли клиенты использовать инфраструктуру, создавая экземпляры объектов и конфигурируя их, и/или переделывая или реализуя ваши объекты?
- Абстракция. Будут ли детали уровня класса представляться в виде интерфейсов или классов? Как будет использоваться видимость, чтобы раскрывать только относительно стабильные детали?
- Создание. Как будут создаваться объекты?
- Методы. Как будут структурированы ваши методы, чтобы быть полезными клиентам и пригодными к изменениям?

Стиль использования

Инфраструктуры могут поддерживать три стиля использования: создание экземпляров, конфигурирование и реализацию. Каждый стиль предлагает различные комбинации полезности, гибкости и стабильности. Можно также слить эти стили в одной инфраструктуре, чтобы обеспечить лучший баланс между свободой проектирования и возможностями для клиентов.

Простейший стиль — создание экземпляров. Когда мне необходим сокет серверного типа, я пишу `new ServerSocket()`. Однажды созданный, объект инфраструктуры работает посредством вызова его методов. Создание экземпляров применимо, когда существует единственный вид вариаций, нужный клиентам, — вариации данных, но не логики.

Конфигурирование — более сложный и гибкий стиль использования, который предполагает создание объекта инфраструктуры клиентом, но клиент должен передавать свои объекты, которые будут вызваны в определенных ситуациях. Например, `TreeSet` может быть вызван с помощью описанного клиентом `Comparator`, чтобы суметь произвести произвольную сортировку элементов.

```
Comparator<Author> byFirstName= new Comparator<Author>() {
  public int compare(Author book1, Author book2) {
    return book1.getFirstName().compareTo(book2.getFirstName());
  }
};
SortedSet<Author> sorted= new TreeSet<Author>(byFirstName);
```

Конфигурирование более гибко, нежели создание экземпляров, потому что оно может вмещать вариации логики так же, как и данных. Однако оно ограничивает свободу конструирования инфраструктуры, поскольку, если вы начали вызывать клиентский объект, нужно будет продолжать вызывать его тем же способом и в то же время, иначе вы рискуете нарушить клиентский код. Другое ограничение конфигурирования — то, что оно может управлять только несколькими измерениями вариабельности. Данный объект может иметь только одну или две опции для конфигурирования, иначе он становится слишком сложным, чтобы быть легким в использовании.

Когда клиентам нужно больше зацепок для их собственной логики, кроме тех, которые предоставлены конфигурированием, тогда можно предложить им реализацию. При использовании этого стиля клиенты создают их собственные классы, которые применяются инфраструктурой. Так как клиентский класс расширяет класс инфраструктуры или реализует ее интерфейс (что лучше выбрать, обсуждается в следующем разделе), то клиент легко может включить нужную ему логику.

Из трех стилей использования объектов реализация имеет наибольшую тенденцию ограничивать будущую свободу конструкции. Каждая деталь предоставленного инфраструктурой суперкласса или интерфейса должна быть сохранена, если вам нужна гарантия, что клиентский код будет продолжать работу. Каждая деталь, открываемая в абстракции инфраструктуры, — это обоюдоострый меч, предлагаю-

щий клиенту место, на которое можно повесить собственный код, но заставляющий разработчика либо поддерживать детали, либо рисковать нарушением клиентского кода.

Пример с `Comparator` демонстрирует простую версию стиля реализации, используемого инфраструктурой. Компаратор `byFirstName` являет собой реализацию абстрактной коллекции инфраструктуры (в этом случае класса). В данном случае реализация проста, так как есть только одна частичка логики, которую нужно внедрить, и она достаточно коротка, чтобы уместиться в одной строке вместе с остальным кодом. Реализации могут быть расположены также во вложенных или в стандартных классах, если они более сложные.

Использование стиля реализации дает гораздо большую масштабируемость, чем конфигурирование, потому что он позволяет ввести любое число независимых вариаций, каждая из которых представлена методом привязки, определенного инфраструктурой.

В `JUnit` смешаны все четыре стиля использования.

- `JUnitCore`. Библиотечный класс со статическим `run(Class...)` методом, запускающим все тесты во всех классах.
- `JUnitCore` также позволяет создавать свои экземпляры, которые обеспечивают лучший контроль над запуском тестов и системой нотификаций.
- Аннотации (annotations) `@Test`, `@Before` и `@After` являются формой конфигурирования на случай, если создателям тестов нужно определить ситуации, в которых будут запускаться части кода.
- Аннотация `@RunWith` является видом реализации, используемым, когда создателям тестов требуется нестандартное поведение во время запуска тестов.

Абстракция

При использовании инфраструктуры в стиле реализации возникает вопрос о представлении абстрактных сущностей в виде интерфейсов или общих суперклассов. Каждый подход имеет свои плюсы и минусы для разработчиков инфраструктур и клиентов. Эти два подхода не исключают друг друга. Инфраструктура может предложить клиентам одновременно интерфейс и реализацию этого интерфейса по умолчанию.

Интерфейс

Значительное преимущество в предоставлении клиентам интерфейсов состоит в том, что в них записано очень немного деталей. Клиенты не могут “случайно” использовать большую часть инфраструктуры, чем позволяет замысел. Эта защита, тем не менее, чего-то стоит. Пока интерфейсы остаются неизменными, все хорошо, но введение нового метода в интерфейс будет нарушать все клиентские реализации интерфейса. Если есть уверенность, что клиенты только используют интерфейс, не реализуя его,

тогда можно вводить новые методы, не вмешиваясь в клиентский код. Несмотря на уязвимость интерфейсов, они широко применяются в мире Java для выражения абстракций, которые существенны сами по себе.

Интерфейсы имеют также тот недостаток, что клиентские классы могут реализовать несколько из них одновременно. Реализация нескольких взаимосвязанных интерфейсов в одном классе может быть ясным и прямым путем коммуникации. Однако класс, который одновременно реализует полностью независимые интерфейсы, возможно, лучше разбить на несколько меньших классов, чтобы передать их назначение более ясно.

Вариация интерфейсов, которая предоставляет дополнительную гибкость ценой некоторой сложности, — это версии интерфейсов. Если вы добавляете операции к интерфейсу, ломается клиентский код. Однако можно создать субинтерфейс и поместить в него новые операции. Клиенты могут передавать объекты, согласуясь с новым интерфейсом, тогда как старый интерфейс остается на месте, и существующий код работает так же, как раньше.

Эта дополнительная гибкость приходит ценой увеличения сложности инфраструктуры. Во время выполнения ей требуется в точности знать, когда нужно вызвать операции нового интерфейса. Например, AWT имеет две версии интерфейса менеджера размещения. В полудюжине мест в AWT появляется код, подобный приведенному ниже.

```
...
if (layout instanceof LayoutManager2) {
  LayoutManager2 layout2= (LayoutManager2) layout;
  layout2.newOperation();
}
...
```

Версии интерфейсов — это разумный компромисс, если вам необходимо ввести новые операции в существующую абстракцию, не повлияв при этом на клиентский код. Это не "долгоиграющее" решение для частого изменения абстракций, потому что оно создает сложности как для разработчиков инфраструктуры, так и для клиентов. Изменение абстракций лучше выражать в суперклассах, если требуется корректное согласование изменений.

Суперкласс

Альтернатива описанию абстракций с помощью интерфейсов — попросить клиентов передать экземпляр класса или одного из субклассов. Преимущества и недостатки этого стиля обратны преимуществам и недостаткам интерфейсов: классы могут определять больше деталей, чем интерфейсы, но добавление операции к суперклассу не нарушает клиентский код. В отличие от интерфейсов, однако, клиентские классы могут расширять только единственный класс инфраструктуры.

Детали суперкласса, видимые клиентам, — это публичные и защищенные методы и поля суперкласса. Каждый такой метод или поле — обещание не изменяться. Если

видимо слишком много деталей, обещаний может быть ужасно много, что существенно ограничивает будущие изменения конструкции.

Осторожность при написании суперкласса может свести эти ограничения к необходимому минимуму, который будет доступен через интерфейс. Поля в инфраструктуре должны быть всегда частными. Если клиентам нужен доступ к данным в полях, обеспечьте его с помощью возвращающих методов. Тщательно изучайте ваши методы и делайте их публичными или, что лучше, защищенными, только при необходимости. Следование этим правилам позволит вам определять суперкласс, который раскрывает лишь чуть больше деталей, чем эквивалентный интерфейс, но дает клиентам бо́льшую гибкость при введении собственной логики.

Ключевое слово `abstract` являет собой возможность сообщить клиентам, где им нужно заполнить логику. Предоставление разумной начальной реализации методов там, где это возможно, дает клиентам возможность легко приступить к их использованию. Однако введение новых абстрактных методов в суперклассе создает несовместимые обновления, так как клиенты должны реализовать методы перед тем, как снова смогут скомпилировать субклассы.

Ключевое слово `final` применительно к классу предотвращает создание субклассов клиентами, принуждая использовать создание экземпляров или конфигурирование. В приложении к методу оно позволяет разработчику инфраструктуры предполагать, что некий код будет выполнен, даже в видимом клиенту методе. Хотя я уважаю стремление разработчиков упрощать стоящие перед ними задачи, я также был разочарован финальными классами и методами. Однажды я провел два дня, напрасно стараясь программно создать SWT-события для тестирования. Именно финальные классы (несущественно, чем они казались мне) помешали мне написать нужный код. Я закончил написанием моих собственных классов событий, которые дублировали SWT-события, так что я мог провести тест без GUI. Оставив `final` на случай, когда оно составит существенный вклад для вас и причинит поменьше проблем клиентам, вы улучшите отношения разработчиков и клиентов инфраструктур.

Во время работы над темой видимости я должен был отметить, что в схеме пакетов Java существует пробел. Инфраструктуры, организованные в несколько пакетов, нуждаются в типе видимости, который означает: “Видимо в инфраструктуре, но не для клиентов”. Одно из решений этой проблемы — разделить пакеты на публичные и внутренние и передать различие, включая имя “internal” во внутренние пути пакетов. В Eclipse, к примеру, вы увидите такие пакеты, как `org.eclipse.jdt...` и `org.eclipse.jdt.internal...`.

Внутренние пакеты являются промежуточной зоной между раскрытием и утаиванием деталей инфраструктуры. Клиенты могут сами выбирать, сколько ответственности они хотят нести за построение кода на потенциально нестабильных частях инфраструктуры. Иногда нужная вам, как клиенту, функциональность есть в инфраструктуре, но она дисквалифицирована (с вашей точки зрения) разработчиками.

Создание экземпляров

Если ваша инфраструктура публикует какие-то конкретные классы, нужно решить, как клиенты будут создавать их экземпляры. Как и с другими конструкторскими решениями при проектировании инфраструктур, вы должны уравновесить всеобщность, сложность, легкость изучения и простоту развития при выборе стиля создания. Четыре стиля, описанных здесь, — это создание экземпляра без клиента, конструкторы, методы-фабрики и объекты-фабрики. Эти стили не исключают один другого. Можно использовать более одного стиля для данного объекта или разные стили для различных частей инфраструктуры.

Запрет создания экземпляров

Простейшая и наименее мощная возможность — запретить клиентам напрямую создавать объекты инфраструктуры. Пример с SWT-событием, приведенный выше, демонстрирует это. Всегда конструируя события в инфраструктуре, разработчики могут гарантировать, что эти события правильно сформированы. Код инфраструктуры может быть проще, если он основан на инвариантах относительно событий.

Недостаток данного подхода — запрещение клиентам создавать экземпляры классов инфраструктуры — препятствует законным способам использования классов, которые не были предусмотрены разработчиками. Для очень сложных задач программирования любое уменьшение сложности приветствуется, и устранение возможности создания объектов клиентом может быть хорошим выбором. Ценность инфраструктуры часто заключается не в том, что изначально предполагалось ее разработчиками. Урезание диапазона возможностей непредвиденного использования снижает вероятность нахождения дополнительных ценных применений инфраструктуры.

Конструкторы

Предложение клиентам возможности создания объектов в конструкторах является простой опцией, но она создает существенные ограничения на будущие изменения. Когда вы публикуете конструктор, вы обещаете, что имя класса, параметры, необходимые для создания, пакет класса и (наибольшее ограничение из всех) конкретный класс возвращаемого объекта не изменятся.

Большинство библиотек Java предлагают создание через конструкторы. С тех пор как Sun опубликовала сведения о том, что списки создаются с помощью `new ArrayList()`, они были соотнесены с классом, названным `ArrayList` в пакете `java.util` без изменений конкретного возвращаемого класса. Это существенные недостатки конструкции, ограничивающие поддержку в неопределенном будущем. Они снижают разнообразие изменений, которые Sun могла бы сделать. Преимущество представления создания объекта в виде конструктора — простота и ясность для клиентов. Если им нужен простой в использовании интерфейс создания и вы не предполагаете обеспечивать возможность менять имя, пакет и конкретный класс ваших абстракций, то конструкторы — разумный выбор.

Статические фабрики

Статические фабрики добавляют сложность в создании объектов для клиентов, но оставляют разработчику инфраструктуры больше свободы в будущих изменениях конструкции. Если клиент создал список с помощью `ArrayList.create()`, вместо использования конструктора, то имя, пакет и конкретный класс возвращаемого объекта могут меняться, не влияя на клиентский код. Следующий шаг — сосредоточение фабричных методов в библиотечном классе `Collections.createArrayList()`. При использовании этого стиля фабрики единственный класс, который нужно оставить в исходном пакете `java.util`, — библиотечный класс. Остальные классы могут перемещаться по необходимости. С другой стороны, чем более абстрактно создание, тем сложнее узнать при чтении кода, какие объекты создаются.

Следующее преимущество фабричных методов в том, что они дают возможность ясно передать клиентам смысл вариаций при конструировании. Назначения двух конструкторов с разными наборами параметров не всегда очевидны, но имя фабричных методов может дать клиентам аргумент, объясняющий, зачем они могли бы создавать объект каждым из способов.

Объект-фабрика

Также можно представить создание экземпляра, посылая сообщения в фабричный объект, вместо вызова статического метода. Например, `CollectionFactory` должна предоставлять методы для создания всех различных видов коллекций. Она может быть использована подобно этому: `Collections.factory().createArrayList()`. Фабричный объект предоставляет даже больше гибкости, чем статическая фабрика, но более сложен для чтения. Чтобы увидеть, какие именно объекты создаются, вам придется проследить за выполнением кода.

Поскольку фабрика доступна только глобально, фабричный объект не дает больше гибкости, чем статические фабричные методы. Фабричные объекты показывают все, на что они способны, когда используются локально. Например, если вы имеете специальные сберегающие пространство коллекции для использования в мобильных устройствах, можно инициализировать объекты, создающие коллекции, фабрикой сберегающих пространство коллекций при запуске на мобильном телефоне и стандартной фабрикой коллекций в случае запуска кода на сервере.

Объекты-фабрики могут быть полезны для создания наборов совмещенных классов. Если виджеты в Wndows работают вместе, но не подходят к виджетам Linux, создание через объект-фабрику даст клиентам способ создания только совместимых классов.

Выводы относительно создания

От того, как вы представите создание объектов в вашей инфраструктуре, зависит, насколько легко ее можно будет использовать и менять. Одной из стратегий является введение фабричных методов для классов, которые предполагается менять, и конструкторов для стабильных классов. Однако также ценна и последовательная стратегия создания объектов с помощью фабричных методов или объектов.

Методы

Методы, не относящиеся к созданию объектов, также влияют на простоту использования и развития инфраструктуры. Общая стратегия остается той же: раскрывать столь мало деталей, сколько достаточно клиентам для решения их проблем. Видимые клиентам вызывающие и устанавливающие методы возможны, только когда структуры данных стабильны. Предлагая клиентам ссылаться на внутренние структуры данных, вы очень сужаете ваш выбор при будущем развитии инфраструктуры. Устанавливающие методы хуже возвращающих в этом отношении. Зачастую возможно найти альтернативный способ вычислить значение, которое сохраняется в поле. Постарайтесь понять, какую проблему решает клиент, устанавливая значение. Вместо публикации устанавливающего метода предоставьте метод, названный по имени проблемы (но не по способу реализации), которую клиенту нужно решить.

Например, при написании графической библиотеки виджетов в классе `Widget` может быть предложен устанавливающий метод `setVisible(boolean)`. Что случится, если будет введено третье состояние, неактивное? Чтобы упростить это для клиента, опубликуйте передающие замысел методы вроде `visible()` и `invisible()`. Они являются тем, что `setVisible()` значит для клиента. Применив данные методы, можно добавить `inactive()` в суперкласс, и это не повлияет на клиентский код.

Абстракции интерфейсов несколько отличаются. Введение `inactive()` в интерфейс разрушит любые клиентские реализации `Widget`. Вместо этого лучше объявить перечисляемый тип `States`, в котором записаны возможные состояния виджетов, и опубликовать метод `setVisible(State)`. Вариант с булевым типом — пример недостаточности информации о конструкции для клиентов. Этот тип предполагает только два возможных состояния. Конструкция с единственным методом и перечисляемым типом в качестве параметра предоставляет свободу добавления других состояний при необходимости.

Это не означает, что возвращающие и устанавливающие методы никогда не должны публиковаться для использования клиентом. Если важная функциональность инфраструктуры реализуется при возвращении или установке поля, публикуйте метод доступа к ней. Но именуйте метод так, чтобы он не раскрывал своей реализации клиентам.

Другая стратегия уровня методов для разработчиков инфраструктур, преследующая цели совместимости, — предоставить значения по умолчанию при добавлении параметров в опубликованные методы. Если вы добавляете параметр к методу, то существующие вызовы метода должны быть модифицированы перед тем, как клиент скомпилирует код. Но можно сохранить клиентский код в рабочем состоянии, если останется старый метод, который будет вызывать новый метод с параметром по умолчанию.

Допустим, к примеру, что в JUnit вам потребовалась возможность передавать `TestResult` в метод, запускающий тесты в классах. Можно всего лишь модифицировать его, добавив этот параметр.

```
public TestResult run(Class... classes) {
....run tests in classes...
}
public void run(TestResult result, Class... classes) {
...run tests in classes...
}
```

Любой клиент, вызвавший `run(Class...)`, должен будет внести изменения, добавив параметр `TestResult`. Однако оригинальный метод может предоставить параметр по умолчанию.

```
public TestResult run(Class... classes) {
  TestResult result= new TestResult();
  run(result, classes);
  return result;
}
```

Предоставляя параметр по умолчанию, вы обеспечиваете работу клиентского кода, даже если интерфейс предлагает новый метод.

Выводы

Разработка инфраструктур и развитие требуют несколько других шаблонов реализации, чем при разработке приложений. Смещение экономического приоритета со стоимости понимания кода в сторону стоимости обновления клиентского кода вызывает существенные сдвиги как в ценностях, так и на практике. При разработке инфраструктур простота, главная направляющая в разработке приложений, имеет меньший приоритет, чем необходимость сохранения степеней свободы для дальнейшего роста. Ситуация усложняется в случае перенесения частей приложений в инфраструктуру. Многие конструкторские решения при проектировании приложений нужно пересмотреть, если вы собираетесь создать эффективную инфраструктуру.

Инфраструктуры развиваются различными путями. Иногда требуется улучшение существующих методов. Иногда вычислениям нужны новые виды параметров. Иногда инфраструктура может быть слегка перенастроена и использована для решения совершенно непредвиденной проблемы. Иногда существующие детали реализации инфраструктуры должны быть опубликованы.

Метафора, хорошо послужившая нам в JUnit, заключается в том, чтобы смотреть на инфраструктуру как на пересечение всей полезной функциональности в целом, а не как на ее объединение. Это задача разработчика инфраструктуры — гарантировать, что клиенты смогут расширить инфраструктуру для решения оставшихся проблем. Это стремление постараться решить широкий диапазон проблем с помощью инфраструктуры. Конфликт заключается в том, что дополнительная функциональность делает инфраструктуру гораздо более сложной к изучению и использованию для всех клиентов.

Если каждый потенциальный пользователь инфраструктуры имеет 90% общих требований и 10% — уникальных, то инфраструктура, полностью удовлетворяющая всех разработчиков, будет гораздо больше, чем инфраструктура, решающая только общие проблемы. Моя цель как разработчика инфраструктуры — удовлетворять все общие нужды, но лишь частично — уникальные. Если все пользователи должны будут добавлять одинаковую функциональность, то она относится к инфраструктуре, а уникальные части лучше отдать тем, кто имеет в них непосредственную необходимость.

Один из способов получать инфраструктуры соответствующего размера — делать их производными от нескольких конкретных примеров, вместо того чтобы начинать с общих случаев. Я написал предка JUnit после полудюжины попыток автоматически протестировать мой код. Каждый вариант решал только ту проблему, которая имелась прямо передо мной в данный момент. Только перспектива писать одинаковый код несколько раз заставила меня увидеть, какие проблемы являются общими для всех тестов и должны быть охвачены инфраструктурой, а какие уникальны в индивидуальных ситуациях.

Заимствуйте концепции вашей инфраструктуры из одной или нескольких ясных и содержательных метафор. Например, если двойная бухгалтерия является метафорой для записи исторической информации, то клиенты будут знать, что искать нужно `Account` и `Transaction`. Сознательно выбирая, применяя метафоры и донося их клиентам, вы делаете инфраструктуру более легкой в изучении, использовании и расширении.

Внедрение инфраструктуры не означает окончания ее развития. Внимательность при конструировании инфраструктуры может быть результатом стабильной базы для клиентских приложений и динамического основания для дальнейшей разработки инфраструктуры.

Приложение А

Замеры производительности

В этом приложении описывается инфраструктура, используемая для сбора данных о производительности коллекций, описанных в главе 9. Вопрос стоит довольно просто: внимательно сравнить время, необходимое для выполнения нескольких операций, в плане масштабируемости. Однако проблема становится более сложной, когда точность таймера намного ниже, чем требуется для выполнения операций. Представленный здесь измеритель времени преодолевает эти проблемы, выполняя операции много раз. Это компенсирует точность таймера, фиксируя время обычных часов для выполнения каждой операции.

Аккуратное измерение операций в оптимизированной реализации Java требует дополнительных знаний как в инфраструктуре, так и в самом тесте. Для получения точных результатов вам нужно знать, что оптимизатор намерен делать с вашим кодом, чтобы быть уверенным, что оптимизация полностью не исключит ваши операции. Если результаты замеров не совпадают с тем, что подсказывает ваша интуиция, то это значит, что надо копать глубже. Либо вы узнаете что-то новое об измерениях, либо о коде, который вы пытаетесь измерить. Приведенный здесь код достаточно хорош для генерации данных, представленных в этой книге. Он может быть сделан более общим. Например, параметры для измерения таймингов представлены как константы, а не как переменные, нет интерфейса командной строки, а отчет прост и печатается на консоль. Один важный навык в программировании — сопоставлять затраты и результат. Изучение того, как использовать шаблоны, должно сопровождаться изучением того, когда их следует использовать, а когда оставить до лучших времен.

Пример

Таймер должен быть в состоянии изменять настолько простые операции, насколько это возможно. Взяв за образец JUnit, тестируемые операции должны быть представлены методами. Экземпляры методов измерителя времени будут созданы с конкретным размером данных, так что замеры времени могут быть сделаны для разных по размеру данных. Например, для замера времени, необходимого для поиска в списке, тест будет выглядеть таким образом:

```
public class ListSearch {
   private List<Integer> numbers;
```

```
   private int probe;

   public ListSearch(int size) {
     numbers= new ArrayList<Integer>();
     for (int i= 0; i < size; i++)
        numbers.add(i);
        probe= size / 2;
     }
   public void search() {
     numbers.contains(probe);
   }
}
```

Результатом работы инфраструктуры с этим классом будет измерение времени, необходимое для выполнения вызова `search()` для наборов размером 1, 10, 100 и так далее элементов.

API

Внешним интерфейсом для таймера является класс `MethodsTimer`. Он создан массивом методов:

```
public class MethodsTimer {
  private final Method[] methods;

  public MethodsTimer(Method[] methods) {
    this.methods= methods;
  }
}
```

Вызывайте `MethodsTimer`, посылая ему `report()`. Например, для замера времени операции, показанного выше `ListSearch`, выполните такой код:

```
public static void main(String[] args) throws Exception {
  MethodsTimer tester= new
    MethodsTimer(ListSearch.class.getDeclaredMethods());
  tester.report();
}
```

Выполнение этого метода завершится печатью результатов на консоли:

```
search 34.89 130.61 989.73 9911.19 97410.83 990953.62
IMPLEMENTATION 133
```

Это значит, что операция поиска занимает 35 наносекунд для списка из одного элемента, 131 наносекунду для десяти элементов и т.д. Таймер не идеально аккуратен. Выполнение его с классом таймера `Nothing` замеряет время для пустого метода, что

теоретически должно давать все нулевые значения. Вместо этого ответы (по крайней мере, на моей машине) отличаются на пару наносекунд:

```
nothing 1.92 -3.24 0.62 0.37 -0.74 2.30
```

Принимайте во внимание эту точность, когда замеряете очень короткие операции. Например, для замера доступа к массиву пришлось написать метод, получающий доступ к массиву десять раз, чтобы получить более точный результат. В основном, однако, при написании таймеров цель программистов — написать простые операции, а затем позволить инфраструктуре повторять эти операции столько раз, сколько потребуется для получения точных результатов.

Реализация

Обратите внимание на то, что таймер печатает шесть значений для каждого метода, который он замеряет. Это происходит потому, что таймер используется для сравнения операций на различных размерах данных. Метод `report()` находится во вложенном цикле, где внешний цикл итерирует по исследуемым методам, а внутренний проходит по размерам данных: 1, 10, 100... 100000.

```
private static final int MAXIMUM_SIZE= 100000;
  public void report() throws Exception {
    for (Method each : methods) {
      System.out.print(each.getName() + "\t");
        for (int size= 1; size <= MAXIMUM_SIZE; size*= 10) {
          MethodTimer r= new MethodTimer(size, each);
          r.run();
          System.out.print(String.format("%.2f\t",
r.getMethodTime()));
        }
      System.out.println();
    }
}
```

Отчет прост, насколько это возможно, с разделяющими знаками табуляции между элементами данных, так что данные можно легко вставить в электронную таблицу. В полнофункциональном таймере, возможно, следовало бы сначала вычислить все `MethodTimers`, а затем распечатать их в отдельном проходе, так что отчет можно сделать более гибким.

`MethodTimer`

`MethodsTimer` опирается на вспомогательный объект `MethodTimer` — код, который вычисляет время, необходимое для выполнения одного метода. Метод будет вызван

столько раз, сколько необходимо для накопления данных для аккуратного замера. Затем полное время, затраченное методом, делится на количество вызовов.

Каждый `MethodTimer` воспринимает метод и размер. Поскольку метод может быть воспроизведен много раз, каждый `MethodTimer` кэширует объект, которому может быть послано сообщение при вызове метода. Создавать новый экземпляр для каждого вызова может оказаться слишком накладно. Если операция занимает 50 наносекунд, то ее нужно вызвать 20 миллионов раз, чтобы накопить одну секунду статистики. Создание списка из ста тысяч элементов, вроде такого, который создается показным выше `ListSearch`, занимает примерно 50 миллисекунд на моей машине, так что выполнение только одного этого теста может занять полторы недели. Кэшируя экземпляр, тест занимает только немногим больше секунды. Это повторное использование экземпляра отличается от подхода, использованного в JUnit. В JUnit для каждого выполняемого теста создается свежий экземпляр объекта. Тесты вольны вносить в объект любые изменения, зная, что они изолированы от последующих тестов. Наши тесты замера времени не дают такой свободы. Каждый измеряемый метод должен оставить состояние тестируемого экземпляра именно таким, каким он был вначале.

Ниже показан конструктор `MethodTimer`.

```
private final int size;
private final Method method;
private Object instance;

MethodTimer(int size, Method method) throws Exception {
   this.size= size;
   this.method= method;
   instance= createInstance();
}
```

Каждый метод имеет класс, и это тот класс, экземпляр которого создается при замерах времени операций. Это предполагает, что текущая реализация не поддерживает измерение времени унаследованных методов из-за вызова `getDeclaredMethods()` в `MethodsTimer`. Вызов `GetDeclaredMethods()` возвращает только методы, объявленные в самом классе, а не в суперклассе. Более мощный дизайн может заключаться в аннотировании методов, которые будут затем исследоваться, с прохождением по всей цепочке суперклассов в поисках таких методов. Иерархии могут удалить некоторые из повторений, которые вы увидите в таймерах, показанных в этой книге. (Специфические таймеры, использованные далее в этом приложении.) Еще раз повторим: эта инфраструктура создана достаточно хорошей только для целей этой книги. Инфраструктура, используемая множеством людей, требует другой логики при разработке. Для широко используемой инфраструктуры требуется более высокая инвестиция, хотя любое улучшение, сделанное в пользу более мощного (и дорогого) варианта, окупится сторицей.

Создание используемого экземпляра включает поиск конструктора, который воспринимает целое как параметр и вызывает его.

```
private Object createInstance() throws Exception {
   Constructor<?> constructor= method.getDeclaringClass().
getConstructor(new
           Class[]{int.class});
   return constructor.newInstance(new Object[]{size});
}
```

На самом деле существуют три фактора для вычисления времени, необходимого для одного вызова метода: число итераций, общее время для вызова этого метода такое количество раз и накладные расходы для косвенного вызова метода с помощью механизма reflection. Поскольку вызов метода через reflection может оказаться дорогим по сравнению с временем, которое требуется на само выполнение метода (около 150 наносекунд на моей машине), то удаление этих накладных расходов улучшит точность таймера. Каждый из этих факторов вычисляется методом `run()` и сохраняется в переменных.

```
private long totalTime;
private int iterations;
private long overhead;

double getMethodTime() {
   return (double) (totalTime - overhead) / (double) iterations;
}
```

Метод `run()` является сердцем таймера. Он вызывает измеряемый метод один раз, затем два раза, затем четыре, и так далее, пока не будет израсходована одна секунда. Затем вычисляются накладные расходы, поскольку это необходимо для множества динамических вызовов. После завершения метода `run()` `MethodTimer` готов к опросу результатов. Временн*ая* зависимость между `run()` и любым методом опроса (вроде показанного `getMethodTime()`) слегка раздражает. Альтернатива в том, чтобы конструктор сам производил замеры времени выполнения. Не очень-то хочется возлагать на конструктор какую-то значительную работу, поскольку мне нравится оставлять себе свободу разделять создание экземпляра и выполнение работы. Если все сделано таким образом, то я могу, при желании, создать коллекцию объектов `MethodTimers` и пустить их в плавание: сериализировать или переслать их по сети, не очень заботясь о производительности.

```
void run() throws Exception {
  iterations= 1;
  while (true) {
    totalTime= computeTotalTime();
    if (totalTime > MethodsTimer.ONE_SECOND)
      break;
    iterations*= 2;
  }
  overhead= overheadTimer(iterations).computeTotalTime();
}
```

Обратите внимание на использование другой константы, ONE_SECOND, в MethodsTimer. Использование констант для конфигурации является недорогим способом обеспечить для пользователей кое-какую гибкость, которые хотят немного подправить исходный код, но такие константы лучше собрать в одном месте, так, чтобы их было легко найти.

```
static final int ONE_SECOND= 1000000000;
```

Исключая накладные расходы

Все, что осталось по части инфраструктуры, — это методы вычисления накладных расходов динамического вызова методов. Статическая фабрика, метод overhead-Timer(), предназначена для логической связи метода с назначением этого специфического таймера. Это немного странно, заставлять один вызов MethodTimer вызывать другой как часть своей работы, но после нескольких экспериментов это оказалось лучшим способом, с помощью которого мне удалось организовать код.

```
private static MethodTimer overheadTimer(int iterations) throws
Exception {
  return new MethodTimer(iterations);
}
private MethodTimer(int iterations) throws Exception {
   this(0, MethodTimer.Overhead.class.getMethod("nothing", new
Class[0]));
   this.iterations= iterations;
}
public static class Overhead {
  public Overhead(int size) {
  }
  public void nothing() {
  }
}
```

Тесты

Далее приводятся тесты, использованные для генерации данных, представленных в главе “Коллекции”. Они демонстрируют использование инфраструктуры таймеров, некоторые особенности классов коллекций и в то же время некоторые ограничения представленной здесь архитектуры.

Сравнение коллекций

Первый пример сравнивает коллекции Set и ArrayList. Конструктор создает две коллекции заданного размера и инициализирует их. В качестве данных используются строки. Хеш-значения строк не будут случайно распределенными. Однако, когда

я пробовал использовать целые в качестве данных, я обнаружил еще более странное поведение, поскольку хеш-значения для больших наборов редко распределяются случайно. Если для вас важна крупномасштабная производительность, то лучше создать "типичные" данные, представляющие используемые вами в вычислениях элементы, и произвести замеры таймингов для собственных целей.

Каждый набор временны*х* тестов представлен классом. В свою очередь тайминги представлены методами этого класса. Класс хранит коллекции, с которыми будут производиться вычисления. Заметьте, что обе определены как `Collection`. Класс также хранит образец, элемент, который впоследствии будет использован для поиска.

```
public class SetVsArrayList {
  private Collection<String> set;
  private Collection<String> arrayList;
  private String probe;
}
```

Для инициализации образца коллекции заполняются элементами. Заметьте, что образец находится в середине коллекции, так что разыгрывается сценарий худшего случая. Более глубокое исследование производительности коллекций может задействовать несколько элементов из начала, середины и конца коллекции.

```
public SetVsArrayList(int size) {
  set= new HashSet<String>(size);
  arrayList= new ArrayList<String>(size);
  for (int i= 0; i < size; i++) {
    String element= String.format("a%d", i);
    set.add(element);
    arrayList.add(element);
  }
  probe= String.format("a%d", size / 2);
}
```

Первая пара операторов сравнивает время, требуемое для тестирования членства элемента в наборе. Единственное отличие между двумя методами — в тестируемой коллекции. Требуемое время для членства в `HashSet` почти постоянно, тогда как время, нужное для определения членства в `ArrayList`, линейно растет вместе с размером коллекции.

```
public void setMembership() {
  set.contains(probe);
}
public void arrayListMembership() {
  arrayList.contains(probe);
}
```

Повторения в этих методах намекают на альтернативный API с меньшими повторениями, где тестируемый класс будет содержать абстрактную реализацию исследуемых методов. Каждый экземпляр будет инициализирован конкретным классом для тестирования.

```
public class CollectionOperations {
  Collection<String> collection;
  String probe;

  public void membership() {
    collection.contains(probe);
  }
}
```

Конкретный класс коллекции может быть инициализирован или через конструктор, или в субклассе. В то время как этот подход приводит к меньшим повторениям и может быть прекрасным для широко распространяемой инфраструктуры, данная реализация достаточно хороша для этой книги, так что я оставляю эти повторения.

Другая пара методов в `SetVsArrayList` измеряет время, требуемое для итерации по всей коллекции. Это требуемое время линейно зависит от числа элементов.

```
public void setIteration() {
  Iterator<String> all= set.iterator();
  while (all.hasNext())
    all.next();
}
public void arrayListIteration() {
  Iterator<String> all= arrayList.iterator();
  while (all.hasNext())
    all.next();
}
```

Я пытался профилировать итерацию в цикле `for (String each : set);`, но реализация Java была достаточно умна, чтобы заметить пустое тело цикла и полностью исключить цикл. В основном главным вызовом при написании методов-таймеров остается создание максимально простого кода, но не настолько, чтобы оптимизация Java удалила метод полностью. Всегда проверяйте ваши результаты, чтобы быть уверенным, что они имеют смысл. В противном случае пытайтесь использовать другие пути для тех же вычислений. Заключительные методы таймингов поверяют время, требуемое для внесения изменений в коллекцию. Методы достаточно аккуратны, чтобы в конце концов оставить коллекции в первоначальном виде. Это ограничение инфраструктуры для измерения времени: поскольку каждый метод будет вызываться множество раз с тем же самым объектом, методы должны оставлять состояние объекта неизменным.

```
public void setModification() {
  set.add("b");
  set.remove("b");
}
public void arrayListModification() {
  arrayList.add("b");
  arrayList.remove("b");
}
```

Тайминги этих операций такие же, как и тайминги при определении членства: почти константа для HashSet и линейная зависимость для ArrayList.

Сравнивая ArrayList и LinkedList

Этот тест похож на предыдущий, за исключением того, что исследуемые коллекции были потомками List, а не Collection, и инициализированы как ArrayList и LinkedList.

```
public class Lists {
  private List<String> arrayList;
  private List<String> linkedList;
  private final int size;
}
```

Скорее чем хранить элемент-образец для поиска, как мы делали при поиске в коллекциях, этот тест помнит размер используемой коллекции для последующего использования в методе get(int) класса List.

```
public Lists(int size) {
  this.size= size;
  arrayList= new ArrayList<String>(size);
  linkedList= new LinkedList<String>();
  for (int i= 0; i < size; i++) {
    String element= String.format("a%d", i);
    arrayList.add(element);
    linkedList.add(element);
  }
}
```

Первая пара тестов замеряет время, необходимое для модификации коллекции, вставляя, а затем удаляя элемент. Заметьте, что элемент вставляется в начало коллекции. ArrayList оптимизирует вставку в конец, так что это занимает время в виде простой линейной зависимости, в отличие от LinkedList.

```
public void arrayListModification() {
  arrayList.add(0, "b");
  arrayList.remove(0);
}
public void linkedListModification() {
  linkedList.add(0, "b");
  linkedList.remove(0);
}
```

Другой тест измеряет время, требуемое для доступа к элементу. Результаты являются зеркальным отображением теста на модификацию: время доступа к ArrayList является постоянным, а доступ к LinkedList изменяется линейно. Моя первая версия этого теста получала доступ к последнему элементу коллекции, но оказалось, что

LinkedList оптимизирует этот случай, проводя поиск с конца для индексов, больших половины его размера.

```
public void arrayListAccess() {
  arrayList.get(size / 2);
}
public void linkedListAccess() {
  linkedList.get(size / 2);
}
```

Сравнивая множества Set

Сравнение наборов Set следовало тому же образцу, который вообще используется в этом списке сравнений. Сравнивались две операции: модификация и проверка членства элемента, поскольку все остальные либо построены на основе этих операций, либо показывают приблизительно такие же профили производительности. В коде, приведенном ниже, показан только один вариант каждого из методов тайминга, поскольку другие варианты идентичны, исключая набор, которому посылались сообщения.

Три реализации, которые сравнивались, — это HashSet, LinkedHashSet и TreeSet. Говоря точно, TreeSet является реализацией SortedSet, но я подумал, что может оказаться полезным сравнить накладные расходы, которые связаны с содержанием элементов в отсортированном порядке.

```
public class Sets {
  private Set<String> hashSet;
  private Set<String> linkedHashSet;
  private Set<String> treeSet;
  private String probe;
}
```

Конструктор инициализирует каждый из этих наборов идентичными элементами и инициализирует образец, который потом используется в проверках членства элемента. Заметьте, что каждый набор создан с запасом для хранения нужного числа элементов. Во всех книгах по Java, которые я читал, упор сделан на важности предварительного распределения памяти для коллекций. Однако мои замеры показали, что распределение нуля элементов находится в 10% от распределения окончательного числа элементов.

```
public Sets(int size) {
  hashSet= new HashSet<String>(size);
  linkedHashSet= new LinkedHashSet<String>(size);
  treeSet= new TreeSet<String>();
  for (int i= 0; i < size; i++) {
    String element= String.format("a%d", i);
    hashSet.add(element);
    linkedHashSet.add(element);
    treeSet.add(element);
```

```
  }
  probe= String.format("a%d", size / 2);
}
```

Для проверки членства поиск проводился по предварительно вычисленному образцу. Другие реализации наборов замерялись похожим образом.

```
public void hashSetContains() {
  hashSet.contains(probe);
}
```

Тест таймингов модификации добавляет и удаляет тот же самый элемент, оставляя набор неизменным.

```
public void hashSetModification() {
  hashSet.add("b");
  hashSet.remove("b");
}
```

Сравнение коллекций `Maps`

Сравнение хеш-таблиц дает почти такие же результаты, как и сравнение наборов. Снова у нас есть три реализации, доступные в библиотеке Java, — `HashMap`, `LinkedHashMap` и `TreeMap`. Тезультаты тайминга такие же, поскольку наборы используют хеш-таблицы в своей работе. Так, `HashSet` использует `HashMap` внутренне для хранения элементов, и т.д.

```
public class Maps {
  private Map<String, String> hashMap;
  private Map<String, String> linkedHashMap;
  private Map<String, String> treeMap;
  private String probe;
```

Инициализация хеш-таблиц требует посылки им `put()` вместо `add()`. Я сделал ключ и значение одинаковыми, поскольку используемые значения не имеют значения для замеров времени.

```
public Maps(int size) {
  hashMap= new HashMap<String, String>(size);
  linkedHashMap= new LinkedHashMap<String, String>(size);
  treeMap= new TreeMap<String, String>();
  for (int i= 0; i < size; i++) {
    String element= String.format("a%d", i);
    hashMap.put(element, element);
    linkedHashMap.put(element, element);
    treeMap.put(element, element);
  }
  probe= String.format("a%d", size / 2);
}
```

Два метода тайминга используют методы хеш-таблиц `containsKey()` и `put()` вместо операций для множеств `contains()` и `add()`. Во всем остальном они идентичны.

```
public void hashMapContains() {
  hashMap.containsKey(probe);
}
public void hashMapModification() {
  hashMap.put("b", "b");
  hashMap.remove("b");
}
```

Выводы

Рассмотренная инфраструктура и приведенные выше примеры преподносят уроки на нескольких уровнях. Первый — это цена получения доступа к данным. Широко распространенные убеждения, такие как “заранее зарезервированные множества улучшают производительность”, требуют внимательного исследования. Перед тем как добавлять сложность в программу, убедитесь, что эта сложность будет иметь свои выгоды. Иногда единственная возможность оценить преимущества — это самому измерить их. Второй урок построения методов, измеряющих время, состоит в изменении стиля программирования в зависимости от контекста. Можно построить и закодировать инфраструктуру по-разному, в зависимости от того, будет ли код иметь более широкую аудиторию, или нужно сделать больше замеров времени. В данном случае я принял упрощающие предположения, так что написание оболочки и самих тестов значительно упростилось. Догматические советы, вроде “всегда вкладывайте всю гибкость, на которую только способны” или “кодируйте для сегодня, забудьте о завтра”, одинаково бестолковы.

Наконец, код в этом разделе использует версии многих шаблонов реализации из остальной части книги. Полные конструкторы, имена, объясняющие назначение переменных, и т.д., представлены в каждой строке этого кода. Помогали ли они объяснить мои намерения или нет — об этом можете судить только вы. Если нет, то определите для себя те шаблоны, которые помогут вам лучше передавать свои мысли. В конце концов, это самый главный урок этой книги: работа программиста во взаимодействии с другими программистами, а не с машиной. Программирование является деятельностью людей, и для людей. Не нужно выходить за рамки общества. Это может стать методом взаимного общения. Да, и к тому же это будет хороший код.

Библиография

Здесь приводится список книг, которые я нашел полезными по мере своего обучения программированию.

Общее программирование

- Kent Beck, Smalltalk Best Practice Patterns, Prentice Hall, 1997. ISBN 013476904X.

  Шаблоны реализации для Smalltalk. Многие похожи на приведенные здесь, но есть и значительные отличия, поскольку языки также отличаются. Чтение этой книги заставило меня замедлиться и обдумать решения, которые я раньше делал инстинктивно.

- Martin Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999. ISBN 0201485672.

  Легко догадаться, что любой проект будет меняться со временем. В этой книге показано, как вносить эти изменения.

- Eric Freeman and Elisabeth Freeman, Head First Design Patterns, O'Reilly Media, 2004. ISBN 0596007124.

  Альтернативное визуально-ориентированное введение в шаблоны реализации.

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995. ISBN 0201633612.

  Классическое описание крупномасштабных повторяющихся структур в коде.

- Daniel Hoffman and David Weiss, Software Fundamentals: Collected Papers by David L. Parnas, Addison-Wesley, 2001. ISBN 0201703696.

  Описывает теоретические предпосылки создания хорошего ПО.

- Andrew Hunt and David Thomas, The Pragmatic Programmer, Addison-Wesley, 2000. ISBN 020161622X.

  Качества профессионального программиста ясны из этой книги: любопытство, честность, постоянное изучение нового.

- Brian Kernighan and Rob Pike, The Practice of Programming, Addison-Wesley, 1999. ISBN 020161586X. (Брайан У. Керниган, Роб Пайк, Практика программирования, пер. с англ. ИД "Вильямс", 2005. ISBN 5-8459-0679-2)

  Еще один портрет глубоко профессиональных программистов за работой.

- Donald Knuth, The Art of Computer Programming: Volume 1, Fundamental Algorithms, 3rd Edition, Addison-Wesley, 1997. ISBN 0201896834. (Дональд Э. Кнут, Искусство программирования, том 1. Основные алгоритмы, 3-е издание, пер. с англ. ИД "Вильямс", 2000. ISBN 5-8459-0080-8)

- Donald Knuth, The Art of Computer Programming: Volume 2, Seminumerical Algorithms, 3rd Edition, Addison-Wesley, 1997. ISBN 0201896842. (Дональд Э. Кнут, Искусство программирования, том 2. Получисленные алгоритмы, 3-е издание, пер. с англ. ИД "Вильямс", 2000. ISBN 5-8459-0081-4)

- Donald Knuth, The Art of Computer Programming: Volume 3, Searching and Sorting, 2nd Edition, Addison-Wesley, 1998. ISBN 0201896850.(Дональд Э. Кнут, Искусство программирования, том 3. Сортировка и поиск, 2-е издание, пер. с англ. ИД "Вильямс", 2000. ISBN 5-8459-0082-1)

  Профессор Кнут явно любит программирование и передает эту любовь в своих книгах.

- Donald Knuth, Literate Programming, Center for the Study of Language and Information, 1992. ISBN 0937073806.

  Одна из первых книг, сфокусированных на потребности программистов в коммуникации с другими программистами. Источник одной из моих любимых цитат: "программу следует читать, как книгу". Не всегда стоит заходить так далеко, как "литературное программирование", но направление задано правильно.

- Steve McConnell, Code Complete: A Practical Handbook of Software Construction, 2nd Edition, Microsoft Press, 2004. ISBN 0735619670.

  Рассматривает технологии, необходимые для программной ответственности.

- Diomidis Spinellis, Code Reading, Addison-Wesley, 2003. ISBN 0201799405. (Диомидис Спинеллис, Анализ программного кода на примере проектов Open Source, пер. с англ. ИД "Вильямс", 2004. ISBN 5-8459-0604-0)

  Взгляд с другой стороны: как читать код. Чтение кода является обратной стороной шаблонов реализации, чтение для понимания против написания для понимания.

- Edward Yourdon, Techniques of Program Structure and Design, Prentice Hall, 1975. ISBN 013901702X.

  Одно из ранних объяснений понятия хорошей программы. Принципы по-прежнему звучат разумно, хотя примеры выглядят устаревшими.

- Edward Yourdon and Larry Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice Hall, 1979. ISBN 0138544719.

  Эта книга представляет собой эквивалент законов физики для разработки программ и основ для обсуждения экономических аспектов разработки.

Философия

- Christopher Alexander, Notes on the Synthesis of Form, Harvard University Press, 1964. ISBN 0674627512.

  Объясняет теорию, стоящую за шаблонами: повторяющиеся решения с повторяющимися шаблонами ограничений и похожими решениями.

- Christopher Alexander, The Timeless Way of Building, Oxford University Press, 1979. ISBN 0195024028.

  Теоретическое описание разработки и конструирования с помощью шаблонов. Общая тема в преимуществах разработки понемногу за раз, используя обратную связь от прошлых разработок, конструирования и использования.

- Christopher Alexander, Sara Ishikawa, Murray Silverstein with Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel, A Pattern Language, Oxford University Press, 1977. ISBN 0195019199.

  Пример языка шаблонов широкого использования. Также полезно при построении рабочих мест и домов.

- Richard Gabriel, Patterns of Software, Oxford University Press, 1996. ISBN 019510269X.

  Собрание эссе о применении шаблонного мышления к разработке ПО.

- Robert Grudin, The Grace of Great Things, Ticknor and Fields, 1990. ISBN 0395588685.

  Прославляет и поощряет исключительно хорошее проектирование.

- Leonard Koren, Wabi-Sabi for Artists, Designers, Poets, and Philosophers, Stone Bridge Press, 1994. ISBN 1880656124.

  Эффективный дизайн не в поиске совершенства, но достаточности. Wabi-sabi является японской эстетикой действительной красоты, иногда грубой, но всегда функциональной.

- D'Arcy Thompson, On Growth and Form, Cambridge University Press, 1961. ISBN 0521437768.

  Часто сложно воспринимаемая книга о сложности, созданной и выраженной в дикой природе.

- Edward Tufte, The Visual Display of Quantitative Information, Graphics Press, 1983. ISBN 0961392142.

  Привлекательный пример основанного на принципах мышления, на этот раз в сфере графического дизайна.

Java

- Joshua Bloch, Effective Java Programming Language Guide, Addison-Wesley, 2001. ISBN 0201310058.

  Раннее описание того, как использовать Java, включая добрый кусок неочевидной информации, почему Java такая, какая есть.

- Bruce Eckel, Thinking in Java, 4th Edition, Prentice Hall, 2006. ISBN 0131872486.

  Моя библия Java. Когда мне нужно узнать, как что-то работает в Java, я открываю эту книгу.

- Steven Metsker, Design Patterns Java Workbook, Addison-Wesley, 2002. ISBN 0201743973.

  Показывает, как Java влияет на основные шаблоны разработки.

Предметный указатель

Л

М

Н

О

П

Р

С

Т

У

Ф

Ц

Ш

Научно-популярное издание

Кент Бек

Шаблоны реализации корпоративных приложений

Литературный редактор *И.А. Попова*
Верстка *Т.А. Корзун*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

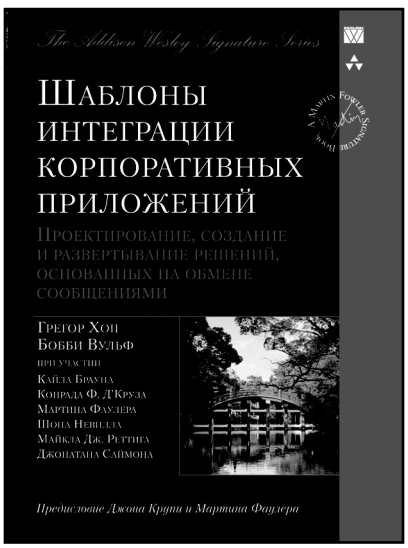
ООО "И.Д. ВИЛЬЯМС"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 31.03.2008. Формат 70×100/16.
Гарнитура Petersburg. Печать офсетная.
Усл. печ. л. 14,19. Уч.-изд. л. 8,55.
Тираж 2000 экз. Заказ № 0000.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

ШАБЛОНЫ ИНТЕГРАЦИИ КОРПОРАТИВНЫХ ПРИЛОЖЕНИЙ

Грегор Хоп,
Бобби Вульф



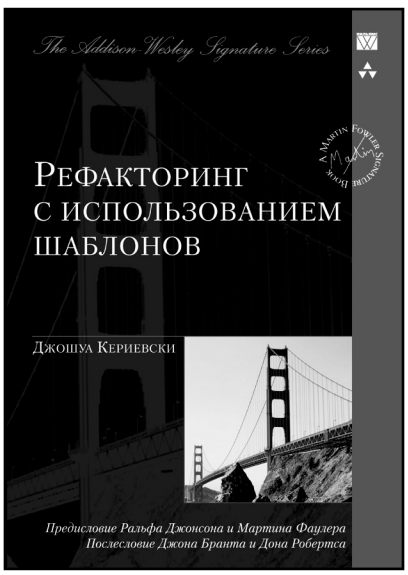
www.williamspublishing.com

В данной книге исследуются стратегии интеграции корпоративных приложений с помощью механизмов обмена сообщениями. Авторы рассматривают шаблоны проектирования и приводят практические примеры интеграции приложений, демонстрирующие преимущества обмена сообщениями и эффективность решений, создаваемых на основе этой технологии. Каждый шаблон сопровождается описанием некоторой задачи проектирования, обсуждением исходных условий и представлением элегантного, сбалансированного решения. Авторы подчеркивают как преимущества, так и недостатки обмена сообщениями, а также дают практические советы по написанию кода подключения приложения к системе обмена сообщениями, маршрутизации сообщений и мониторинга состояния системы.
Книга ориентирована на разработчиков программного обеспечения и системных интеграторов.

ISBN 978-5-8459-0741-9 **в продаже**

РЕФАКТОРИНГ С ИСПОЛЬЗОВАНИЕМ ШАБЛОНОВ

Джошуа Кериевски



www.williamspublishing.com

Данная книга представляет собой результат многолетнего опыта профессионального программиста по применению шаблонов проектирования. Авторский подход к проектированию состоит в том, что следует избегать как недостаточного, так и избыточного проектирования, постоянно анализируя готовый работоспособный код и реорганизуя его только в том случае, когда это приведет к повышению его эффективности, упрощению его понимания и сопровождения. Автор на основании как собственного, так и чужого опыта детально рассматривает различные признаки кода, требующего рефакторинга, описывает, какой именно рефакторинг наилучшим образом подходит для той или иной ситуации, и описывает его механику, подробно разбирая ее на конкретных примерах из реальных задач. Книга может рассматриваться и как учебник по рефакторингу для программиста среднего уровня, и как справочное пособие для профессионала.

ISBN 5-8459-1087-0

в продаже

ПРИМЕНЕНИЕ DDD И ШАБЛОНОВ ПРОЕКТИРОВАНИЯ

ПРОБЛЕМНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЙ С ПРИМЕРАМИ НА C# И .NET

Джимми Нильссон

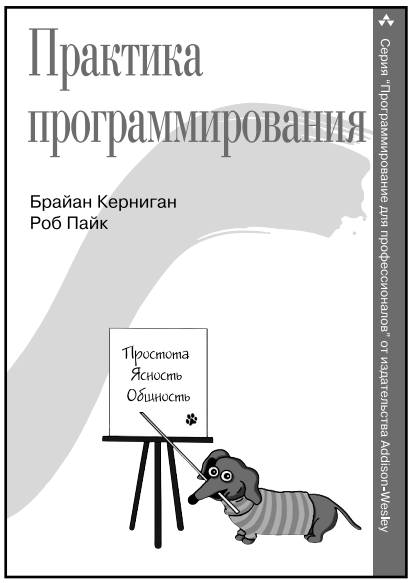
www.williamspublishing.com

Эта книга о разработке корпоративных программных приложений в среде .NET с применением шаблонов проектирования. В ней описаны: проблемно-ориентированные методы проектирования (DDD, или Domain Driven Design), разработка посредством тестирования (TDD, или Test-Driven Development), объектно-реляционное преобразование, т.е. методы, которые многие относят к ключевым технологиям разработки программного обеспечения. Хотя большинство примеров кода представлено на языке C#, материал книги может оказаться полезным и для тех, кто работает на платформе Java. Книга адресована опытным разработчикам архитектуры и прикладного программного обеспечения уровня предприятий, в том числе и в среде .NET.

ISBN 978-5-8459-1296-1 **в продаже**

ПРАКТИКА ПРОГРАММИРОВАНИЯ

Брайан У. Керниган
Роб Пайк



www.williamspublishing.com

Вашему вниманию предлагается перевод на русский язык исправленного и дополненного издания (уже восьмого по счету) популярной книги, вышедшего из печати в январе 2004 года. Верификацию кода в русском издании выполнили сами авторы книги — Брайан Керниган и Роб Пайк, что лишний раз свидетельствует об их огромной ответственности перед читателями.

В книге рассматриваются принципы практического профессионального программирования, которые, выходя за рамки простого написания кода, включают в себя проектирование, правильный выбор алгоритмов и структур данных, отладку и тестирование, оптимизацию быстродействия и переносимости, автоматизацию рабочего процесса. Изложение проиллюстрировано примерами из сложных, практически важных систем.

Книга предназначена для повышения квалификации программистов. Может быть полезна студентам и преподавателям компьютерных специальностей.

ISBN 5-8459-0679-2

в продаже

АНАЛИЗ ПРОГРАММНОГО КОДА НА ПРИМЕРЕ ПРОЕКТОВ OPEN SOURCE

Диомидис Спинеллис

www.williamspublishing.com

ISBN 5-8459-0604-0

в продаже

Книга посвящена важному аспекту программирования, недостаточно освещенному в литературе — чтению и анализу программного кода на языках высокого уровня с целью доработки, извлечения готовых технических решений или изучения новых методов. Даются ценные рекомендации по улучшению стиля программирования.
Изложение проиллюстрировано большим количеством примеров, взятых из больших программных проектов с открытым кодом, находящихся на прилагаемом компакт-диске.
Книга предназначена для повышения квалификации программистов. Может быть полезна студентам и преподавателям соответствующих специальностей, а также начинающим программистам.

ПРИМЕНЕНИЕ UML 2.0 И ШАБЛОНОВ ПРОЕКТИРОВАНИЯ, третье издание

Крэг Ларман

www.williamspublishing.com

В книге вы найдете новые сведения об итеративном и гибком моделировании, шаблонах проектирования GRASP и GoF, прецедентах, архитектурном анализе и многоуровневой архитектуре, а также рефакторинге, разработке на основе обязанностей и многих других вопросах. Весь материал рассматривается в контексте использования унифицированного процесса (UP) как легкого и гибкого подхода к разработке совместно с приемами из других итеративных методов, таких как XP и Scrum.

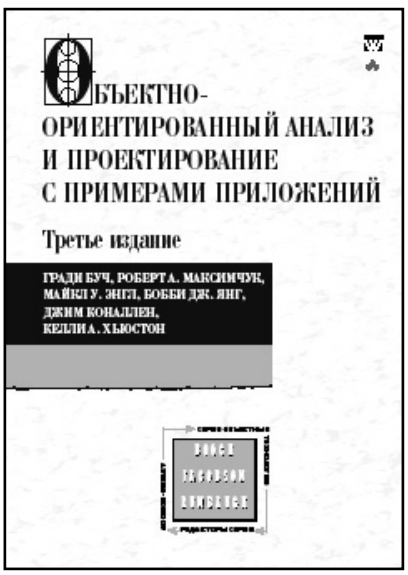
Данная книга будет прекрасным руководством для как для новичков, так и для специалистов, кто интересуется вопросами ООА/П, языком моделирования UML 2 и самыми современными эволюционными подходами к разработке программного обеспечения.

ISBN 978-5-8459-1185-8

в продаже

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ С ПРИМЕРАМИ ПРИЛОЖЕНИЙ
ТРЕТЬЕ ИЗДАНИЕ

Гради Буч,
Роберт А. Максимчук,
Майкл У. Энгл
и др.



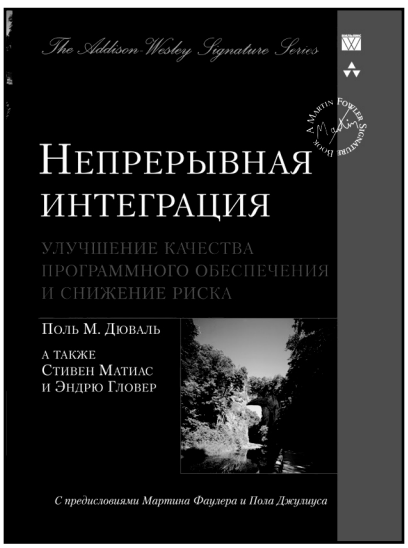
www.williamspublishing.com

Книга представляет собой новое издание бестселлера Гради Буча по объектно-ориентированному анализу и проектированию. Авторы описывают объектные методы решения сложных проблем, связанные с разработкой систем и программного обеспечения. Используя многочисленные примеры, они иллюстрируют основные концепции объектно-ориентированного подхода на примере разработки систем управления, сбора данных и искусственного интеллекта. Читатели найдут в книге практические советы, касающиеся важных вопросов анализа, проектирования, реализации и оптимального управления проектами. Книга будет полезна системным аналитикам и архитекторам, программистам, преподавателям и студентам высших учебных заведений, а также всем специалистам по информационным технологиям.

ISBN 978-5-8459-1401-9 **в продаже**

НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ
УЛУЧШЕНИЕ КАЧЕСТВА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И СНИЖЕНИЕ РИСКА

Поль М. Дюваль,
Стивен Матиас
и Эндрю Гловер



www.williamspublishing.com

В этой книге рассматриваются некоторые из наиболее типичных процессов разработки программного обеспечения: компиляция кода, определение данных и манипулирование ими в базе данных; осуществление проверки, просмотр кода и в конечном итоге развертывание программного обеспечения. Но главное, в ней описано, как непрерывная интеграция способна снизить риски, которые подстерегают при создании приложений.
В системе непрерывной интеграции большинство этих процессов автоматизировано, и они запускаются после каждого изменения разрабатываемого программного обеспечения.
В этой книге обсуждаются аспекты автоматизации непрерывной интеграции, большинство предоставляемых ей преимуществ в области повторяемых и склонных к ошибкам процессов.

ISBN 978-5-8459-1408-8 **в продаже**

ШАБЛОНЫ C++: СПРАВОЧНИК РАЗРАБОТЧИКА

Дэвид Вандевурд, Николай М. Джосаттис

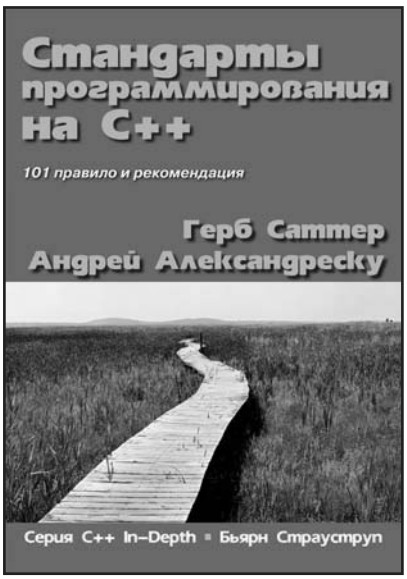
www.williamspublishing.com

в продаже

Несмотря на то, что шаблоны вошли в язык программирования C++ более десяти лет назад, они представляют собой активно развивающуюся часть C++, предоставляющую программисту новые возможности быстрой разработки эффективных и надежных программ и повторного использования кода. Будучи одной из наиболее мощных возможностей языка, шаблоны одновременно являются одной из самых запутанных, трудно понимаемых и зачастую неверно используемых частей C++. Данная книга, написанная в соавторстве теоретиком C++ и программистом-практиком с большим опытом, удачно сочетает строгость изложения и полноту освещения темы с вопросами практического использования шаблонов. В книге содержится масса разнообразного материала, относящегося к программированию с использованием шаблонов, включая такие вопросы, как новые идиомы программирования, связанные с шаблонами, метапрограммирование или возможные направления развития шаблонов в будущем. Кроме того, книга снабдит читателя материалом, который даст опытным программистам возможность преодолеть современные ограничения в этой области Книга предполагает наличие у читателя достаточно глубоких знаний языка C++, поскольку в книге дается детальное описание конкретного средства языка программирования, но не основ самого языка. Тем не менее стиль изложения обеспечивает доступность материала как для квалифицированных специалистов, так и для программистов среднего уровня.

СТАНДАРТЫ ПРОГРАММИРОВАНИЕ НА C++

Герб Саттер
Андрей Александреску



www.williamspublishing.com

Эта книга поможет новичку стать профессионалом, так как в ней представлен сконцентрированный лучший опыт программистов на C++, обобщенный двумя экспертами мирового класса. Если вы — начинающий программист, эта книга для вас: вы сможете найти в ней простые и понятные рекомендации для ежедневного использования, подкрепленные примерами их конкретного применения на практике. Если вы — программист с опытом, эта книга для вас: в ней вы найдете как советы, до которых вы дошли самостоятельно, так и новые рекомендации, которые вы сможете по достоинству оценить и принять на вооружение. Если вы — программист-профессионал, эта книга для вас: вы сможете использовать ее как основу для разработки собственных стандартов кодирования для себя лично или для группы, которой вы руководите. Конечно, книга рассчитана в первую очередь на профессиональных программистов с глубокими знаниями языка, однако она будет полезна любому, кто захочет углубить свои знания в данной области.

ISBN 978-5-8459-0859-9 **в продаже**