O'REILLY®

/ Кевлин Хенни
Триша Джи /

97 ВЕЩЕЙ, О КОТОРЫХ ДОЛЖЕН ЗНАТЬ КАЖДЫЙ JAVA-ПРОГРАММИСТ

{СОВЕТЫ ЛУЧШИХ ЭКСПЕРТОВ}

- новые функции
- уязвимости Java-библиотеки
- объектно-ориентированные принципы в коде

O'REILLY®

/ Kevlin Henney
Trisha Gee /

97 THINGS EVERY JAVA PROGRAMMER SHOULD KNOW:

{COLLECTIVE WISDOM FROM THE EXPERTS}

O'REILLY®

/ Кевлин Хенни
Триша Джи /

97 ВЕЩЕЙ, О КОТОРЫХ ДОЛЖЕН ЗНАТЬ КАЖДЫЙ JAVA-ПРОГРАММИСТ

{СОВЕТЫ ЛУЧШИХ ЭКСПЕРТОВ}

- новые функции
- уязвимости Java-библиотеки
- объектно-ориентированные принципы в коде

Москва 2023

УДК 004.43
ББК 32.973.26-018.1
Х38

97 Things Every Java Programmer Should Know
Kevlin Henney, Trisha Gee

Хенни, Кевлин.

Х38 97 вещей, о которых должен знать каждый Java-программист : советы лучших экспертов / Кевлин Хенни, Триша Джи ; [перевод с английского М. А. Райтмана]. — Москва : Эксмо, 2023. — 288 с. — (Мировой компьютерный бестселлер).

ISBN 978-5-04-169254-4

Что должен знать каждый Java-программист? Ответов на этот вопрос может быть очень много. Авторы этой книги собрали мнения нескольких десятков опытных разработчиков на Java, чтобы создать единое руководство для тех, кто только начинает свой путь в программировании.

Внутри вы найдете подробные инструкции по основным темам, касающимся работы с Java. И еще — советы от экспертов и их истории профессионального развития в разработке!

УДК 004.43
ББК 32.973.26-018.1

ISBN 978-5-04-169254-4

Содержание

В память о тех, кто с мудростью
и состраданием выпестовал нас

Предисловие

Ум — это не сосуд, который нужно наполнить, а факел, который нужно зажечь.

Плутарх

Что должен знать каждый Java-программист? Зависит от обстоятельств. От того, зачем, кого и когда вы спрашиваете. Ответов по крайней мере столько же, сколько точек зрения. В языке, платформе, экосистеме и сообществе, которые влияют на программное обеспечение (ПО) и жизни многих людей и делали так из двадцатого века в двадцать первый, от одного ядра ко многим, от мегабайт к гигабайтам, ответ зависит от большего количества факторов, чем возможно охватить в одной книге единственным автором.

Вместо этого в данной книге мы опираемся на некоторые из этих многочисленных точек зрения, чтобы дать вам поперечный срез и представление о типах мышления во вселенной Java. Это не *все* мнения, но 97 из них от 73 участников. Процитируем предисловие «97 вещей, которые должен знать каждый программист» (O'Reilly):

> Когда так много нужно знать, так много нужно сделать и есть так много способов сделать это, ни один человек или один источник не может утверждать, что его путь — «единственный истинный». Идеи не совпадают, как модульные части, и никто к этому не стремится — пожалуй, даже наоборот. Ценность каждого вклада проистекает из его уникальности. Ценность коллекции заключается в том, как материалы дополняют, подтверждают и даже противоречат друг другу. Здесь нет всеобъемлющего повествования: вы должны прочувствовать идеи, поразмыслить над ними и связать воедино, примеряя их к своей ситуации, знаниям и опыту.

Что должен знать каждый Java-программист? 97 тем, которые мы выбрали в качестве ответа, охватывают язык, JVM, методы тестирования, JDK, сообщество, историю, гибкое мышление, ноу-хау реализации, профессионализм, стиль, суть, парадигмы программирования, программистов как людей, архитектуру ПО, навыки за пределами программирования, инструментарий, механику GC, языки JVM, отличные от Java… и многое другое.

Разрешения

В духе первой книги из серии «97 вещей...» каждый интеллектуальный вклад в данный том внесен на основе неограничивающей модели с открытым исходным кодом. Каждый вклад автора лицензируется в соответствии с Creative Commons Attribution 4.0 license (*https://oreil.ly/zPsKK*). Многие материалы также впервые появились в публикации «97 вещей...» на странице Medium (*https://medium.com/97-things*).

Все это — топливо и огонь для ваших мыслей и кода.

Онлайн-обучение O'Reilly

Более сорока лет *O'Reilly Media* помогает компаниям добиваться успеха, предоставляя им технологии и бизнес-тренинги, сведения и наработки.

Наше уникальное сообщество экспертов и новаторов делится знаниями и опытом с помощью книг, статей, конференций и нашей платформы онлайн-обучения O'Reilly. Последняя предоставляет вам доступ по запросу к практическим подготовительным курсам, методам углубленного обучения, интерактивным средам программирования и обширной коллекции текстов и видео от O'Reilly и более 200 других издателей. Чтобы узнать подробности, посетите *http://oreilly.com*.

Как с нами связаться

Вы можете получить доступ к веб-странице этой книги, где найдете примеры, обнаруженные неточности и прочую дополнительную информацию, по адресу *https://oreil.ly/97Tejpsk*.

Чтобы прокомментировать или задать технические вопросы об этой книге, напишите письмо на *bookquestions@oreilly.com*.

Для получения новостей и информации о наших книгах и курсах посетите *http://oreilly.com*.

Присоединяйтесь к нам в Twitter *http://twitter.com/oreillymedia*, а также посмотрите *http://twitter.com/97_Things*.

Смотрите нас на YouTube: *http://youtube.com/oreillymedia*.

Благодарности

Многие люди вложили свои знания и время, как прямо, так и косвенно, в проект «97 вещей, которые должен знать каждый Java-программист». Все они заслуживают похвалы.

Мы хотели бы поблагодарить всех тех, кто потратил время и силы, чтобы внести свой вклад в эту книгу. Мы также благодарны Брайану Гетцу за дополнительные отзывы, комментарии и предложения.

Спасибо O'Reilly за поддержку, которую они оказали этому проекту, в том числе Зан Маккуэйд и Корбину Коллинзу за их руководство и заботу об участниках и содержании, а также Рэйчел Румелиотис, Сьюзан Конант и Майку Лукидесу за их вклад в проделанную работу.

Кроме того, Кевлин хотел бы поблагодарить свою жену Каролин за то, что она понимает смысл его бреда, и своих сыновей Стефана и Янника за то, что они понимают своих родителей.

Триша желала бы добавить благодарность своему мужу Исре (за то, что он подсказал ей: пребывание в стрессе из-за того, что она делает слишком мало, мешает ей делать вообще что-либо) и своим дочерям Эви и Эми за бескорыстную любовь и обнимашки.

Мы надеемся, что эта книга станет для вас информативной, поучительной и вдохновляющей. Наслаждайтесь!

Всё, что вам нужно, — это Java*

Андерс Норас

Работая над первой крупной редакцией Visual Studio, команда Microsoft представила миру трех персонажей-разработчиков: Морта, Элвиса и Эйнштейна.

Морт, приспосабливающийся работник, быстро исправлял и придумывал что-то по ходу дела. Элвис, прагматичный сотрудник, создавал решения на века, обучаясь на рабочем месте. Эйнштейн, программист-параноик, как одержимый разрабатывал наиболее эффективное решение и старался во всем разобраться, прежде чем писать код.

Языки программирования (ЯП) разделены, как религиозные касты. Мы со стороны Java смеялись над Мортами и хотели быть Эйнштейнами, которые создают фреймворки, чтобы вынудить Элвисов писать свой код «правильно».

Это происходило на заре эры фреймворков, и, если вы не владели новейшим, лучшим объектно-реляционным мэппером и фреймворком с инверсией управления, вы не были настоящим Java-программистом. Библиотеки превратились в фреймворки с установленной архитектурой. И по мере того, как они становились технологическими экосистемами, многие из нас забыли о маленьком, но очень способном языке — Java.

Java — отличный ЯП, и в его библиотеке классов есть инструменты на все случаи жизни. Нужно работать с файлами? `java.nio` тебя прикроет. Базы данных? `java.sql` — вам сюда. Почти каждый дистрибутив Java даже оснащен полноценным HTTP-сервером, однако вам придется перейти от модуля с именем Java на `com.sun.net.httpserver`.

По мере того как наши приложения переходят к бессерверным архитектурам, где единицами развертывания могут быть отдельные функции, преимущества, которые мы получаем от фреймворков, уменьшаются. Это связано с тем,

* В оригинале All You Need Is Java — отсылка к All You Need Is Love, классическому хиту Beatles 1967 года. — *Прим. ред.*

что мы, вероятно, будем тратить не так много времени на решение технических и инфраструктурных проблем, сосредоточив усилия по программированию на бизнес-возможностях, которые реализуют наши приложения.

Как выразился Брюс Джойс, автор книг про образование и педагогику:

> *Нам приходится время от времени изобретать колесо, и не потому, что нам нужно много колес, а потому, что нам нужно много изобретателей.*

Многие задались целью создания универсальных фреймворков для бизнес-логики, чтобы максимально использовать их повторно. Большинство потерпели неудачу, поскольку на самом деле нет никаких универсальных бизнес-проблем. Чем один бизнес отличается от другого? Тем, что делает нечто особенное уникальным, специфичным для него образом. Вот почему вы гарантированно будете писать бизнес-логику практически для каждого проекта. Возможно, у вас возникнет соблазн ввести механизм правил или что-то подобное ради того, чтобы создать нечто универсальное и многоразовое. Но в конечном счете настройка механизма правил — это программирование, часто на языке, уступающем Java. Почему бы вместо этого не попробовать просто написать на Java? Вы с удивлением обнаружите, что конечный результат легко читаем, а это, в свою очередь, упрощает поддержку кода даже для тех программистов, которые на Java не работают.

Довольно часто можно заметить, что библиотека классов Java немного ограниченна, и в таких случаях требуется нечто, способное сделать чуть более комфортной работу с датами, сетевое взаимодействие или что-то еще. Это прекрасно. Задействуйте любую библиотеку. Разница в том, что теперь вы будете применять ее, потому что возникла конкретная необходимость, а не потому, что она входила в стек, который вы всегда использовали.

В следующий раз, когда вам в голову придет идея небольшой программы, выведите свои знания о библиотеке классов Java из спящего режима, вместо того чтобы лезть в дебри JHipster. Хипстеризм — это уже не комильфо; жить простой жизнью — вот что актуально сейчас. Бьюсь об заклад, Морту нравилась простая жизнь.

Тестирование на одобрение

Эмили Бач

Вы когда-нибудь писали тестовое утверждение (assertion) с фиктивным или пустым ожиданием? Что-то вроде этого...

```
assertEquals("", functionCall())
```

...где `functionCall()` возвращает строку, и вы не уверены точно, какой она должна быть, но поймете, что она верная, когда увидите ее? Когда вы запускаете тест в первый раз, конечно, он завершается неудачей, потому что `functionCall()` возвращает непустую строку. (Возможно, вам потребуется несколько попыток, пока возвращаемое значение не будет выглядеть правильным.) Затем вы вставляете это значение вместо пустой строки в `assertEquals`. Теперь тест должен пройти. Есть результат! Все это я бы назвала *тестированием на одобрение*.

Ключевой шаг здесь — это когда вы решаете, что выходные данные верны, и используете их в качестве ожидаемого значения. Вы «одобряете» результат — он достаточно хорош, чтобы его сохранить. Я предполагаю, что вы делали нечто подобное, даже не задумываясь по-настоящему. Возможно, вы называете это по-другому: есть варианты «snapshot-тестирование» или «тестирование характеристик» (*golden master testing*). По моему опыту, если у вас есть фреймворк, специально разработанный для поддержки такого типа тестирования, то многое становится на свои места и процесс, таким образом, становится проще.

С классическим фреймворком модульного тестирования, таким как JUnit, обновлять эти ожидаемые строки при их изменении порой бывает немного мучительно. В итоге вы вставляете что-то в исходный код. С помощью инструмента тестирования на одобрение утвержденная строка вместо этого сохраняется в файле. Это сразу же открывает новые возможности. Вы можете использовать инструмент, показывающий различия в двух файлах (diff tool), чтобы просмотреть изменения и объединить их одно за другим. Или получить подсветку синтаксиса для строк JSON и тому подобного. Или искать и заменять обновления в нескольких тестах в разных классах.

Итак, в каких ситуациях тестирование на одобрение будет уместно? Вот несколько примеров.

Код без модульных тестов, который вам нужно изменить

Если код находится в процессе разработки, то все, что он делает, по умолчанию считается правильным и может быть одобрено. Трудная часть создания тестов преобразуется в проблему поиска стыков и вырезания фрагментов логики, которые возвращают нечто интересное, что вы можете одобрить.

REST API и функции, возвращающие JSON или XML

Если результатом является более длинная строка, то возможность сохранить ее вне исходного кода — серьезное преимущество. JSON и XML можно отформатировать с согласованным пробелом, поэтому их легко сравнить с ожидаемым значением. Если в JSON или XML есть значения, которые сильно различаются — например, даты и время, — вам может потребоваться проверить их отдельно, прежде чем заменять их фиксированной строкой и одобрять оставшуюся часть.

Бизнес-логика, которая создает сложный возвращаемый объект

Прежде всего напишите класс `Printer`, который способен принимать ваш сложный возвращаемый объект и форматировать его как строку. Подумайте о классах `Receipt(чек)`, или `Prescription(рецепт)`, или `Order(заказ)`. Любой из них можно успешно представить в виде понятного человеку многострочного текста. Класс `Printer` может выбрать вывод только краткого резюме и будет просматривать схему объектов, чтобы извлечь соответствующие сведения. Затем ваши тесты будут применять различные бизнес-правила и использовать `Printer` для создания одобряемой строки. Тогда, например, ваш менеджер продукта или бизнес-аналитик, незнакомые с программированием, все равно сумеют прочитать результаты тестирования и убедиться в их правильности.

Если у вас уже есть тесты для проверки многострочных переменных, то я рекомендую узнать больше о тестировании на одобрение и начать использовать инструмент, который его поддерживает.

Усильте Javadoc AsciiDoc'ом

Джеймс Эллиотт

Разработчики Java уже знакомы с Javadoc. Опытные программисты помнят, как радикально все изменилось, когда Java стал первым широко распространенным языком, интегрировавшим генератор документации прямо в компилятор и стандартный набор инструментов. Вслед за этим резко расширились объемы API-документации (пусть даже не всегда отличной или «причесанной»), что принесло огромную пользу всем нам, и эта тенденция распространилась на многие другие языки. Как рассказывал Джеймс Гослинг (*https://oreil.ly/Y_7rk*), Javadoc изначально вызывал споры, потому что «хороший технический писатель справился бы намного лучше», но в мире гораздо больше API, чем технических авторов, которые документируют их, поэтому вполне очевидно, как ценен инструмент, доступный всем.

Однако иногда вам нужно нечто большее, чем документация по API, — гораздо больше, чем вы способны уместить на страницах обзора пакетов и проектов, предоставляемых Javadoc. Ориентированные на конечного пользователя руководства и пошаговые инструкции, подробные сведения об архитектуре и теории, объяснения того, как объединить несколько компонентов… ничего из этого нет в Javadoc.

Как же мы можем удовлетворить эти дополнительные потребности? Ответы со временем менялись. В 80-х годах флагманом стал FrameMaker, новаторский кроссплатформенный графический интерфейс для технических документов. В Javadoc даже включали MIF Doclet для создания красивой печатной API-документации с помощью FrameMaker, но от него осталась только рудиментарная версия для Windows. DocBook XML обладает аналогичными инструментами для создания структуры и ссылок, с открытой спецификацией и кроссплатформенным набором инструментов, но его необработанный формат XML неудобен для применения. Гнаться за инструментами для редактирования оного стало дорого и утомительно, причем даже лучшие из них ощущались неуклюжими и затрудняли процесс написания.

Я очень рад, что нашел более удачный вариант: AsciiDoc (*https://oreil.ly/NYrJI*) предлагает все возможности DocBook в текстовом формате, удобном для

записи и чтения, где выполнение простых действий тривиально, а сложных — возможно. Большинство конструкций AsciiDoc так же легко читаемы и доступны, как и другие упрощенные форматы разметки, например Markdown, которые становятся общепринятыми благодаря онлайн-форумам. А если вам захочется чего-то изысканного, то можете включить сложные уравнения в форматах MathML или LaTeX, отформатированные листинги исходного кода с пронумерованными и связанными выносками для текстовых абзацев, блоки предупреждений различных видов и многое другое.

AsciiDoc был написан на Python и представлен в 2002 году. Текущая официальная версия (и управляющий язык) — это Asciidoctor (*https://oreil.ly/aRRvG*), выпущенный в 2013. Его Ruby-код также можно запускать в JVM через AsciidoctorJ (*https://oreil.ly/UT8EP*) (с плагинами Maven и Gradle) или переносить в JavaScript (*https://oreil.ly/E_6qn*), все это прекрасно работает в средах непрерывной интеграции. Когда вам нужно создать целый сайт со связанной документацией (даже из нескольких репозиториев), инструменты вроде Antora (*https://antora.org*) потрясающе упростят процесс. Сообщество (*https://oreil.ly/PtWwa*) дружелюбно и всегда готово поддержать, а наблюдение за его ростом и прогрессом за прошедший год было вдохновляющим. И если для вас это важно, то уже ведется стандартизация формальной спецификации AsciiDoc (*https://oreil.ly/BaXa8*).

Мне нравится создавать обширную, аккуратную документацию (*https://oreil.ly/H_rSW*) для проектов, которыми я делюсь. С AsciiDoc этот процесс намного упростился: циклы обработки документации убыстрились настолько, что ее шлифовка и совершенствование стали увлекательным занятием (*https://oreil.ly/7sbtj*). Я надеюсь, что вы тоже оцените эти возможности. И если вы решите с головой окунуться в AsciiDoc, то отмечу, что круг в каком-то смысле замкнулся: теперь есть даже Doclet (*https://oreil.ly/9KgQq*) под названием Asciidoclet, который позволяет вам писать Javadoc с помощью AsciiDoc!

Будьте внимательны к контейнерному окружению

Давид Делабассе

Имеется опасность контейнеризации устаревших Java-приложений в существующем виде с их устаревшей виртуальной машиной Java (JVM), поскольку такие JVM из-за своей структуры будут ошибаться в оценке доступных ресурсов при запуске внутри контейнеров Docker.

Фактически контейнеры стали механизмом создания артефактов со средой выполнения кода. Они дают множество преимуществ: определенный уровень изоляции, улучшенное использование ресурсов, возможность развертывания приложений в различных средах и многое другое. Контейнеры также помогают ослабить связь между приложением и базовой платформой, поскольку приложение можно упаковать в портативный контейнер. Этот метод иногда используется для модернизации устаревших приложений. В случае Java контейнер интегрирует устаревшее приложение Java вместе с его зависимостями, в том числе более старую версию JVM, используемую этим приложением.

Практика контейнеризации устаревших приложений Java с их средами, безусловно, продлевает работу старых приложений на современной поддерживаемой инфраструктуре, отделяя их от старой, неподдерживаемой инфраструктуры. Но, кроме потенциальных преимуществ, такая практика сопряжена с особым набором рисков из-за особенностей процессов JVM.

Эти процессы (*https://oreil.ly/h3hTh*) позволяют JVM настраивать себя в зависимости от двух ключевых показателей среды: количества процессоров и объема доступной памяти. Опираясь на данные факторы, JVM определяет важные параметры — например, какой сборщик мусора использовать, как его настроить, размер кучи, размер `ForkJoinPool` и так далее.

Поддержку контейнеров Linux Docker добавили в обновлении 191 JDK 8 (*https://oreil.ly/ C_1AW*). Благодаря ему JVM, полагаясь на группы Linux (*https://oreil.ly/nDIwb*), получает сведения о ресурсах, выделенных контейнеру, в котором она выполняется. Любая JVM более ранней версии не знает, что она запущена в контейнере, и будет получать доступ к метрикам из операционной

системы хоста, а не из самого контейнера. И, учитывая, что в большинстве случаев контейнер настроен на использование только подмножества ресурсов хоста, JVM станет конфигурировать себя по неверным показателям. Это быстро приведет к нестабильной ситуации, в которой контейнер, скорее всего, будет уничтожен хостом, так как попытается задействовать больше ресурсов, чем ему доступно.

Следующая команда показывает, какие параметры были заданы функционалом JVM:

```
java -XX:+PrintFlagsFinal -version | grep ergonomic
```

Поддержка контейнеров JVM включена по умолчанию, но ее можно отключить с помощью флага JVM `-XX:-UseContainerSupport`. Использование этого флага JVM в контейнере с ограниченными ресурсами (процессор и память) даст вам возможность наблюдать и и исследовать, как влияет на систему внутренняя структура JVM с поддержкой контейнера и без нее.

Естественно, запускать устаревшие JVM в контейнерах Docker не рекомендуется. Но если других вариантов нет, такую JVM необходимо хотя бы настроить так, чтобы она не забирала ресурсов больше, чем выделено контейнеру, в котором она выполняется. Идеальное очевидное решение таково: использовать современную поддерживаемую JVM (например, JDK 11 или более поздней версии), которая не только будет по умолчанию понимать, что запущена в контейнере, но также обеспечит актуальную и безопасную среду выполнения.

Поведение — это «легко»; состояние — это сложно

Эдсон Янага

Когда я только начинал изучать объектно-ориентированное программирование, в первую очередь меня, помимо прочего, познакомили со «святой троицей»: полиморфизм, наследование и инкапсуляция. И, честно говоря, мы потратили довольно много времени, пытаясь понять их и работать с ними. Но (по крайней мере, так считаю я), чрезмерное внимание уделялось первым двум концепциям и недостаточное — третьей, самой важной из них: инкапсуляции.

Инкапсуляция позволяет нам укротить рост состояния и сложности, который постоянно наблюдается в разработке ПО. В основе проектирования и программирования комплексных информационных систем лежит идея, что мы можем сделать состояние внутренним, скрыть его от других компонентов и выводить наружу только тщательно разработанный API для любого изменения состояния.

Но как минимум в мире Java нам не удалось распространить некоторые из лучших практик построения хорошо инкапсулированных систем. Свойства JavaBean в слабых классах, которые просто отображают внутреннее состояние через *геттеры* и *сеттеры*, являются общими, и с архитектурами Java Enterprise мы, похоже, популяризировали концепцию, согласно которой большую часть бизнес-логики, если не всю ее, следует реализовывать в служебных классах. Внутри них мы применяем геттеры для извлечения информации, обрабатываем их, чтобы получить результат, а затем помещаем его обратно в наши объекты с помощью сеттеров.

И, когда вылезают ошибки, мы копаемся в логах, используем отладчики и пытаемся выяснить, что происходит с нашим кодом в продакшене. Довольно «легко» обнаружить ошибки, вызванные проблемами с поведением: фрагменты кода делают не то, что им положено. С другой стороны, когда кажется, что код работает правильно, а у нас по-прежнему есть ошибки, все становится намного сложнее. По моему опыту, самые запутанные ошибки — те, которые

вызваны *несоответствующим состоянием*. То есть ваша система достигает состояния, для нее вроде бы невозможного, но вот оно — `NullPointerException` для свойства, которое никогда не должно становиться `null`, отрицательное значение там, где допустимы только положительные, и так далее.

Очень мало шансов определить, какие именно шаги привели к такому несоответствующему состоянию. У наших классов есть слишком изменчивые и слишком легкодоступные плоскости: любой фрагмент кода в любом месте системы может изменить состояние, поскольку для него не имеется сдержек и противовесов.

Мы очищаем предоставленные пользователем входные данные с помощью проверочных фреймворков, но даже тогда этот «безобидный» сеттер не пропадает, и его может вызывать любой фрагмент кода. И я даже не буду обсуждать вероятность того, что кто-то использует оператор `UPDATE` непосредственно в базе данных, чтобы изменить некоторые столбцы в объектах, сопоставленных с БД.

Как же нам решить эту проблему? *Неизменяемость* — один из возможных ответов. Если мы можем гарантировать, что объекты неизменяемы, а согласованность состояния проверяется при создании объекта, у нас никогда не будет несоответствующего состояния в системе. Но стоит принять во внимание, что большинство фреймворков Java не очень хорошо справляются с неизменяемостью, поэтому мы должны по крайней мере стремиться уменьшить изменчивость, насколько возможно. Правильно запрограммированные фабричные методы и конструкторы также могут помочь нам достичь этого минимально изменяемого состояния.

Поэтому не создавайте сеттеры бездумно. Найдите время, чтобы поразмыслить над ними. Вам действительно нужен этот сеттер в коде? И если вы решите, что да, нужен — например, из-за каких-то требований фреймворка, — подумайте, не использовать ли *защитный слой*, чтобы оберегать и проверять ваше внутреннее состояние после данных взаимодействий с сеттером.

Бенчмаркинг — это сложно, JMH поможет

Майкл Хангер

Бенчмаркинг на JVM, особенно микробенчмаркинг, очень сложен. Недостаточно провести наносекундное измерение вокруг вызова метода или цикла и успокоиться на этом. Вы должны учитывать прогрев, компиляцию HotSpot, оптимизации кода — например, встраивание выражений (inlining) и устранение мертвого кода, — многопоточность, согласованность измерений и так далее.

К счастью, Алексей Шипилев, автор многих замечательных инструментов JVM, создал JMH, Java Microbenchmarking Harness (*https://oreil.ly/gR0fd*), в OpenJDK. Он состоит из небольшой библиотеки и подключаемого модуля системы сборки.

Библиотека предоставляет аннотации и утилиты для объявления ваших тестов как классов и методов Java с помощью аннотаций, включая класс `BlackHole`, который использует сгенерированные в тестах значения, чтобы избежать удаления кода. Библиотека также обеспечивает корректную обработку состояний при наличии многопоточности.

Подключаемый модуль системы сборки генерирует JAR с соответствующим инфраструктурным кодом для правильного запуска и измерения тестов. Это включает в себя выделенные фазы прогрева, правильную многопоточность, запуск нескольких форков и усреднение по ним и многое другое.

Инструмент также выдает важные рекомендации о том, как использовать собранные данные и их ограничения. Вот пример того, как измеряется влияние предопределения размера коллекции:

```
public class MyBenchmark {
    static final int COUNT = 10000;
    @Benchmark
    public List<Boolean> testFillEmptyList() {
```

```
        List<Boolean> list = new ArrayList<>();
        for (int i=0;i<COUNT;i++) {
            list.add(Boolean.TRUE);
        }
        return list;
    }
    @Benchmark
    public List<Boolean> testFillAllocatedList() {
        List<Boolean> list = new ArrayList<>(COUNT);
        for (int i=0;i<COUNT;i++) {
            list.add(Boolean.TRUE);
        }
        return list;
    }
}
```

Чтобы сгенерировать проект и запустить его, используйте команду JMH Maven archetype:

```
mvn archetype:generate \
-DarchetypeGroupId=org.openjdk.jmh \
-DarchetypeArtifactId=jmh-java-benchmark-archetype \
-DinteractiveMode=false -DgroupId=com.example \
-DartifactId=coll-test -Dversion=1.0

cd coll-test

# add com/example/MyBenchmark.java

mvn clean install

java -jar target/benchmarks.jar -w 1 -r 1

...
# JMH version: 1.21
...
# Warmup: 5 iterations, 1 s each
# Measurement: 5 iterations, 1 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
```

```
# Benchmark mode: Throughput, ops/time
# Benchmark: com.example.MyBenchmark.testFillEmptyList

...

Result "com.example.MyBenchmark.testFillEmptyList":
  30966.686 ±(99.9%) 2636.125 ops/s [Average]
  (min, avg, max) = (18885.422, 30966.686, 35612.643), stdev = 3519.152
  CI (99.9%): [28330.561, 33602.811] (assumes normal distribution)

# Run complete. Total time: 00:01:45

REMEMBER: The numbers below are just data. To gain reusable insights,
you need to follow up on why the numbers are the way they are.
Use profilers (see -prof, -lprof), design factorial experiments,
perform baseline and negative tests that provide experimental control,
make sure the benchmarking environment is safe on JVM/OS/HW level,
ask for reviews from the domain experts.
Do not assume the numbers tell you what you want them to tell.
Benchmark                          Mode  Cnt     Score     Error   Units
MyBenchmark.testFillAllocatedList thrpt 25  56786.708 ± 1609.633  ops/s
MyBenchmark.testFillEmptyList     thrpt 25  30966.686 ± 2636.125  ops/s
```

Итак, мы видим, что коллекция с предопределенным размером работает почти вдвое быстрее, чем созданная по умолчанию, потому что ее размер не нужно изменять во время добавления элементов. JMH — это мощное средство в вашем наборе инструментов для написания правильных микробенчмарков. Если вы запускаете их в одной и той же среде, они будут сопоставимы, а именно к этому и стоит стремиться при интерпретации их результатов. Их также можно использовать для профилирования, поскольку они обеспечивают стабильные, воспроизводимые результаты. Алексею (*https://oreil.ly/5zWU1*) есть еще много чего сказать по этой теме, если вам интересно.

Преимущества систематизации и проверки архитектурного качества

Дэниел Брайант

Ваш конвейер непрерывной сборки должен быть основным местом, где систематизируются и применяются согласованные архитектурные качества для ваших приложений. Однако недопустимо, чтобы эти автоматизированные проверки заменили регулярные командные обсуждения стандартов и уровней качества, и их определенно не следует использовать, чтобы избежать общения внутри команд или между ними. Тем не менее проверка и публикация показателей качества в конвейере сборки могут предотвратить постепенное ухудшение качества архитектуры, которое в противном случае было бы трудно заметить.

Если вам интересно, почему следует протестировать свою архитектуру, страница мотивации ArchUnit (*https://oreil.ly/q1OCY*) раскрывает эту тему. Все начинается со знакомой истории: давным-давно архитектор нарисовал серию красивых архитектурных диаграмм, которые иллюстрировали компоненты системы и то, как они должны взаимодействовать. Затем проект стал больше, а варианты использования — сложнее, появились новые разработчики, а старые уволились.

В итоге новые функции стали добавлять любым подходящим способом. Вскоре все стало зависеть от всего, и всякое изменение могло оказать непредвиденное влияние на какой угодно компонент. Уверен, многим читателям знаком такой сценарий.

ArchUnit (*https://www.archunit.org*) — это расширяемая библиотека с открытым исходным кодом для проверки архитектуры вашего Java-кода с помощью фреймворка модульного тестирования Java — например, JUnit или TestNG. ArchUnit может проверять как циклические зависимости, так и зависимости между пакетами и классами, слоями и срезами и многое другое. Он делает все это, анализируя байт-код Java и импортируя все классы для анализа.

Чтобы использовать ArchUnit в сочетании с JUnit 4, включите следующую зависимость от Maven Central:

```
<dependency>
   <groupId>com.tngtech.archunit</groupId>
   <artifactId>archunit-junit</artifactId>
   <version>0.5.0</version>
   <scope>test</scope>
</dependency>
```

По сути ArchUnit предоставляет инфраструктуру для импорта байт-кода Java в структуры кода Java.

Это делается с помощью `ClassFileImporter`. Вы можете создавать архитектурные правила, такие как «доступ к сервисным классам должен осуществляться только контроллерами», используя DSL-подобный API, который, в свою очередь, допустимо применять к импортированным классам:

```
import static com.tngtech.archunit.lang.syntax.ArchRuleDefinition;
// ...
@Test
public void Services_should_only_be_accessed_by_Controllers() {
   JavaClasses classes =
      new ClassFileImporter().importPackages("com.mycompany.myapp");
   ArchRule myRule = ArchRuleDefinition.classes()
      .that().resideInAPackage("..service..")
      .should().onlyBeAccessed()
      .byAnyPackage("..controller..", "..service..");
   myRule.check(classes);
}
```

Расширим предыдущий пример: вы также можете применить дополнительные правила доступа на основе уровней, используя этот тест:

```
@ArchTest
public static final ArchRule layer_dependencies_are_respected =
layeredArchitecture()
.layer("Controllers").definedBy("com.tngtech.archunit.eg.controller..")
.layer("Services").definedBy("com.tngtech.archunit.eg.service..")
```

```
.layer("Persistence").definedBy("com.tngtech.archunit.eg.persistence..")
.whereLayer("Controllers").mayNotBeAccessedByAnyLayer()
.whereLayer("Services").mayOnlyBeAccessedByLayers("Controllers")
.whereLayer("Persistence").mayOnlyBeAccessedByLayers("Services");
```

Еще можно проверить, соблюдаются ли соглашения о наименованиях (например, префиксы имен классов), или указать, что класс, названный определенным образом, должен находиться в соответствующем пакете. Начните с изучения примеров ArchUnit (*https:// oreil.ly/Xv8CI*), во множестве содержащихся на GitHub, — они подарят вам новые идеи.

Вы можете попытаться обнаружить и устранить все упомянутые здесь архитектурные проблемы, попросив опытного разработчика или архитектора раз в неделю просматривать код, выявлять нарушения и исправлять их. Однако люди печально известны тем, что действуют непоследовательно, и, когда проект неизбежно сталкивается с нехваткой времени, зачастую в первую очередь приходится жертвовать именно ручной проверкой.

Более практичный метод — систематизировать согласованные архитектурные рекомендации и правила с помощью автоматизированных тестов, используя ArchUnit или другой инструмент, и включить их в процесс непрерывной интеграции. Тогда любые проблемы сумеет быстро обнаружить и устранить тот же инженер, что вызвал их.

Разбивайте проблемы и задачи на небольшие фрагменты

Жанна Боярски

Вы учитесь программировать. Получаете небольшое задание. Пишете под тысячу строк кода. Вы печатаете его и тестируете. Потом добавляете вывод в консоль или используете отладчик. Возможно, делаете перерыв на кофе. Затем пытаетесь понять, что наворотили.

Звучит знакомо? И это всего лишь учебное задание. Рабочие задачи и системы гораздо масштабнее. Для решения масштабных проблем требуется время. Хуже того: слишком многое нужно удерживать в оперативной памяти вашего мозга.

Хороший способ справиться с этим — разбить проблему на небольшие фрагменты. Чем меньше, тем лучше. Заставив одну маленькую часть работать, вы сможете забыть о ней и перейти к следующей. Если вы хотите правильно решать данную задачу, то вам нужно писать автоматические тесты для каждой небольшой проблемы. Вы также должны часто коммитить свой код, чтобы иметь точку отката, если что-то начнет работать не так, как ожидалось.

Я помню, как помогала товарищу по команде, который застрял. Проще всего было бы сделать откат и повторно внести изменения, но на вопрос о том, когда он в последний раз коммитил результаты работы, прозвучал ответ: «Неделю назад». Тут проблем у него стало две: первоначальная и мое нежелание помогать ему отлаживать недельный труд.

После этого случая я провела тренинг для своей команды на тему того, как разбивать задачи на более мелкие части. Старшие разработчики сказали мне, что у них «особенные» задачи, которые «никак нельзя разделить». Когда вы слышите слово «особенный» в такой ситуации, то должны сразу же насторожиться.

Я решила запланировать вторую встречу. От каждого разработчика требовалось привести пример «особенной» задачи, а от меня — помочь разбить

ее. Мы начали с экранной страницы, на разработку которой отводилось две недели. Я разделила задачу так:

- Создать экран *hello world* по нужному URL-адресу — никаких полей, просто выводится *hello world*.
- Добавить функциональность для отображения списка из базы данных.
- Добавить текстовую область.
- Добавить выпадающий список выбора.
- *<Длинный перечень более мелких задач>*.

И знаете что? После выполнения каждой из этих крошечных задач можно делать коммит. Это означает, что коммитить код допустимо много раз в день.

Затем мне сказали, что так можно поступать со страницами, но обработка файлов — «особенное» дело. Что я там говорила о слове «особенный»? Эту задачу я тоже разделила на части:

- Считать строку из файла.
- Проверить первое поле, включая вызов базы данных.
- Проверить второе поле и преобразовать его с помощью бизнес-логики.
- *<Еще куча полей>*.
- Применить первое правило бизнес-логики ко всем полям.
- *<Еще куча правил>*.
- Добавить сообщение в очередь.

Опять же, задача не была особенной. Если вы считаете особенной свою задачу, сделайте паузу и подумайте почему. Скорее всего, вы поймете, что методика разбиения все равно применима.

Наконец разработчик сказал мне, что не сможет закоммитить свой код раньше чем через неделю. В итоге задачу перепоручили мне, и я сделала несколько дополнительных коммитов, чтобы подчеркнуть свою точку зрения. Если подсчитать, то у меня получилось 22 коммита за те 2 дня, которые потребовались, чтобы выполнить задачу. Может быть, если бы он коммитил чаще, то решил бы задачу быстрее!

Создавайте разнообразные команды

Икшель Руис

Много лет назад всякий хороший врач сам все знал и сам все делал: вправлял вывихи, лечил переломы, проводил операцию, пускал кровь. Хороший врач был независимым и самодостаточным, а самостоятельность высоко ценилась.

Перенесемся в сегодняшний день. Объем знаний неимоверно расширился: отдельный индивидуум не может владеть ими всеми и вынужден специализироваться. Чтобы обеспечить адекватность решения от начала до конца, будет задействовано много специалистов, и разные команды должны будут сотрудничать между собой.

Это относится и к разработке программного обеспечения.

Сотрудничество в настоящее время является одной из самых высоко ценимых черт «хороших» профессионалов. В прошлом независимости и самодостаточности хватало, чтобы быть «хорошим». Теперь мы все должны вести себя как экипаж механиков «Формулы-1» — как единое целое.

Сложность в том, чтобы создать одновременно успешные и разнообразные команды.

Четыре типа разнообразия — по опыту работы в отрасли, стране происхождения, карьерному пути и полу — положительно коррелируют с новшествами. В однородной команде, независимо от академического образования, могут встречаться дублирующиеся точки зрения. Тогда как женщины, например, привносят прорывные инновации.

Насколько велико это влияние? В управленческих командах с высоким гендерным разнообразием наблюдалось увеличение прибыли от новшеств на 8%.

Несхожесть членов команды также может привести к ценным находкам, поскольку при наличии участников с разным социальным положением,

жизненным опытом и идеями увеличивается суммарный объем информации, навыков и связей. Люди с более разнообразными взглядами прибегают к конструктивным обсуждениям, чтобы достичь консенсуса. Если обмен идеями ведется в позитивной среде, творческие решения будут возникать естественным путем.

Но увеличение разнообразия групп — непростая задача. Иногда разнородные команды не взаимодействуют эффективно или разделяются на фракции, что приводит к конфликту. Люди предпочитают сотрудничать с теми, кто похож на них.

В сплоченной группе разовьется собственный язык и культура, появится недоверие к посторонним. В командах, члены которых работают на расстоянии и сталкиваются с типовыми ошибками при цифровой связи, особенно часто встречаются проблемы вида «мы против них» и неполная информированность.

Итак, как нам получить преимущества разнообразия и избежать недостатков? Ключ к сотрудничеству — развитие психологической безопасности и доверия внутри вашей команды.

Если мы находимся среди людей, на которых можем положиться, даже если они отличаются от нас, то более уверенно идем на риск и экспериментируем. Когда мы доверяем другим разработчикам, то обращаемся к ним за сведениями или мнениями, которые помогут решить сложную проблему. Тем самым создаются возможности для сотрудничества. Запрашивая обратную связь, мы преодолеваем уязвимые ситуации.

В командах с психологической безопасностью людям легче поверить, что выгода от высказывания своей позиции перевешивает издержки. Участие приводит к меньшему сопротивлению изменениям, и чем чаще люди вовлекаются в обсуждение, тем больше вероятность того, что они предложат новые идеи.

Личность также имеет значение при разработке программного обеспечения, поэтому не менее важно создать атмосферу доверия для людей с разными характерами.

У всех нас есть коллега, который готов протестировать каждую новую библиотеку, фреймворк или инструмент, — кто-то думающий над тем, как использовать или изучить новую блестящую красную игрушку, иногда с удивительными результатами. Некоторые склонны устанавливать новые процессы, стили форматирования кода или шаблоны для коммит-сообщений; если мы

не станем следовать надлежащей процедуре, они укажут нам на это. Вероятно, у вас окажутся товарищи по команде, которые «обещают меньше, но делают больше», а также те, кто размышляют обо всех возможных трудностях: обновлении зависимостей, установке исправлений, угрозах безопасности и т.д. Учитывайте, что каждый по-своему уникален, и не давите слишком сильно.

Мы можем увеличить разнообразие в своих командах по двум параметрам: опыт и личностные черты. Мы добьемся большего успеха как программисты, если достигнем хорошей групповой динамики и продолжим укреплять доверие друг к другу.

Сборки не обязательно должны быть медленными и ненадежными

Дженн Стретер

Некоторое время назад я участвовала в стартапе на ранней стадии, где кодовая база и команда разработчиков росли с каждым днем. По мере того как мы добавляли всё больше и больше тестов, сборки выполнялись всё дольше и дольше. Примерно на восьмиминутной отметке я начала это замечать, и вот почему это конкретное число врезалось мне в память. Время сборки, сначала составлявшее 8 минут, почти удвоилось. Поначалу это было даже неплохо. Я запускала ее, уходила выпить кофе и болтала с коллегами из других команд. Но через несколько месяцев это стало раздражать. Я успевала выпить кофе и узнать, над чем работают все вокруг, но сборки все еще не завершались, поэтому я проверяла соцсети или помогала коллегам в моей команде. Затем мне приходилось переключаться, когда я возвращалась к своей работе.

Кроме того, сборка была ненадежной. Как обычно бывает с любым программным проектом, у нас появилось несколько непредсказуемых тестов. Первое, хотя и наивное, решение состояло в том, чтобы отключить (т. е. `@Ignore`) тесты, которые не проходили. В конце концов дошло до того, что стало проще вносить изменения и полагаться на конвейер непрерывной интеграции (CI), чем запускать тесты локально. Загвоздка с этой тактикой заключалась в том, что она продвигала проблему дальше по линии. Если тест завершался неудачей на этапе CI, отладка занимала гораздо больше времени. И если изначально тест с ошибками проходил и обнаруживался только после слияния кода, он блокировал всю команду, вынуждая нас тратить время на выяснение того, настоящая ли это проблема.

Расстроенная, я попыталась исправить некоторые проблемные тесты. Один из них особенно мне запомнился. Он появлялся только при запуске всего набора тестов, поэтому каждый раз, когда я вносила изменения, мне приходилось ждать результатов более 15 минут. Из-за таких невероятно длинных циклов

обратной связи и общего отсутствия соответствующих данных я впустую потратила несколько дней на поиск этой ошибки.

И случалось подобное не в одной компании. Я часто меняю работу, что дает мне всяческие преимущества: например, я видела, как действуют различные команды. Казалось, что эти проблемы в порядке вещей, пока я не начала работать в компании, где именно такие недостатки устраняются.

Команды, которые следуют технологии повышения производительности разработчиков, методам и доктринам улучшения результатов программистов с помощью анализа данных, могут усовершенствовать свои медленные и ненадежные сборки. Тогда разработчикам живется приятнее, у них повышается производительность, а значит, и заказчики радуются.

Независимо от того, какой инструмент сборки они используют, люди, ответственные за продуктивность разработчиков, могут эффективно измерять производительность сборки и отслеживать исключения и регрессии как для локальных, так и для CI-сборок. Они уделяют время анализу результатов и поиску узких мест в процессе сборки. Когда что-то идет не так, они делятся отчетами (например, отчетами сканирования Gradle-сборки) с товарищами по команде и сравнивают неудачные и удачные сборки, чтобы точно определить проблему — даже если ее не удается воспроизвести на их машинах.

Со всеми этими данными они действительно могут найти способ оптимизировать процесс и уменьшить фрустрацию, с которой сталкиваются программисты. Их труд бесконечен — они регулярно запускают цикл поддержки производительности разработчиков заново. Это непростая задача, но команды, которые работают над ней, способны вообще не допустить возникновения проблем, которые я описала.

«Но на моем компьютере это работает!»

Бенджамин Мушко

С вами когда-нибудь случалось такое: вы присоединились к новой команде или проекту, а теперь пытаетесь пробиться через инфраструктуру, чтобы собрать проект на своей машине? Если да, вы не одиноки. Возможно, у вас возникали такие вопросы:

- Какая версия и дистрибутив JDK требуются для компиляции кода?
- Что делать, если я использую Linux, но все остальные работают на Windows?
- Какая среда IDE применяется и какая версия мне нужна?
- Какую версию Maven или другого инструмента сборки мне нужно установить, чтобы правильно запускать потоки задач разработчика?

Надеюсь, полученный вами ответ не звучал как: «Сейчас я посмотрю, что у меня там установлено». Каждый проект должен иметь четко определенный набор инструментов, совместимых с техническими требованиями для компиляции, тестирования, выполнения и упаковки кода. Если вам повезет, все они задокументированы в плане проекта или в вики, хотя, как мы знаем, документация легко устаревает, а синхронизация инструкций с последними изменениями требует согласованных усилий.

Есть лучший способ преодолеть эту проблему. Следуя концепции *инфраструктура как код (IaC)*, поставщики инструментов разработали *оболочку* — решение, которое помогает обеспечить стандартизированную версию среды выполнения средства сборки без ручного вмешательства. Оно *оборачивает* собой инструкции, необходимые для загрузки и установки среды выполнения. В мире Java есть Gradle Wrapper (*https://oreil.ly/CmZP1*) и Maven Wrapper (*https://oreil.ly/xu50T*). Даже другие инструменты, такие как Bazel, инструмент сборки Google с открытым исходным кодом, предоставляют механизм запуска (*https://oreil.ly/OY7R7*).

Давайте посмотрим, как Maven Wrapper работает на практике. Для создания так называемых файлов-оболочек на вашем компьютере должна быть установлена среда выполнения Maven. Файлы-оболочки представляют собой сценарии, конфигурацию и инструкции, которые каждый разработчик проекта применяет для создания проекта с использованием предопределенной версии среды выполнения Maven. Следовательно, эти файлы необходимо загружать в SCM вместе с исходным кодом проекта для дальнейшего распространения. Ниже выполняется задача Wrapper, предоставленная плагином Takari Maven (*https://oreil.ly/sI2pO*):

```
mvn -N io.takari:maven:0.7.6:wrapper
```

Следующая структура каталогов показывает типичный проект Maven, дополненный файлами-оболочками:

```
.
├── .mvn
│   └── wrapper
│       ├── MavenWrapperDownloader.java
│       ├── maven-wrapper.jar
│       └── maven-wrapper.properties
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    └── ...
```

При наличии файлов-оболочек создать проект на любой машине несложно: запустите желаемую задачу с помощью скрипта *mvnw*. Сценарий автоматически гарантирует, что среда выполнения Maven будет установлена с предопределенной версией, заданной в *maven-wrapper.properties*. Конечно, процесс установки вызывается только в том случае, если среда выполнения еще не доступна в системе.

При выполнении следующей команды сценарий используется для запуска задач *clean* и *install* в системах Linux, Unix или macOS:

```
./mvnw clean install
```

В Windows используйте batch-скрипт, заканчивающийся расширением файла *.cmd*:

```
mvnw.cmd clean install
```

Как насчет выполнения типовых задач в IDE или из конвейера CI/CD? Вы обнаружите, что другие среды выполнения получают ту же конфигурацию из свойств оболочки. Вам просто нужно убедиться, что для вызова сборки используются скрипты оболочки.

Прошли времена «Но на моем компьютере это работает!» — стандартизируйте один раз, создавайте везде! Внедряйте концепцию оболочки в любой проект JVM, чтобы улучшить воспроизводимость сборки и удобство обслуживания.

Дело против fat JAR

Дэниел Брайант

В современной веб-разработке на Java сама мысль об упаковке и запуске приложений во что-либо, кроме fat JAR (самодостаточных или толстых дистрибутивов), воспринимается почти как ересь. Однако в создании и развертывании этих артефактов есть явные недостатки. Одна из очевидных проблем — типично большой размер fat JAR, поглощающий избыточное место и пропускную способность сети. Кроме того, процесс монолитной сборки может занять много времени и привести к тому, что ждущие разработчики переключатся на какое-то иное занятие. Отсутствие общих зависимостей также может привести к несогласованности при использовании утилит, таких как логгирование, и проблемам с реализацией обмена данными или сериализации между приложениями.

Использование fat JAR для развертывания Java-приложений стало популярным вместе с развитием микросервисной архитектуры, DevOps и облачных технологий — например, общедоступного облака, контейнеров и платформы оркестрации. Поскольку приложения были разбиты на набор более мелких служб, которые запускались и управлялись независимо, с операционной точки зрения имело смысл объединить весь код приложения в один исполняемый двоичный файл. Единственный артефакт легче отслеживать, а благодаря автономному выполнению исчезает необходимость в запуске дополнительных серверов приложений. Тем не менее некоторые организации сейчас противостоят этой тенденции и создают skinny JAR (зависимые или тонкие дистрибутивы). Команда инженеров HubSpot обсудила, как перечисленные выше проблемы повлияли на их жизненный цикл разработки, в статье в блоге: «Ошибка в наших JAR: почему мы перестали создавать fat JAR» (*https://oreil.ly/WqX2D*). В итоге они разработали новый подключаемый модуль Maven: SlimFast (*https://oreil.ly/3Kf5Y*). От классического плагина Maven Shade, знакомого большинству разработчиков Java, этот плагин отличается тем, что отделяет код приложения от связанных зависимостей и, соответственно, создает и загружает два отдельных артефакта. Может показаться, что неэффективно создавать и загружать зависимости приложения отдельно, но данный шаг выполняется только в том случае, если зависимости

изменились. Во многих приложениях зависимости меняются редко, следовательно, этот шаг часто не выполняется. JAR-файл зависимостей пакета загружается в удаленное хранилище лишь минимально необходимое количество раз.

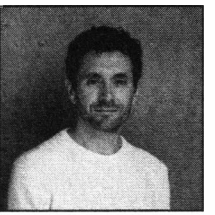
Плагин SlimFast использует плагин Maven JAR для добавления в `Class-Path` в skinny JAR записи манифеста, который указывает на JAR-файл зависимостей, и генерирует файл JSON с информацией обо всех артефактах зависимостей в S3, чтобы их можно было загрузить позже. В процессе развертывания сборка загружает все зависимости приложения, но затем кэширует эти артефакты на каждом из серверов приложений, поэтому данный шаг обычно не выполняется. В итоге во время сборки в удаленное хранилище загружается только skinny JAR-файл приложения, размер которого обычно составляет всего несколько сотен килобайт. В процессе развертывания в целевую среду нужно загрузить только этот же тонкий JAR, что занимает доли секунды.

Одна из основных идей, на базе которых возникли DevOps, заключается в том, что команда разработки и эксплуатации (и все другие команды) должны работать вместе ради достижения общей цели. Так, выбор формата артефакта развертывания — важное решение в рамках задачи, требующей обеспечить возможность непрерывного развертывания функциональности для конечных пользователей. Все обязаны сотрудничать, чтобы понимать требования в аспекте того, как они влияют на практические действия разработчиков и способность управлять ресурсами, задействованными в развертывании.

Плагин SlimFast в настоящее время привязан к AWS S3 для хранения артефактов, но код доступен на GitHub, и настройки можно адаптировать для любого типа внешнего хранилища.

Реставратор кода

Авраам Марин-Перез



Запомни, на самом деле мы трудимся для того, кто будет реставрировать этот предмет лет через сто. Это на него мы хотим произвести впечатление.*

Эта фраза принадлежит Хоби, персонажу романа Донны Тартт «Щегол».

Хоби — реставратор антикварной мебели. Меня особенно радует данная цитата, поскольку она прекрасно выражает то, что я всегда думал о коде: лучший код написан с мыслью о программистах, которые придут после нас.

Я думаю, что современные методы разработки ПО страдают от болезни, вызванной чрезмерной поспешностью. Подобно деревьям в переполненных джунглях, они стремятся перерасти конкурентов. Деревья, соперничающие за свет, часто чрезмерно вытягиваются, становятся высокими и тонкими, восприимчивыми даже к незначительным возмущениям. Сильный ветер или легкая инфекция могут повалить их. Я не говорю, что нам не нужно учитывать краткосрочные выгоды. Напротив, я поощряю такой подход, только не в ущерб долгосрочной стабильности.

Нынешняя индустрия программного обеспечения подобна этим деревьям. Многие «современные» команды сосредоточены только на следующей неделе или месяце. Компании борются просто за то, чтобы прожить еще один день, еще один рывок, еще один цикл. И, похоже, это никого не беспокоит. Разработчики всегда сумеют устроиться в другом месте, как и менеджеры. Предприниматели могут попытаться обналичить деньги до того, как акции компании потеряют ценность. То же верно и для венчурного капитала, поддержавшего первоначальные инвестиции. Слишком часто ключ к успеху в своевременном выходе из игры. В том, чтобы уйти до того, как люди поймут: этот удивительный рост — всего лишь опухоль.

С другой стороны, возможно, это не так уж и плохо. Некоторые предметы мебели рассчитаны на сотни лет, а некоторые, скорее всего, рассыплются

* Перевод А. Завозовой.

в течение десятилетия. Вы можете потратить тысячи на аукционе Sotheby's на сервант с фарфором или пойти в обычный мебельный магазин и за те же деньги обставить себе весь дом. Может быть, нам просто нужно понять эту новую, созданную нами экономику, где все эфемерно и преходяще. Активы в ней не должны служить *долго* — лишь *достаточно* долго. Мы не обязаны создавать вещи, которые выдерживают испытание временем, — только испытание на *прибыльность*.

И все же я считаю, что есть золотая середина. Начинает формироваться новая роль — реставратор кода. Делать то, что с первой попытки будет работать вечно, настолько дорого, что оно того не стоит, но, если сосредоточить внимание только на краткосрочной прибыли, у вас получится код, который рухнет под собственным весом. Вот тут-то и появляется реставратор кода — тот, чья работа заключается не в том, чтобы «воссоздать то же самое, но лучше» (распространенное желание, которое почти всегда оборачивается неудачей), а в том, чтобы взять существующую кодовую базу и медленно изменить ее, чтобы восстановить управляемость. Добавьте сюда несколько тестов, разбейте этот уродливый класс, удалите неиспользуемую функциональность и верните код улучшенным.

Мы, как программисты, должны решить, какое ПО хотим создать. Можно на какое-то время сосредоточиться на прибыли, создать то, что «будет держаться», но в определенный момент нам придется выбирать между долговечностью (тщательно изменять код) и доходом (отказаться от прежнего и начать заново). В конце концов, прибыль очень важна, но есть то, что дороже денег.

Параллелизм в JVM

Марио Фуско

Первоначально JVM располагала только такой моделью параллелизма, как необработанные потоки, и они по-прежнему применяются по умолчанию для написания программ, использующих параллелизм и многопоточность на Java. Однако же 25 лет назад, в период создания Java, аппаратное обеспечение было совершенно иным. Спрос на запуск параллельных приложений был ниже, а преимущества параллелизма нивелировались отсутствием многоядерных процессоров — задачи удавалось разделить, но они не выполнялись одновременно.

В настоящее время ограничения явной многопоточности стали очевидными ввиду доступности распараллеливания и ожиданий от него. Потоки и блокировки слишком низкоуровневые: их сложно использовать правильно, а разобраться в модели памяти Java еще сложнее. Потоки, которые взаимодействуют через общее изменяемое состояние, непригодны для массового параллелизма, что приводит к неопределенным неожиданностям, если доступ не синхронизирован должным образом. Более того, даже если ваши блокировки расположены правильно, цель блокировки состоит в ограничении количества параллельных потоков, что снижает степень параллелизма вашего приложения.

Поскольку Java не поддерживает распределенную память, невозможно масштабировать многопоточные программы горизонтально на нескольких компьютерах. И если писать такие программы тяжело, то тщательно тестировать их практически невозможно, и при обслуживании они часто становятся кошмаром наяву.

Самый простой способ преодолеть ограничения общей памяти — координировать потоки с помощью распределенных очередей вместо блокировок. Здесь передача сообщений заменяет общую память, что также улучшает независимость приложений. Очереди хороши для однонаправленной связи, но могут привести к задержке.

В инструментарии Akka применяется модель акторов, популяризированная Erlang, доступная в JVM и более знакомая программистам на Scala. Каждый актор — это объект, ответственный за управление только своим состоянием.

Параллелизм реализуется посредством потока сообщений между акторами, поэтому их допустимо рассматривать как более структурированный способ использования очередей. Акторов можно организовать в иерархию, что обеспечит встроенную отказоустойчивость и восстановление с помощью супервизии. У акторов также есть отдельные недостатки: нетипизированные сообщения плохо сочетаются с текущим отсутствием сопоставления паттерну в Java, неизменяемость сообщений необходима, но в настоящее время не может быть внедрена в Java, композиция бывает неудобной, и взаимоблокировка между акторами все еще возможна.

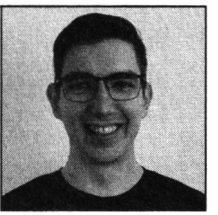
Clojure использует другой подход со встроенной программной транзакционной памятью, превращая кучу JVM в набор транзакционных данных. Как и в обычной БД, они модифицируются с помощью (оптимистичной) транзакционной семантики. Транзакция автоматически повторяется, когда сталкивается с каким-либо конфликтом выполнения. Преимущество такого подхода заключается в неблокировании, что устраняет многие проблемы, связанные с явной синхронизацией. Это облегчает их подготовку. Кроме того, многие разработчики знакомы с транзакциями. К сожалению, этот подход неэффективен в широкомасштабных параллельных системах, где выше вероятность одновременной записи. В таких ситуациях повторные попытки становятся все более дорогостоящими, а производительность может стать непредсказуемой.

Лямбды в Java 8 способствуют использованию в коде свойств функционального программирования — например, неизменяемости и ссылочной прозрачности. В то время как модель актора уменьшает вредные последствия изменяемого состояния, предотвращая совместное использование, функциональное программирование делает состояние *доступным* для совместного использования, поскольку запрещает изменяемость. Распараллеливание кода, состоящего из чистых функций без побочных эффектов, бывает тривиальной задачей, но функциональная программа может оказаться менее эффективной по времени, чем ее императивный эквивалент, и создать бóльшую нагрузку на сборщик мусора. Лямбды также облегчают применение в Java парадигмы реактивного программирования, заключающейся в асинхронной обработке потоков событий.

Для параллелизма нет панацеи, но есть много всяческих вариантов с различными плюсами и минусами. Ваша обязанность как программиста — знать их и выбрать тот, который наилучшим образом соответствует поставленной задаче.

CountDownLatch — друг или враг?

Алексей Сошин



Давайте представим ситуацию, в которой мы хотели бы запустить несколько одновременных задач, а затем дождаться их завершения, прежде чем продолжить. Функция `ExecutorService` упрощает первую часть задачи:

```
ExecutorService pool = Executors.newFixedThreadPool(8);
Future<?> future = pool.submit(() -> {
    // ваша задача
});
```

Но как нам дождаться завершения всех потоков? Класс `CountDownLatch` приходит нам на помощь. Объект `CountDownLatch` принимает количество вызовов в качестве аргумента конструктора. Затем каждый поток содержит ссылку на него, вызывая метод `countDown()` по завершении задачи:

```
int tasks = 16;
CountDownLatch latch = new CountDownLatch(tasks);
for (int i = 0; i < tasks; i++) {
    Future<?> future = pool.submit(() -> {
        try {
            // ваша задача
        }
        finally {
            latch.countDown();
        }
    });
}
if (!latch.await(2, TimeUnit.SECONDS)) {
    // Обработка времени ожидания
}
```

Этот пример кода запустит 16 потоков, затем дождется их завершения, прежде чем продолжить. Однако есть несколько важных моментов, на которые следует обратить внимание:

1. Убедитесь, что вы освободили защелку (`latch`) в блоке `finally`. В противном случае, если возникает исключение, ваш основной поток будет ждать вечно.

2. Используйте метод `await`, который принимает период ожидания. Таким образом, даже если вы забудете о первом пункте, ваш поток рано или поздно продолжит выполнение.

3. Проверьте возвращаемое значение метода. Если время истекло, он вернет значение `false`. Если все задачи удалось выполнить в срок, результатом станет `true`.

Как упоминалось ранее, объект `CountDownLatch` получает значение счетчика при создании. Его нельзя ни увеличить, ни сбросить. Если вы ищете возможности, аналогичные таковым у `CountDownLatch`, но с потенциалом сброса количества, то вместо него вам следует обратить внимание на класс `CyclicBarrier`.

Класс `CountDownLatch` очень полезен во многих ситуациях, а особенно в тех случаях, когда вы тестируете свой параллельный код, так как позволяет гарантировать, что все задачи будут выполнены, прежде чем вы начнете проверять их результаты.

Рассмотрим следующий реальный пример. У вас есть прокси-сервер и встроенный сервер, и вы хотели бы проверить, что при вызове прокси-сервера он вызывает правильный эндпоинт на вашем сервере.

Очевидно, что нет особого смысла отправлять запрос до запуска как прокси-сервера, так и сервера. Одним из решений является передача `CountDownLatch` обоим методам и продолжение теста только тогда, когда обе стороны будут готовы:

```
CountDownLatch latch = new CountDownLatch(2);
Server server = startServer(latch);
Proxy proxy = startProxy(latch);
boolean timedOut = !latch.await(1, TimeUnit.SECONDS);
assertFalse(timedOut, "Timeout reached");
// Продолжить тест, если утверждение проходит
```

Вам просто нужно убедиться, что оба метода — `startServer` и `startProxy` — вызывают `latch.countDown` сразу, как только они успешно запущены.

Класс `CountDownLatch` очень полезен, но есть одна важная загвоздка: вы не должны применять его в рабочем коде, который использует параллельные библиотеки или фреймворки, такие как сопрограммы Kotlin, Vert.x или Spring WebFlux. Все потому, что `CountDownLatch` блокирует текущий поток выполнения. Разные модели параллелизма плохо сочетаются друг с другом.

Декларативное выражение — вот путь к параллелизму

Рассел Уиндер

Вначале Java была императивным объектно-ориентированным языком программирования. На самом деле это все еще так. Однако с годами Java развивалась, на каждом этапе становясь все более и более декларативным языком. Императивный подход — это все, что связано с кодом, явно указывающим компьютеру, что делать. Декларативный подход — это код, который выражает цель, абстрагируясь от способа ее достижения. Абстракция лежит в основе программирования, и поэтому переход от императивного кода к декларативному естественен.

Декларативное выражение основывается на использовании функций более высокого порядка — таких, которые принимают функции в качестве параметров и/или возвращают функции. Изначально оно не было неотъемлемой частью Java, но с появлением Java 8 переместилось в центр внимания: Java 8 стала поворотным моментом в эволюции языка, позволив заменить императивное выражение декларативным.

Ради примера — пусть тривиального, но показательного для основной проблемы, — напишем функцию, которая возвращает список `List`, содержащий квадраты переданного аргумента `List`. Императивно мы могли бы написать:

```
List<Integer> squareImperative(final List<Integer> datum) {
  var result = new ArrayList<Integer>();
  for (var i = 0; i < datum.size(); i++) {
    result.add(i, datum.get(i) * datum.get(i));
  }
  return result;
}
```

Функция создает абстракцию над неким низкоуровневым кодом, скрывая детали от кода, который ее использует.

С Java 8 и выше мы можем применять потоки и выражать алгоритм более декларативным способом:

```
List<Integer> squareDeclarative(final List<Integer> datum) {
  return datum.stream()
              .map(i -> i * i)
              .collect(Collectors.toList());
}
```

Этот фрагмент кода излагает на более высоком уровне то, что должно быть сделано. Детали того, как это сделать, оставлены для реализации библиотеки. Классическая абстракция. Правда, реализация находится внутри функции, которая уже абстрагируется и скрывается, но что бы вы предпочли сохранить: низкоуровневую императивную реализацию или высокоуровневую декларативную реализацию?

Почему это так важно? Вышеизложенный пример — классический образец затруднения параллельных вычислений. Оценка каждого результата зависит только от элемента входных данных; связи нет. Следовательно, мы можем написать:

```
List<Integer> squareDeclarative(final List<Integer> datum) {
  return datum.parallelStream()
              .map(i -> i * i)
              .collect(Collectors.toList());
}
```

Поступая таким образом, мы получаем максимальный параллелизм, который библиотека способна извлечь из платформы. Дело в том, что мы абстрагируемся от подробностей «как?» и, сосредоточившись только на цели, легко превращаем *последовательное* вычисление с параллельными данными в *параллельное*.

В качестве упражнения предложу читателям (попытаться) написать параллельную версию императивного кода, если они того пожелают. Почему? Потому что для проблем с параллельными данными использование потоков является правильной абстракцией. Делать что-либо еще — значит отрицать Java 8 как этап эволюции Java.

Поставляйте качественное ПО быстрее

Бурк Хуфнагель

Я настоятельно рекомендую вам принять руководящий принцип: «поставляйте качественное ПО быстрее», поскольку он описывает, как вам удовлетворить запросы пользователей. Кроме того (и, возможно, это даже важнее), если вы начнете следовать ему, то вас, вероятно, будет ждать более приятная и интересная карьера. Чтобы понять, в чем суть, давайте рассмотрим три части этой важной идеи:

1. *Поставка* — вы обязаны взять на себя ответственность не только за написание и отладку кода. Что бы вы ни думали, вам платят не за то, чтобы вы писали код. Вам платят за то, чтобы вашим пользователям было проще делать то, что они считают ценным. Пока ваш код не будет запущен в производство, ваш тяжкий труд не принесет им никакой пользы.

   Чтобы переключить внимание с написания кода на поставку программного обеспечения, вы должны изучить общий процесс поставки изменений в производство, а затем выполнить два ключевых действия:

   - Убедитесь, что вы *не делаете* ничего, что замедлило бы процесс: например, угадываете смысл расплывчатых требований вместо того, чтобы попросить разъяснения перед выполнением задачи.

   - Убедитесь, что вы *делаете* все, чтобы ускорить процесс: например, пишете и запускаете автоматические тесты, показывая тем самым, что ваш код соответствует приемочным критериям.

2. *Качественное ПО* — это условное обозначение двух концепций, с которыми вы уже наверняка знакомы: «создать то, что нужно» и «создать так, как нужно». Согласно первой, необходимо обеспечить, чтобы ваша работа соответствовала всем требованиям и приемочным критериям. Вторая идея заключается в том, что нужно писать код, который легко понятен другому программисту, чтобы он мог успешно исправлять ошибки или добавлять новые функции.

Хотя это может показаться простым делом, особенно если вы следуете такой практике, как разработка на основе тестирования (TDD), многие команды отклоняются по одному из двух направлений:

- Непрограммисты вынуждают разработчиков как можно скорее предоставить новые функции и не оставляют времени на нормальную проверку. Программисту остается надеяться, что он вернется позже и «сделает все как следует».

- Иногда программисты, которые недавно чему-то научились, пытаются использовать свои новообретенные знания везде, где получится, даже если знают, что более простое решение будет работать так же хорошо.

В любом случае баланс теряется, и, пока его не восстановят, будет возникать технический долг, из-за которого удлиняются сроки поставки важных обновлений вашим пользователям.

3. *Быстрее* относится как к *поставке,* так и к *качественному ПО,* и тут могут возникнуть затруднения, потому что люди, пытающиеся быстро выполнять сложные задачи, как правило, совершают ошибки. Для меня очевидно, что здесь требуется, помимо прочего:

   - Задействовать такой процесс, как TDD, для создания автоматизированных тестов, а затем регулярно выполнять автоматизированные модульные, интеграционные и пользовательские приемочные тесты для проверки поведения системы.

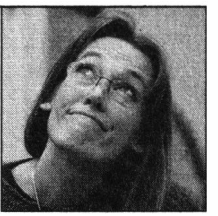
   - Создать и запустить автоматизированный процесс, который выполняет все тесты в нескольких средах и (если все они успешно пройдены) развертывает код в рабочей среде.

   Оба этих процесса будут выполняться несколько раз и потребуют большого внимания к деталям — а именно такие задачи компьютер решает быстрее и точнее человека. Это хорошо, потому что у меня есть еще одна рекомендация: чаще внедряйте изменения в рабочую среду, чтобы на каждое развертывание приходилось меньше изменений и, следовательно, снижалась вероятность возникновения проблем, а ваши пользователи быстрее получали результат вашей работы.

Добиться, чтобы принцип «поставляйте качественное ПО быстрее» стал для вас руководящим, — одновременно сложная и увлекательная задача. Имейте в виду, что потребуется время для поиска и исправления всех мест, требующих обработки, но награда того стоит.

Не знаете, который час?

Кристин Горман



«В какое время прибывает самолет Scandinavian Airlines из Осло в Афины в понедельник?» Почему вопросы, которые кажутся такими простыми в повседневной жизни, так сложны в программировании? Время — это ведь элементарная штука, всего лишь течение секунд, а компьютер отлично умеет их отсчитывать.

```
System.currentTimeMillis() = 1570964561568
```

Ответ формально правильный, но `1570964561568` — это не то, что мы хотим увидеть, когда спрашиваем, который час. Для нас удобнее нечто вроде «13:15, 13-е октября 2019 г.».

Оказывается, время — это две разные вещи. С одной стороны, у нас проходят секунды. С другой, есть несчастливый брак между астрономией и политикой. Ответ на вопрос «Который час?» зависит от положения солнца на небе относительно ваших координат, а также от политических решений, принятых в данном регионе на текущий момент.

Многие проблемы, с которыми мы сталкиваемся при работе с датой и временем в коде, возникают из-за смешения этих двух понятий. Вам поможет использование самой поздней версии библиотеки `java.time` (или Noda Time (*https://nodatime.org*) в .NET). Вот три основных понятия, которые помогут правильно рассуждать о времени: `LocalDateTime`, `ZonedDateTime` и `Instant`.

`LocalDateTime` относится к концепции «13:15, 13-е октября 2019 г.». Таких моментов на временной шкале может быть сколько угодно. `Instant` указывает *определенную* точку на временной шкале, которая будет иметь одинаковое значение что в Бостоне, что в Пекине. Чтобы перейти от `LocalDateTime` к `Instant`, потребуется ввести понятие часовых поясов `TimeZone`. Сейчас в него включены смещения всемирного координированного времени (UTC) и правила перехода на сезонное время (DST). Также есть `ZonedDateTime`, который, по сути, является `LocalDateTime` с `TimeZone`.

Какие из них вы используете? Здесь так много подводных камней! Позвольте показать вам некоторые из них.

Допустим, мы пишем ПО для организации международной конференции. Сработает ли это?

```
public class PresentationEvent {
  final Instant start, end;
  final String title;
}
```

Не-а.

Хотя нам нужно представить здесь определенный момент времени, в случае будущих событий, даже если нам известны время и часовой пояс, мы неспособны определить момент заранее, потому что правила DST или параметры UTC могут измениться между «сейчас» и «тогда». Нам потребуется `ZonedDateTime`.

Как насчет регулярно происходящих событий, таких как полет? Сработает ли это?

```
public class Flight {
  final String flightReference;
  final ZonedDateTime departure, arrival;
}
```

Не-а.

Такой вариант может приводить к сбою два раза в год. Представьте себе рейс, вылетающий в субботу в 10:00 вечера и прибывающий в воскресенье в 6:00. Что произойдет, если мы переведем стрелки на час назад из-за перехода на сезонное время? Если самолет не будет бесполезно кружить в течение дополнительного часа, он приземлится в 5:00, а не в 6:00. Когда мы продвинемся вперед на один час, он прибудет в 4:00. Для повторяющихся продолжительных событий невозможно зафиксировать как начало, так и конец. Вот что нам нужно:

```
public class Flight {
  final String flightReference
  final ZonedDateTime departure;
  final Duration duration;
}
```

А как насчет мероприятий, которые начинаются в 2:30 ночи? Какие именно «2:30»? При переводе стрелок назад их два, при переводе вперед их нет. В Java для обработки осеннего перехода на сезонное время применяются следующие методы:

```
ZonedDateTime.withEarlierOffsetAtOverlap()
ZonedDateTime.withLaterOffsetAtOverlap()
```

В Noda Time можно явно указать оба перехода на сезонное время с помощью `Resolvers`.

Я лишь слегка затронула возможные проблемы, но, как говорится, хорошие инструменты — уже половина работы. Используйте `java.time` (или Noda Time), и вы избавите себя от множества ошибок.

Не скрывайте от себя все инструменты, используя IDE

Гейл Оллис

Без какого инструмента никак не обойтись ни одному Java-программисту? Eclipse? IntelliJ IDEA? NetBeans? Нет. Это *javac*. Без него всё, что у вас есть, — файлы с текстом странного вида. Если без интегрированных сред разработки (IDE) писать код можно — спросите людей вроде меня, трудившихся в прежние времена, — то без необходимых инструментов разработки программировать невозможно.

Учитывая, что такие инструменты, как *javac*, незаменимы для выполнения задачи, удивительно, как редко люди напрямую используют их. Важно, знаете ли вы, как эффективно применять IDE, но решающее значение имеет то, понимаете ли вы, что она делает и как.

Когда-то давно я участвовала в создании проекта с двумя подсистемами, одной на C++, а другой на Java. Те, кто работал на C++, применяли выбранный ими редактор и командную строку. Java-программисты использовали IDE. Однажды заклинание для взаимодействия с системой контроля версий изменилось. Программисты C++ просто внесли правку в командной строке и немедленно двинулись дальше, но команда Java потратила все утро на борьбу со своей конфигурацией Eclipse. К продуктивной работе они вернулись только во второй половине дня.

Эта прискорбная история подвергает сомнению мастерство той команды Java в использовании выбранных ими инструментов. Но также она показывает, насколько те программисты отдалились в своих повседневных процессах от основных инструментов собственной профессии, работая исключительно в среде IDE. Сокрытие информации — отличный принцип, позволяющий сосредоточиться на полезной абстракции, а не на массе деталей. Но здесь подразумевается, что вы не отказываетесь от знания подробностей вообще, а вникаете в них лишь в тех случах, когда это уместно.

Полагаясь исключительно на IDE, программист может утратить навыки применения своих инструментов, потому что IDE намеренно скрывает

внутренние компоненты. Конфигурация часто формируется просто в результате следования чужим инструкциям, и порой о ней забывают, как только она будет выполнена. Есть много преимуществ в том, чтобы также знать, как непосредственно использовать основные инструменты:

- Сценарии вида «На моем компьютере это работает» менее вероятны, и их не так сложно разрешить, если вы понимаете взаимосвязи между инструментами, исходным кодом, другими ресурсами и сгенерированными файлами. Также вам проще будет разобраться, что нужно упаковать для установки.

- Вы сумеете чрезвычайно быстро и легко настраивать различные параметры. Начните с команд вроде `javac --help`, чтобы посмотреть, какие возможности они дают вам.

- Знакомство с основными инструментами имеет большое значение, когда вы помогаете людям, использующим другую среду. Также это пригодится, если что-то пойдет не так: трудно устранить неполадки, когда не работают встроенные инструменты. Визуализация процесса лучше в командной строке, и вы можете изолировать части процесса, в точности как при отладке кода.

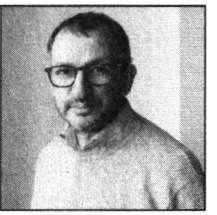
- У вас появляется доступ к более обширному набору инструментов. Вы можете интегрировать любую комбинацию инструментов, имеющих интерфейс командной строки (например, скрипты или команды Linux), а не только те, которые поддерживаются в IDE.

- Учтите: конечные пользователи не будут запускать ваш код в среде IDE! Если вам хочется произвести хорошее впечатление на пользователя, протестируйте всё с самого начала, запустив код так, как он будет выполняться на компьютере пользователя.

Ничто из вышесказанного не отменяет преимуществ IDE. Но, чтобы обрести отличные навыки в выбранном ремесле, разбирайтесь в своих основных инструментах и не давайте им заржаветь.

Не меняйте свои переменные

Стив Фримен



Я стараюсь добавлять как можно больше переменных `final`, потому что мне легче рассуждать о неизменяемом коде.

При написании кода это упрощает мне жизнь, что для меня первоочередная задача — я уже потратил слишком много времени, пытаясь точно выяснить, как изменяется содержимое переменной на протяжении всего блока кода.

Конечно, поддержка неизменяемости в Java более ограничена, чем в некоторых других языках, но мы все равно можем кое-что сделать.

Назначить однажды

Вот небольшой пример структуры, которую я вижу повсюду:

```
Thing thing;
if (nextToken == MakeIt) {
    thing = makeTheThing();
} else {
    thing = new SpecialThing(dependencies);
}
thing.doSomethingUseful();
```

Я не наблюдаю здесь недвумысленного указания на то, что мы собираемся установить значение переменной `thing` перед тем, как использовать его и больше уже не менять.

Мне требуется время, чтобы пройтись по коду и понять, что она не примет значение `null`. Также здесь притаилась проблема, ждущая, когда нам придется добавить больше условий, не совсем правильно понимая логику. Да, современные IDE предупредят о незаданном значении — но, опять же, многие программисты игнорируют предупреждения.

На первом этапе решения можно использовать *условное* выражение:

```
final var thing = nextToken == MakeIt
                  ? makeTheThing()
                  : new SpecialThing(dependencies);
thing.doSomething();
```

Единственный результат выполнения этого кода — присвоение переменной `thing` значения.

Следующий шаг — обернуть это поведение в функцию, которой я могу дать описательное имя:

```
final var thing = aThingFor(nextToken);
thing.doSomethingUseful();

private Thing aThingFor(Token aToken) {
    return aToken == MakeIt
           ? makeTheThing()
           : new SpecialThing(dependencies);
}
```

Теперь жизненный цикл `thing` легко увидеть. Этот рефакторинг часто показывает, что `thing` используется только один раз, поэтому я могу удалить переменную:

```
aThingFor(aToken).doSomethingUseful();
```

Данный подход позволяет нам подготовиться к тому моменту, когда условие неизбежно усложнится. Обратите внимание, что оператор `switch` упрощается, если не использовать повторяющееся слово `break`:

```
private Thing aThingFor(Token aToken) {
    switch (aToken) {
        case MakeIt:
            return makeTheThing();
        case Special:
            return new SpecialThing(dependencies);
        case Green:
            return mostRecentGreenThing();
```

```
        default:
            return Thing.DEFAULT;
    }
}
```

Локализовать область видимости

Вот еще один вариант:

```
var thing = Thing.DEFAULT;
// много кода, в котором нужно разобраться и понять что такое nextToken
if (nextToken == MakeIt) {
    thing = makeTheThing();
}
thing.doSomethingUseful();
```

Такой вариант хуже, потому что присвоения значений для переменной `thing` расположены далеко друг от друга и могут даже не произойти. Опять же, мы извлекаем это во вспомогательный метод:

```
final var thing = theNextThingFrom(aStream);

private Thing theNextThingFrom(Stream aStream) {
    // много кода, в котором нужно разобраться
    if (nextToken == MakeIt) {
        return makeTheThing();
    }
    return Thing.DEFAULT;
}
```

В качестве альтернативы — дальнейшее разделение проблем:

```
final var thing = aThingForToken(nextTokenFrom(aStream));
```

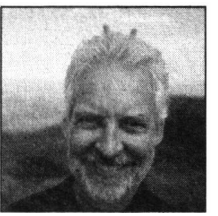
Благодаря локализации во вспомогательный метод области действия всего, что является изменяемым, код верхнего уровня становится предсказуемым. Наконец, хотя некоторые программисты к этому не привыкли, мы могли бы попробовать потоковый подход:

```
final var thing = nextTokenFrom(aStream)
                    .filter(t -> t == MakeIt)
                    .findFirst()
                    .map(t -> makeTheThing())
                    .orElse(Thing.DEFAULT);
```

Регулярно замечаю: когда я пытаюсь найти все, что не меняется, мне приходится более тщательно продумывать свой проект и устранять потенциальные ошибки. Это вынуждает меня четко представлять, где что-то может измениться, и включать такое поведение в локальные области видимости.

Научитесь использовать SQL-мышление по максимуму

Дин Уэмплер



Взгляните на этот запрос:

```
SELECT c.id, c.name, c.address, o.items FROM customers c
JOIN orders o
ON o.customer id = c.id
GROUP BY c.id
```

Мы получаем информацию обо всех клиентах, у которых есть заказы, включая их имена и адреса, а также детали их заказов. Четыре строчки кода. Этот запрос может понять любой, у кого есть небольшой опыт работы с SQL, в том числе непрограммист.

Теперь подумайте о реализации в Java. Мы могли бы объявить классы для `Customer` и `Order`.

Я помню, как консультанты с благими намерениями говорили, что мы также должны создавать классы для инкапсуляции коллекций, а не использовать «голые» коллекции Java.

Нам все еще нужно создать запрос в базу данных, поэтому мы применяем инструмент объектно-реляционного сопоставления (ORM) и пишем для этого код. Четыре строчки быстро превращаются в десятки или даже сотни. Несколько минут, которые потребовались для написания и уточнения SQL-запроса, растягиваются на часы или дни редактирования, создания модульных тестов, проверки кода и так далее.

Разве мы не можем просто реализовать все решение с помощью только SQL-запроса? *Точно* нет? Даже если действительно такой возможности нет, нельзя ли нам устранить все лишнее и написать только то, что необходимо?

Рассмотрим свойства SQL-запроса:

Нам не нужна новая таблица для вывода объединенного результата, поэтому мы ее не создаем.

Самым большим недостатком *прикладного* объектно-ориентированного программирования была вера в то, что требуется точно воспроизводить свою модель предметной области в коде. В действительности же несколько определений основных типов полезны для инкапсуляции и наглядности, но все, что нам нужно в остальное время работы, — кортежи, наборы, массивы и т. д. Ненужные классы становятся бременем по мере развития кода.

Запрос является декларативным.

Нигде в нем не сообщается базе данных, как выполнить запрос. В нем просто указаны *реляционные ограничения*, которым должна удовлетворять БД. Поскольку Java — императивный язык, мы склонны писать код, который говорит, что делать. Но вместо этого нам следует объявить ограничения и желаемые результаты, а затем изолировать реализацию «как?» в одном месте или делегировать библиотеке, способной реализовать ее для нас. Как и функциональное программирование, SQL декларативен. В функциональном программировании эквивалентные декларативные реализации достигаются с использованием составных примитивов вроде *map*, *filter*, *reduce* и так далее.

Язык, специфичный для конкретной предметной области (DSL), хорошо соответствует этой проблеме.

Порой DSL несколько противоречивы. Разработать хороший очень сложно, и его реализация бывает запутанной. SQL — это DSL данных, что необычно, однако его долговечность доказывает, насколько хорошо он отражает типичные потребности в обработке данных.

Все приложения на самом деле применяются для обработки данных. В конечном счете мы пишем только программы для управления данными, и не важно, что мы об этом думаем. Примите этот факт, и ненужный шаблон сразу выдаст себя, а вы сможете писать только то, что необходимо.

События между компонентами Java

А. Махди Абдель-Азиз

Одна из основных концепций объектной ориентации в Java заключается в том, что каждый класс допустимо рассматривать как *компонент*. Их можно расширять или объединять для формирования более крупных компонентов. Конечное приложение также считается компонентом. Они подобны кубикам Lego, из которых создается большая структура.

Событие в Java — это действие, изменяющее состояние компонента. Например, если ваш компонент представляет собой кнопку, то нажатие на эту кнопку является событием, которое изменяет состояние *нажимаемой* кнопки.

События не обязательно происходят только в визуальных компонентах. Например, у вас может быть событие на USB-компоненте, к которому *подключено устройство*. Или событие в сетевом компоненте, при котором *передаются данные*. События помогают убрать зависимости между компонентами.

Предположим, у нас есть компоненты `Oven` (духовка) и `Person` (человек). Они существуют параллельно и работают независимо друг от друга. Нам не стоит делать `Person` частью `Oven` или наоборот. Для создания «умного дома» нам потребуется, чтобы `Oven` приготовила еду, как только `Person` проголодается.

Вот две возможные реализации:

1. `Oven` проверяет `Person` через определенные короткие интервалы. Это раздражает `Person` и затратно для `Oven`, если мы хотим осуществлять проверку нескольких экземпляров `Person`.

2. У `Person` есть общедоступное событие — `Hungry` (голод), на которое подписывается `Oven`. Как только событие `Hungry` происходит, `Oven` получает уведомление и начинает готовить еду.

Второе решение использует событийную архитектуру для эффективной связи между компонентами и без прямого соединения между Person и Oven, ведь Person будет запускать событие, и любой компонент, например Oven, Fridge (холодильник) и Table (стол), может слушать это событие без какой-либо специальной обработки от компонента Person.

Реализация событий для компонента Java способна принимать различные формы в зависимости от того, как они должны обрабатываться.

Чтобы реализовать минимальный Hunger Listener для компонента Person, сначала создайте интерфейс слушателя:

```
@FunctionalInterface
public interface HungerListener {
   void hungry();
}
```

Затем в классе Person определите список для хранения слушателей:

```
   private List<HungerListener> listeners = new ArrayList<>();
```

Определите API для добавления нового слушателя:

```
public void addHungerListener(HungerListener listener) {
   listeners.add(listener);
}
```

Вы можете создать аналогичный API для удаления слушателя. Кроме того, напишите метод для запуска действия «испытывать голод», чтобы уведомлять всех слушателей о событии:

```
public void becomesHungry() {
   for (HungerListener listener : listeners)
      listener.hungry();
}
```

Наконец, добавьте код в класс Oven, который прослушивает событие и реализует действие при запуске события:

```
Person person = new Person();
person.addHungerListener(() -> {
   System.err.println("The person is hungry!");
   // The person is hungry – «Человек проголодался».
   // Здесь Oven начинает совершать действия
});
```

И попробуйте вызвать метод:

```
   person.becomesHungry();
```

Для полностью несвязанного кода последний раздел должен находиться в независимом классе, который имеет экземпляр `Person` **и** `Oven` **и обрабатывает логику между ними. Аналогично мы можем добавить другие действия для** `Fridge`, `Table` **и так далее. Все они получат уведомление только после того, как** `Person becomesHungry`.

Циклы обратной связи

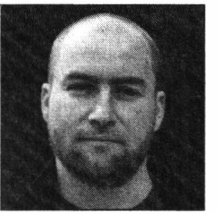
Лиз Кио

- Поскольку наши менеджеры по продуктам не знают, чего хотят, они узнают об этом от клиентов. Иногда они понимают их неправильно.
- Поскольку наши менеджеры по продуктам не знают всего о системах, они приглашают других экспертов стать заинтересованными сторонами в проекте. Заинтересованные стороны понимают их неправильно.
- Поскольку я не знаю, какую задачу решать своим кодом, я узнаю об этом у наших менеджеров по продуктам. Иногда мы понимаем друг друга неправильно.
- Поскольку я допускаю ошибки при написании кода, я работаю с IDE. Среда IDE поправляет меня, когда я совершаю промашки.
- Поскольку я допускаю ошибки в понимании существующего кода, я использую статически типизированный язык. Компилятор поправляет меня, когда я совершаю промашки.
- Поскольку я совершаю ошибки в процессе размышлений, я работаю в паре. Мой напарник поправляет меня, когда я ошибаюсь.
- Поскольку мой напарник — человек и тоже допускает ошибки, мы пишем модульные тесты. Модульные тесты исправляют нас, когда мы совершаем промашки.
- Поскольку у нас есть команда, которая также занимается программированием, мы интегрируем свой код с их кодом. Наш код не будет компилироваться, если мы ошибаемся.
- Поскольку наша команда допускает ошибки, мы пишем приемочные тесты, которые проверяют всю систему. Приемочные тесты провалятся, если мы ошибемся.
- Поскольку мы допускаем ошибки при написании приемочных тестов, то собираем трех друзей вместе, чтобы обсудить их. Наши друзья скажут нам, если мы ошибаемся.

- Поскольку мы забываем запускать приемочные тесты, то заставляем нашу сборку запускать их за нас. Сборка скажет нам, если мы ошибаемся.
- Поскольку мы не можем продумать каждый сценарий, мы привлекаем тестировщиков для изучения системы. Тестировщики скажут нам, если что-то не так.
- Поскольку мы заставили программу работать только на ноутбуке Генри, мы развертываем систему в реалистичной среде. Тесты скажут нам, если что-то идет не так.
- Поскольку мы иногда неправильно понимаем своего менеджера по продуктам и другие заинтересованные стороны, мы показываем им систему. Заинтересованные стороны скажут нам, если мы ошибаемся.
- Поскольку наш менеджер по продуктам иногда неправильно понимает людей, которым нужна система, мы запускаем систему в производство. Люди, которым она требуется, скажут нам, если мы ошибаемся.
- Поскольку люди чаще замечают, когда что-то идет не так, чем когда все работает как должно, мы не просто полагаемся на мнения. Мы используем аналитику и данные. Данные подскажут нам, если мы ошибаемся.
- Поскольку рынок постоянно меняется, однажды мы окажемся неправы, даже если раньше были правы.
- Поскольку промахи дорого обходятся, мы делаем всё это так часто, как только можем. Таким образом, мы всегда ошибаемся только чуточку.
- Не задумывайтесь о том, чтобы сделать все правильно. Беспокойтесь о том, *как* вы узнаете о проблеме и как проще будет исправить последствия, когда это случится. Потому что вы, скорее всего, ошиблись.
- Ошибаться — это нормально.

На всю катушку

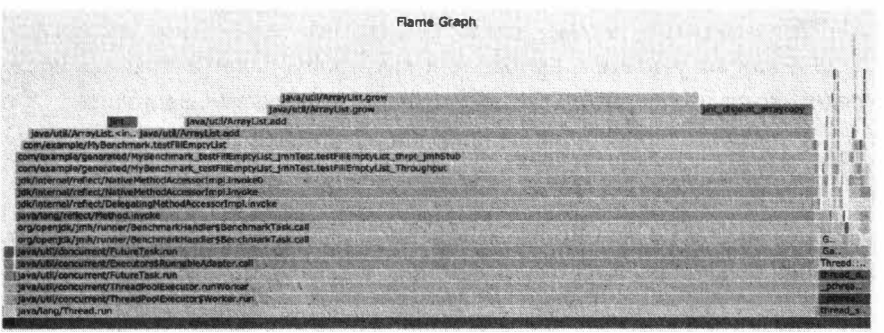
Майкл Хангер



Традиционные профилировщики Java используют либо инструментирование байтового кода, либо выборку (взятие трассировок стека с короткими интервалами), чтобы определить, где теряется время. Оба подхода добавляют свои собственные перекосы и странности. Понимание результатов работы этих профилировщиков — отдельное искусство, требующее немалого опыта.

К счастью, Брендан Грег (*https://oreil.ly/dhd5O*), инженер по производительности из Netflix, придумал «графики пламени» *(flame graphs)* (*https://oreil.ly/2kCDd*), хитроумный вид диаграммы для трассировки стека, которые могут быть собраны практически из любой системы.

«График пламени» сортирует и агрегирует трассировки каждого уровня стека, так что их количество на уровне отображает процент от общего времени, затраченного на эту часть кода. Отображение этих блоков в виде реальных блоков (прямоугольников) с шириной, пропорциональной процентной доле, и наложением блоков друг на друга оказалось очень наглядным.



«Языки пламени» представляют течение процесса снизу вверх от точки входа программы или потока (`main` или цикл событий) до завершения выполнения — на кончиках языков пламени. Обратите внимание, что порядку «слева направо» не придается никакого значения. Обычно это просто сортировка по алфавиту. То же самое касается и цветов.

Важны только относительная ширина и глубина стека.

Вам сразу видно, отнимают ли определенные части программы неожиданно много времени. Чем выше на диаграмме это происходит, тем хуже, особенно если у вас там очень широкий блок: тогда вы понимаете, что нашли узкое место, которое не позволяет делегировать работу куда-нибудь еще. Устранив недостатки, выполните повторное измерение и, если общая проблема с производительностью не исчезнет, просмотрите диаграмму, чтобы получить новые показания.

Многие современные инструменты устраняют недостатки традиционных профилировщиков, используя внутреннюю функцию JVM (`AsyncGetCallTrace`), которая позволяет собирать трассировки стека за пределами безопасных точек. Кроме того, они объединяют измерение операций JVM с кодом аппаратной платформы и системными вызовами ОС, так что время, проведенное в сети, в процессах ввода/вывода (I/O) или сборки мусора, также попадает на график пламени.

Такие инструменты, как Honest Profiler, perf-map-agent, async-profiler и даже IntelliJ IDEA, заметно упрощают сбор информации и создание графиков пламени.

В большинстве случаев вы просто загружаете инструмент, сообщаете ему идентификатор (PID) вашего Java-процесса и указываете инструменту запускаться в течение определенного времени, генерируя интерактивную масштабируемую векторную графику (SVG):

```
# загрузить и распаковать async profiler для вашей ОС из:
# https://github.com/jvm-profiling-tools/async-profiler
./profiler.sh -d <duration> -f flamegraph.svg -s -o svg <pid> && \
open flamegraph.svg -a "Google Chrome"
```

SVG, создаваемый инструментами, не только красочен, но и интерактивен. Вы можете увеличивать масштаб разделов, искать символы и многое другое.

«Графики пламени» — это впечатляюще мощный инструмент, позволяющий быстро получить общее представление о характеристиках производительности ваших программ. Вы можете сразу увидеть проблемные точки и сосредоточиться на них. Если вы добавите параметры, не связанные с JVM, это также поможет увидеть общую картину.

Следуйте скучным стандартам

Адам Биен

В начале эпохи Java на рынке были десятки несовместимых серверов приложений, и их поставщики придерживались совершенно разных парадигм. Некоторые из них даже частично реализовывались на «родных» языках, таких как C++. Разобраться с несколькими серверами было сложно, а переносить приложения с одного на другой практически не удавалось.

API-интерфейсы вроде JDBC (введены с JDK 1.1), JNDI (введены с JDK 1.3), JMS, JPA или сервлеты абстрагировали, упрощали и унифицировали уже созданные продукты. EJBs и CDI сделали модели развертывания и программирования независимыми от поставщика. J2EE (позже Java EE, а теперь Jakarta EE) и MicroProfile определили минимальный набор API, которые полагалось реализовать серверу приложений. После появления J2EE программист уже мог разрабатывать и развертывать приложения, если знал набор API-интерфейсов J2EE.

Несмотря на эволюцию серверов, API-интерфейсы J2EE и Java EE оставались совместимыми. Вам больше не требовалось переносить свое приложение для запуска на более новой версии сервера приложений. Даже обновление до более высокой версии Java EE проходило безболезненно. От вас требовалось лишь повторно протестировать приложение, даже не перекомпилируя его. Только в том случае, если вы хотели воспользоваться преимуществами новых API, вам следовало провести рефакторинг приложения. С появлением J2EE разработчики смогли освоить несколько серверов приложений, не слишком углубляясь в их специфику.

Сейчас у нас очень похожая ситуация в экосистеме web/JavaScript. Такие фреймворки, как jQuery, Backbone.js, AngularJS 1, Angular 2+ (полностью отличается от AngularJS 1), ReactJS, Polymer, Vue.js и Ember.js, следуют совершенно разным соглашениям и парадигмам. Стало трудно осваивать несколько фреймворков одновременно. Изначально цель многих фреймворков состояла в том, чтобы решить проблему несовместимости между различными браузерами. Когда браузеры стали удивительно совместимыми, фреймворки начали поддерживать привязку данных, однонаправленный

поток данных и даже корпоративные функции Java — например, внедрение зависимостей.

В то же время браузеры стали не только более совместимыми, но и предоставили функции, ранее возможные только со сторонними фреймворками. Функция `querySelector` включена во все браузеры и обеспечивает функциональность, сопоставимую с возможностями доступа к DOM в jQuery. Веб-компоненты с пользовательскими элементами, Shadow DOM и шаблонами позволяют разработчикам создавать новые элементы, содержащие пользовательский интерфейс и поведение, и даже структурировать целые приложения. Начиная с ECMAScript 6, JavaScript стал более похожим на Java, а модули ES6 сделали упаковку зависимостей в один файл необязательной. Веб-стандарты получили собственное хранилище в форме MDN (Mozilla Developer's Framework), результата объединенных усилий Google, Microsoft, Mozilla, W3C и Samsung.

Теперь можно даже создавать интерфейсы без применения фреймворков. У браузеров отличный послужной список в части обратной совместимости, но любые фреймворки все равно должны использовать API-интерфейсы браузера, поэтому, изучая стандарты, вы также лучше поймете фреймворки. Пока в браузеры не вносятся никакие критические изменения, просто опирайтесь на веб-стандарты без каких-либо фреймворков, и ваше приложение будет работать долго.

Сосредоточив внимание на стандартах, вы сумеете постепенно приобретать знания с течением времени, а это эффективный способ обучения. Исследовать популярные фреймворки увлекательно, однако полученные сведения не обязательно будут применимы к следующей «модной штучке».

Частые релизы снижают риск

Крис О'Делл

«Частые релизы снижают риск» — такую фразу вы постоянно слышите в разговорах о непрерывной поставке (CD). Как же так получается? Звучит нелогично. Несомненно, более частые релизы должны приводить к увеличению волатильности в производстве, так ведь? Разве не менее рискованно откладывать выпуск как можно дольше, неторопливо проводя тестирование, чтобы твердо убедиться в работоспособности продукта? Давайте обсудим, что мы подразумеваем под *риском*.

Что такое риск?

Риск — это фактор вероятности возникновения сбоя в сочетании с наихудшими последствиями этого сбоя:

Риск = вероятность возникновения сбоя × наихудшие последствия сбоя

Следовательно, деятельность с чрезвычайно низким уровнем риска — это когда вероятность сбоя критически мала, а его последствия незначительны. К видам деятельности с низким уровнем риска также относятся те, в которых один из этих факторов — вероятность или последствия — настолько мал, что серьезно снижает влияние другого.

Игра в лотерею сопряжена с низким риском: вероятность неудачи (т. е. невыигрыша) очень высока, но последствия неудачи (т. е. потери стоимости билета) минимальны, поэтому игра в лотерею не оборачивается крупными неприятностями.

Полеты также сопряжены с низким риском, но тут факторы сбалансированы противоположным образом. Вероятность отказа крайне мала — у авиарейсов очень хорошие показатели безопасности, — но последствия отказа чрезвычайно высоки. Мы часто летаем, так как считаем риск очень низким.

Деятельность с высоким риском характеризуется крупными значениями обоих факторов: высокой вероятностью сбоя и его серьезными последствиями.

К таким занятиям относятся экстремальные виды спорта — например, свободное одиночное скалолазание и дайвинг в пещерах.

Более рискованно выпускать редкие, но объемные релизы

Объединение набора изменений в одно обновление увеличивает вероятность возникновения сбоя, так как одновременно происходит большое количество изменений.

Наихудший вариант воздействия сбоя — сборка, из-за которой приложение перестает работать или происходит масштабная потеря данных. Любое изменение в релизе может привести к этому.

Естественная реакция здесь — пробы и тестирование каждого неудачного сценария. Это, конечно, разумно, но невозможно. Мы можем протестировать *известные* сценарии, но не такие, о которых не знаем, пока не столкнемся с ними («неизвестные неизвестные»).

Это не значит, что тестирование бессмысленно, — напротив, оно позволяет убедиться, что изменения не нарушили ожидаемого, известного поведения. Сложная часть заключается в том, чтобы найти баланс между стремлением к тщательному тестированию, вероятностью того, что тесты обнаружат сбой, и временем, затрачиваемым на их выполнение и обслуживание.

Создайте автоматизированный набор тестов, которые защищают от известных вам сценариев сбоев. Каждый раз, когда обнаруживается новый сбой, добавляйте его в набор тестов. Увеличьте свой набор регрессионных тестов, но сделайте их легкими, быстрыми и воспроизводимыми.

Независимо от того, сколько вы тестируете, важна только успешная проверка рабочей версии — это единственное место, где успех имеет значение. Небольшие, но частые релизы уменьшают вероятность сбоя. Если вы поместите в релиз наименьшее возможное изменение, то снизите вероятность того, что оно приведет к сбою.

Нет никакого способа уменьшить последствия сбоя — в худшем случае релиз по-прежнему может привести к отказу всей системы и серьезной потере данных, — но мы нивелируем общий риск, когда выпускаем небольшие релизы.

Вносите изменения небольшими порциями, чтобы снизить вероятность сбоя и, следовательно, уменьшить риск от изменений.

От головоломок к продуктам

Джессика Керр

Я занялась программированием, потому что это было легко. Весь день я решала головоломки, а в половине шестого возвращалась домой и тусовалась с друзьями. Двадцать лет спустя я остаюсь в сфере программного обеспечения, потому что это сложно.

Это сложно, потому что я перешла от разгадывания головоломок к созданию продуктов, от одержимости правильностью к поиску оптимальных решений.

В начале карьеры я сосредоточилась на одной области системы. Руководитель моей команды выдавал требования к новым функциям. Так определялась «правильность», и моя задача считалась выполненной, если код соответствовал запросам.

Доступные средства были ограничены: мы работали на C со стандартной библиотекой плюс Oracle. В качестве вишенки на торте, мы писали такой же код, как у всех остальных.

В течение нескольких лет мой кругозор расширился: я встречалась с заказчиками, участвовала в переговорах между группами проектировки и внедрения. Если какая-то конкретная новая функция уводила код в неудобном направлении, мы возвращались к заказчику с другими предложениями по решению той же проблемы. Теперь я помогаю составлять головоломки, а также решать их.

Решение головоломок — это обязательное условие, а не суть моей работы. Суть моей работы состоит в том, чтобы предоставить возможности остальной части организации (или всему миру). Я реализую это через управление полезным продуктом.

У головоломок есть конечное состояние, и цель решения головоломки заключается в том, чтобы достичь этого состояния. Здесь есть предопределенный финал, как в бейсбольном матче. В случае с продуктами цель состоит в том, чтобы сохранять полезность — как бейсболисты хотят продолжать карьеру после очередной игры.

У головоломок есть определенные возможности, как у настольной игры. Рост продуктов обеспечен целым миром библиотек и сервисов — множеством головоломок, решенных для нас. Это больше похоже на игру, где мы даем волю воображению, вбирая все, что можем найти.

На следующем этапе карьеры мой кругозор расширился еще заметнее.

Когда я создаю удовлетворительный код, моя работа только начинается. Мне хочется большего, чем просто изменить код: я стремлюсь к системным преобразованиям. Новая функция в моем приложении должна взаимодействовать с текущими системами, которые зависят от моей. Сотрудничая с владельцами этих систем, я помогаю им начать использовать новую функцию.

Теперь я вижу свое занятие как разработку изменений, а не кода. Код — это деталь.

Разработка изменений подразумевает описание функционала, обратную совместимость, миграцию данных и постепенное развертывание. Она означает документацию, полезные сообщения об ошибках и социальные контакты со смежными командами.

Еще плюс: как же все эти операторы `if` для функций, устаревших методов и обработки обратной совместимости? Они больше не уродливы. Они выражают изменение. И вся суть кроется в изменениях, а не в каком-то конкретном состоянии кода.

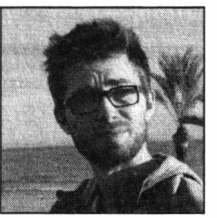
При проектировании нужно добавлять наблюдаемость, чтобы я могла определить, кто все еще применяет устаревшую функцию, а кто уже успешно использует новую. Когда я решала головоломки, мне не требовалось заботиться о том, нравится ли людям эта функция или вообще до сих пор находится в разработке, тогда как рост продукта меня очень волнует. Получая опыт производства, мы узнаем, как сделать свои продукты более полезными.

У продуктов нет единого определения «правильности». Часто встречается то, что явно неправильно, поэтому нам нужно осторожнее обращаться с определением «не сломано». Помимо того, мы стремимся к «улучшению».

Сложно обеспечить рост продукта иным способом, нежели решение головоломок. Вместо тяжелой работы, за которую нас поощряет чувство выполненного долга, мы упорно бредем по трясине двусмысленности, политических махинаций и сопутствующих обстоятельств. Однако награда здесь — нечто большее, чем просто ощущение того, что мы справились с задачей. Ваш труд может оказать осязаемое влияние на вашу компанию и, следовательно, на весь мир. Это приносит больше удовольствия, чем обычное развлечение.

«Разработчик полного цикла» — это образ мышления

Мацей Валковяк



В 2007 году, когда началась моя карьера в качестве программиста на Java, в повседневной веб-разработке использовался довольно узкий диапазон технологий. В большинстве случаев из всех типов баз данных сотруднику требовалось знать лишь реляционные БД. Для создания интерфейса применялись только HTML и CSS, приправленные щепоткой JavaScript. Сама разработка на Java подразумевала в первую очередь взаимодействие с любой библиотекой Hibernate: Spring или Struts. Этот набор технологий охватывал почти все, что в то время требовалось для написания приложений. Большинство разработчиков Java на самом деле представляли собой *разработчиков полного цикла*, хотя этот термин тогда еще не придумали.

С 2007 года ситуация значительно изменилась. Мы начали создавать все более и более сложные пользовательские интерфейсы и справлялись с этой сложностью, полагаясь на продвинутые фреймворки JavaScript. Сейчас мы используем базы данных NoSQL, и почти каждая из них значительно отличается от других. Мы передаем данные с помощью Kafka, отправляем сообщения с помощью RabbitMQ и делаем многое другое. Очень часто мы также отвечаем за настройку или обслуживание инфраструктуры с помощью Terraform или CloudFormation, а еще применяем или даже настраиваем кластеры Kubernetes. Общая сложность систем выросла до такой степени, что теперь есть отдельные должности фронтенд-разработчика, бэкенд-разработчика и DevOps-инженера. Возможно ли по-прежнему быть разработчиком полного цикла? Зависит от того, как понимать этот термин.

Вы не можете быть экспертом во всем. Учитывая, насколько разрослась экосистема Java, трудно быть экспертом даже в самой Java. Хорошо то, что *не обязательно* быть им. Для многих проектов, особенно в небольших компаниях, наиболее выгодная структура команды выглядит так: каждой областью знаний занимается по крайней мере один эксперт, но эти сотрудники

не ограничиваются работой только в данной области. Программисты, специализирующиеся на создании сервисов для бэкенда, могут писать интерфейсный код — пусть не идеальный, — и то же самое относится к фронтенд-разработчикам. Это помогает быстрее продвигать проекты, поскольку один человек может создать изменение, затрагивающее каждый слой приложения. Кроме того, в такой ситуации увеличивается вовлеченность в ходе обсуждений по задачам, поскольку не требуется обсуждать такие, что предназначены только для определенной группы людей.

Самое главное, что в отсутствие строгого ограничения одной областью меняется ваш подход к выполнению задач. Больше нет дискуссий на тему «Я не обязан это делать» — разработчикам рекомендуется учиться. Когда кто-то уходит в отпуск, проблемы больше не возникают, потому что всегда найдутся коллеги, способные его заменить (возможно, не так эффективно, но достаточно умело, чтобы продолжить разработку проекта, пусть и, вероятно, не с такими хорошими результатами). Отсюда также следует, что, когда возникает необходимость внедрить новую технологию в стек, вам не нужно искать нового сотрудника, потому что нынешние члены команды уже отлично чувствуют себя вне зоны комфорта своих знаний.

Следовательно, *разработчик полного цикла* — это образ мышления. Он начальник и подчиненный одновременно, он исполнителен и деятелен.

Сборщик мусора — ваш друг

Холли Камминс

Бедный старый сборщик мусора. Один из невоспетых героев Java, которого часто обвиняют, но редко хвалят. До того как Java популяризовала сборку мусора, программистам оставалось только отслеживать всю память, которую они выделили вручную, и освобождать ее, как только прекращалось ее использование. Это тяжело. Даже при соблюдении дисциплины ручное очищение часто становится причиной утечек памяти (если проводится слишком поздно) и сбоев (если проводится слишком рано).

Java GC (garbage collection — сборка мусора) часто рассматривается как необходимое зло, и среди советов по производительности нередко звучит «сокращайте время, затрачиваемое на GC». Однако современная сборка мусора может оказаться быстрее, чем `malloc/free`, и если вы проведете какое-то время в GC, то сумеете ускорить работу приложения в целом. Почему? Сборщики мусора не просто освобождают память: они также обрабатывают выделение памяти и расположение объектов в ней. Благодаря хорошему алгоритму управления памятью можно добиться эффективности распределения, уменьшив фрагментацию и количество конфликтов. Также удается повысить пропускную способность и сократить время отклика посредством перестановки объектов.

Почему расположение объекта в памяти влияет на производительность приложения? При выполнении программы значительная часть времени тратится на аппаратную остановку в ожидании доступа к памяти. По сравнению с обработкой команд доступ к куче медленный, как движение континентов, поэтому современные компьютеры используют кэши. Когда объект извлекается в кэш процессора, его соседи следуют за ним. Если к ним будет инициирована следующая попытка доступа, он окажется быстрым. Наличие объектов, которые используются одновременно рядом друг с другом в памяти, называется *локальностью объектов*, и это выигрыш в производительности.

Преимущества эффективного распределения более очевидны. Если куча фрагментирована, когда программа пытается создать объект, ей придется долго искать достаточно большой кусок свободной памяти и выделение

станет затратным. В качестве эксперимента вы можете заставить GC больше сжимать данные. Это значительно увеличивает накладные расходы на GC, но зачастую производительность приложений улучшается.

Стратегии GC варьируются в зависимости от реализации JVM, и каждая виртуальная машина предлагает ряд настраиваемых опций. Значения по умолчанию в JVM обычно хороши для начала, но стоит разобраться в некоторых механизмах и возможных вариациях. Можно повысить пропускную способность ценой увеличения задержки — в итоге на оптимальный выбор влияет рабочая нагрузка.

Сборщики stop-the-world останавливают все действия программы, чтобы иметь возможность безопасно собирать данные. Параллельные сборщики разгружают работу по сбору данных потокам приложений, поэтому глобальные паузы не возникают. Вместо них каждый поток испытывает крошечные задержки. Хотя у них нет очевидных задержек, параллельные сборщики менее эффективны, чем сборщики stop-the-world, поэтому они хороши для приложений, где паузы заметны пользователю (например, воспроизведение музыки или графический интерфейс).

Сам сбор осуществляется путем копирования или путем разметки и сборки мусора. С помощью «разметки и сборки» выполняется обход кучи для определения свободного места, и в найденных промежутках затем размещаются новые объекты. Копирующие сборщики делят кучу на две зоны. Объекты размещаются в «новой области». Когда эта зона заполнена, ее содержимое, не относящееся к мусору, копируется в резервное пространство, а пробелы меняются местами. При типичной рабочей нагрузке большинство объектов умирают молодыми (этот процесс называют «гипотезой поколений»). С недолговечными объектами шаг копирования будет исключительно быстрым (ведь копировать нечего!). Однако, если объекты станут околачиваться в памяти, сбор мусора станет неэффективным. Копирующие сборщики отлично подходят для неизменяемых объектов, но приводят к катастрофам при «оптимизации» объединения объектов (в любом случае обычно это плохая идея). В качестве бонуса копирующие сборщики уплотняют кучу, что обеспечивает почти мгновенное выделение объектов и быстрый доступ к ним (меньше пропусков в кэше).

При оценке эффективности нужно учитывать бизнес-значения. Оптимизируйте количество транзакций в секунду, среднее время обслуживания или наихудшую задержку. Но не пытайтесь заниматься микрооптимизацией времени, затрачиваемого на GC, потому что, уделив внимание сборке мусора, вы действительно можете ускорить программу.

Называйте вещи своими именами

Питер Хилтон

Но самое главное — пусть смысл выбирает слова, а не наоборот. Самое худшее, что можно сделать со словами в прозе, — это сдаться на их милость.*

Джордж Оруэлл

Умение давать удачные названия несравненно упрощает поддержку кода, который вы пишете. Хотя их одних недостаточно для создания хорошо поддерживаемого кода, правильно подбирать слова, как известно, сложно, и этим обычно пренебрегают. К счастью, программисты любят непростые задачи.

Во-первых, избегайте названий следующих типов: бессмысленных (`foo`), слишком абстрактных (`data`), дублированных (`data2`), расплывчатых (`DataManager`), усеченных или коротких (`dat`). Хуже всего одиночные буквы (`d`). Такие имена неоднозначны, и это замедляет работу всего коллектива, поскольку программисты тратят больше времени на чтение кода, чем на его написание.

Затем примите рекомендации по улучшению названий — выбирайте термины с точным значением, которые четко описывают смысл кода.

Используйте до четырех слов для каждого имени и не прибегайте к сокращениям (за исключением `id` и тех, которые вы заимствуете из проблемной области). Одного термина редко бывает достаточно. Больше четырех — громоздко, теряется смысл. Java-программисты используют длинные имена классов, но часто предпочитают короткие имена локальных переменных, даже если такие термины выглядят хуже.

Изучайте и используйте терминологию проблемной области — *вездесущий словарь* предметно-ориентированного проектирования. Он часто бывает исчерпывающим: так, в издательском деле правка текста верно называется как *revision* (переработка), так и *edit* (редактирование), в зависимости от того, кто вносит изменения. Вместо того чтобы придумывать слова, прочитайте нужную страницу

* Перевод В. Голышева.

в «Википедии», поговорите с людьми, которые работают в интересующей вас области, и пополните ваш глоссарий терминами, что звучат в их речи.

Замените множественное число собирательными существительными (например, переименуйте `appointment_list` в `calendar`). В общем и целом расширяйте ваш словарный запас английского языка, чтобы находить более короткие и точные имена. Это сложнее, если английский для вас неродной, но любой разработчик все равно обязан выучить жаргон предметной области.

Переименовывайте пары объектов с названиями отношений (например, вместо `company_person` пишите `employee`, `owner`, `shareholder`). Если речь о поле, обозначайте связь между типом поля и классом, к которому оно принадлежит. В общем, часто бывает ценным извлечь новую переменную, метод или класс только для того, чтобы получить возможность явно дать им имя.

Java помогает вам с правильным именованием, потому что вы называете классы отдельно от объектов. Но не забывайте самостоятельно обозначать свои типы вместо того, чтобы полагаться на примитивы и классы JDK: взамен `String` обычно следует вводить класс с более конкретным именем, например `CustomerName`. В противном случае вам понадобятся комментарии для документирования неприемлемых строк — в частности пустых.

Не путайте класс и имена объектов: переименуйте поле даты с названием `dateCreated` в `created` и логическое поле с названием `isValid` в `valid`, чтобы не дублировать типы в терминах. Присваивайте объектам разные обозначения: вместо `Customer` с названием `customer` выберите более конкретное слово, например `recipient` (получатель) при отправке уведомлений или `reviewer` (обозреватель) при публикации отзыва о продукте.

Первым шагом в раздаче названий — установить базовые правила именования, в частности, применение фраз из существительных для имен классов. Следующий шаг — соблюдать хорошую доктрину именования с использованием подобных рекомендаций. Но у руководящих принципов есть пределы. Спецификация JavaBeans научила поколение Java-программистов нарушать инкапсуляцию объектов и применять неопределенные имена методов — в частности, `setRating`, когда `rate`, возможно, лучше. Вам не нужно называть необязательные методы глагольными фразами, как в сборщиках API-интерфейсов — например, `Customer.instance().rating(FIVE_STARS).active()`. В конце концов, мастерство именования заключается в выборе того, какие правила нарушать.

Эй, Фред, не мог бы ты передать мне HashMap?

Кирк Пеппердайн

Представьте себе такую сцену: тесный обшарпанный офис с несколькими старыми деревянными столами, составленными впритык. На каждом из них — старинный дисковый телефон черного цвета, вокруг множество пепельниц. На одном из столов находится черный класс `HashMap`, содержащий `ArrayList`, заполненный данными клиентов. Сэм, которому нужно связаться с компанией «Рога и копыта», осматривает офис в поисках `HashMap`. Пробегая глазами по помещению, он замечает `HashMap` и кричит: «Эй, Фред, не мог бы ты, пожалуйста, передать мне `HashMap`?». Можете представить такую картину?.. Ага, я и не думал...

Важная часть написания программы заключается в развитии словарного запаса. Каждый термин в глоссарии обязан соответствовать некоему элементу области, которую мы моделируем. В конечном счете именно это кодовое выражение нашей модели должны будут прочитать и уяснить другие люди. Следовательно, сформированный нами лексикон может либо помочь, либо помешать пониманию кода. Как ни странно, выбор лексики влияет отнюдь не на одну лишь удобочитаемость: слова, которые мы используем, воздействуют на то, как мы думаем о рассматриваемой проблеме, что, в свою очередь, влияет на структуру нашего кода, подбор алгоритмов, то, как мы формируем свои API, насколько хорошо система будет соответствовать поставленным целям, как легко ее будет поддерживать и расширять и, наконец, насколько успешно она станет работать. Да, словарный запас, который мы развиваем при написании кода, имеет большое значение. Настолько, что при написании кода иногда удивительно полезно держать под рукой глоссарий.

Возвращаясь к нелепому примеру, конечно, никто не стал бы просить о `HashMap`. Скорее всего, вы бы встретили непонимающий взгляд Фреда, если бы попросили его передать `HashMap`. Тем не менее когда мы разбираемся, как моделировать домен, то слышим о необходимости поиска контактных данных клиентов, упорядоченных по имени. Это кричит `HashMap`. Если углубиться в предметную область, то, скорее всего, можно заметить, что контактная

информация записана на карточке, которая аккуратно упакована в Rolodex. Замена терминов, HashMap на Rolodex, не только улучшает абстракцию в нашем коде, но и непосредственное влияет на то, как мы думаем о рассматриваемой проблеме, и для нас это лучший способ выразить свои мысли читателю нашего кода.

Вывод здесь такой: техническим классам редко отводится место в словаре предметных областей, с которыми мы работаем. Вместо этого они предоставляют нам строительные блоки для более глубоких, более значимых абстракций. Необходимость в служебных классах — сигнал об опасности, указывающий, что вам не хватает абстракции. Так же должны рассматриваться и технические классы в API.

Например, проанализируем случай, когда сигнатура метода принимает String для представления имени и String для фамилии. Они используются для поиска данных, хранящихся в HashMap:

```
return listOfNames.get(firstName + lastName);
```

Вопрос в том, где здесь недостающая абстракция. Наличие двух полей, образующих ключ, обычно известно как составной ключ. Используя эту абстракцию, мы получаем:

```
return listOfNames.get(new CompositeKey(firstName, lastName));
```

Когда вы вносите данное изменение в бенчмарк, код выполняется в три раза быстрее. Я бы сказал, что это также более выразительно: использование CompositeKey лучше выражает суть рассматриваемой проблемы.

Как избежать Null

Карлос Обрегон

Тони Хоар называет `null` «ошибкой на миллиард долларов». Это действительно ошибка, так что вам нужно выработать привычку запрещать коду использовать `null`. Если у вас есть ссылка на объект, который может иметь значение `null`, обязательно выполните проверку на `null`, прежде чем пытаться вызвать какой-либо его метод. Но поскольку нет никакой очевидной разницы между ссылкой на `null` и не-`null`, это слишком легко забыть — и получить в итоге `NullPointerException`.

Самый надежный способ предотвратить проблемы в будущем — использовать альтернативы, когда это возможно.

Избегайте инициализации переменных в значение Null

Обычно не рекомендуется объявлять переменную, пока вы не узнаете, какое значение она должна содержать. Для сложной инициализации переместите всю логику инициализации в метод. Например, вместо того чтобы делать так:

```
public String getEllipsifiedPageSummary(Path path) {
   String summary = null;
   Resource resource = this.resolver.resolve(path);
   if (resource.exists()) {
      ValueMap properties = resource.getProperties();
      summary = properties.get("summary");
   } else {
      summary = "";
   }
   return ellipsify(summary);
}
```

поступите так:

```
public String getEllipsifiedPageSummary(Path path) {
   var summary = getPageSummary(path);
   return ellipsify(summary);
}

public String getPageSummary(Path path) {
   var resource = this.resolver.resolve(path);
   if (!resource.exists()) {
      return "";
   }
   var properties = resource.getProperties();
   return properties.get("summary");
}
```

Инициализация переменной значением `null` может привести к непреднамеренному появлению `null`, если вы не будете осторожны с обработкой ошибок. Другой разработчик может изменить поток управления, не осознавая проблемы, причем этим другим разработчиком вполне способны оказаться и вы — через три месяца после того, как впервые написали код.

Избегайте возврата Null

Когда вы читаете сигнатуру метода, то должны разобраться, всегда ли он возвращает объект `T` или порой это не так. Возврат `Optional<T>` — наилучший вариант, делающий код более явным. API `Optional` очень упрощает работу со сценарием, в котором не было создано `T`.

Избегайте передачи и получения Null-параметров

Если вам нужен объект `T`, запросите его. Если получится обойтись без него, тогда не просите. Для операции, которая может иметь необязательный параметр, создайте два метода: один с параметром и один без него.

Например, у метода `drawImage` из класса `Graphics` в JDK есть версия, которая получает пять параметров и шестой, необязательный, — `ImageObserver`.

Если у вас нет `ImageObserver`, следует передать `null` таким образом:

```
g.drawImage(original, X_COORD, Y_COORD, IMG_WIDTH, IMG_HEIGHT, null);
```

А лучше задействовать другой метод только с первыми пятью параметрами.

Допустимые Null

В каких же случаях приемлемо использовать `null`? Как деталь реализации класса, т. е. значение атрибута. Код, который должен знать об этом отсутствии значения, содержится в том же файле, и гораздо проще помнить об этом и не допускать появления `null`.

Поэтому не забывайте: если у вас нет атрибута, всегда можно уйти от применения `null`, применив улучшенную конструкцию в коде. Если вы прекратите использование `null` в тех местах, где этого не требуется, то утечка `null` и получение `NullPointerException` станут невозможными. И если вы избежите этих исключений, то поможете решить проблему на миллиард долларов вместо того, чтобы усугублять ее.

Как вывести из строя JVM

Томас Ронзон

Существует так много новых API, классных библиотек и обязательных к использованию методов, которые вам нужно знать, что порой трудно оставаться в курсе последних событий.

Но действительно ли это все, в чем вам следует разбираться как Java-программисту? А как насчет среды, в которой работает ваше ПО? Не произойдет ли так, что проблема в ней приведет к сбою вашего программного обеспечения и вы даже не сумеете понять или найти эту проблему, потому что она находится за пределами экосистемы библиотек и кода? Готовы ли вы посмотреть на такую ситуацию с другой стороны?

Вот задача: попробуйте найти способы аварийного завершения работы вашей виртуальной машины JVM! (Или, по крайней мере, привести ее нормальное выполнение к внезапной и неожиданной остановке.)

Чем больше способов вы знаете, тем лучше вы разбираетесь в окружении и понимаете, какие загвоздки могут возникнуть с работающей системой ПО. Вот несколько методов, которые помогут вам начать:

1. Постарайтесь выделить как можно больше памяти. Оперативная память не бесконечна — если ОЗУ больше невозможно выделять, ваш процесс завершится неудачей.

2. Попробуйте записывать данные на свой жесткий диск, пока он не заполнится. Та же проблема, что и с оперативной памятью: хотя объем постоянной памяти больше ОЗУ, дисковое пространство тоже не безгранично.

3. Постарайтесь открыть как можно больше файлов. Известно ли вам максимальное количество файловых дескрипторов для вашей среды?

4. Постарайтесь создать как можно больше потоков. В системе Linux если вы посмотрите на `/proc/sys/kernel/pid_max`, то увидите, сколько процессов можно запустить в вашей системе. Как много потоков вам разрешено создавать в своей системе?

5. Попробуйте изменить ваши файлы *.class* в файловой системе, и текущий запуск вашего приложения будет последним!

6. Попробуйте найти собственный идентификатор процесса, а затем попытайтесь уничтожить его с помощью `Runtime.exec` (например, вызвав `kill -9` для вашего идентификатора процесса).

7. Попробуйте создать во время выполнения класс, который вызывает только `System.exit`, динамически загрузите его через загрузчик классов, а затем вызовите его.

8. Постарайтесь открыть как можно больше подключений к сокетам. В системе Unix максимальное количество таких подключений равняется максимальному количеству файловых дескрипторов (обычно 2048). Сколько из них доступно там, где запущено ваше приложение?

9. Попробуйте взломать свою систему. Загрузите эксплойт с помощью кода или `wget`. Запустите эксплойт, а затем вызовите `shutdown -h` от имени root в системе Unix или `shutdown /s` от имени администратора в системе Windows.

10. Попробуйте «прыгнуть без страховочной сетки». Частично безопасность Java обеспечивается ее языковым дизайном, а частично — проверкой байт-кода в вашей JVM. Запустите свою JVM с флагом `-noverify` или `-Xverify:none`, которые отключают любую проверку байт-кода и записывают то, что в противном случае было бы запрещено запускать.

11. Попробуйте использовать `Unsafe`. Этот класс backdoor используется для получения доступа к низкоуровневым средствам, таким как управление памятью. Вот вам и синтаксис Java, вот вам и безопасность C!

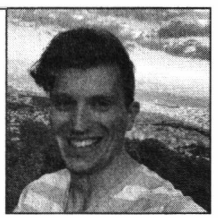
12. Попробуйте пройти нативным путем. Напишите какой-нибудь машинный код. Вот вам и синтаксис C, вот вам и безопасность C!

Попробуйте найти собственные методы генерации сбоя вашей JVM и попросите коллег поделиться своими идеями. Также подумайте о том, чтобы спросить кандидатов на собеседовании о том, как бы они это сделали. Какой бы ответ вы ни услышали, вам скоро станет ясно, способен ли интервьюируемый видеть мир за пределами своего окна IDE.

P. S. Если вы найдете другие креативные способы генерации сбоя JVM, пожалуйста, дайте мне знать!

Улучшение повторяемости и контролируемости посредством непрерывной поставки

Билли Корандо



Изделия ручной работы ценятся из-за затраченного времени и усилий, а также из-за небольших изъянов, которые придают им характерные особенности и уникальность. Подобные качества ценны в еде, мебели или предметах искусства, однако, если речь идет о предоставлении кода, они становятся серьезными препятствиями для успеха организации.

Люди по природе своей не очень хорошо справляются с повторяющимися задачами. Независимо от того, насколько человек ориентирован на детали, он допускает ошибки при выполнении ряда сложных шагов, необходимых для развертывания приложения. Возможно, он пропустит шаг, осуществит его в неправильной среде или выполнит некорректно иным образом, что приведет к сбою развертывания.

Когда происходят сбои в развертывании, на выяснение причин тратится весьма много времени. Ход расследования труден, поскольку ручные процессы часто не имеют центральной точки контроля и выглядят довольно непрозрачно. Когда основная причина определена, в качестве решения, как правило, добавляют дополнительные уровни контроля, чтобы предотвратить повторение проблемы, но обычно процесс развертывания лишь становится более сложным и мучительным!

Поскольку трудности с поставкой кода в организациях — давно не новость, для решения этой проблемы они начали переходить на непрерывную поставку (CD). CD — это подход к автоматизации этапов поставки кода в производство. С момента, когда разработчик фиксирует изменение, до момента развертывания этого изменения в рабочей среде любой шаг, который можно автоматизировать, нужно автоматизировать, будь то тестирование, контроль изменений, процесс развертывания и т. д.

Ключевая мотивация для перехода на CD — сокращение времени и усилий, требуемых для развертывания кода. Но значительные преимущества непрерывной поставки этим не ограничиваются! CD также улучшает повторяемость и контролируемость процесса развертывания. Вот несколько причин, по которым вы должны заботиться об этих качествах.

Повторяемость

Автоматизация шагов по развертыванию кода означает написание скриптов для каждого шага, чтобы его выполнял компьютер, а не человек. Это значительно улучшает повторяемость процесса развертывания, поскольку компьютеры превосходно решают повторяющиеся задачи.

Повторяющийся процесс по сути своей менее рискован, что может побудить организации выпускать версии чаще и с меньшими наборами изменений. Так у них появятся дополнительные преимущества — возможность ориентировать выпуск на устранение конкретных проблем (например, с производительностью). Релиз может содержать только изменения, направленные на производительность, что позволит оценить, улучшилась она в результате изменений, ухудшилась или осталась прежней.

Контролируемость

Автоматизация развертывания значительно повышает прозрачность процесса, что, естественно, дает дополнительные возможности проверять его. Скрипты, используемые для выполнения шагов, и значения, предоставленные им, могут храниться в системе управления версиями, что позволяет легко просматривать их. Автоматизированные развертывания также способны создавать отчеты, помогающие осуществлять проверку. Благодаря улучшенной контролируемости процесса развертывания CD превращается из нишевой концепции для стартапов и некритичных приложений в незаменимое подспорье даже в самых жестко регулируемых и регламентируемых отраслях.

Когда я впервые услышал о CD, концепция развертывания по требованию показалась мне опьяняющей. Прочитав *Continuous Delivery* Джез Хамбл и Дэвида Фарли (издательство Addison-Wesley), я узнал, что сокращение времени и усилий во многом вторично по сравнению с повторяемостью и контролируемостью, которые предлагает CD. Если у вашей организации возникают трудности с внедрением кода в производство, я надеюсь, что этот текст поможет вам обосновать руководству, почему следует перейти на CD.

В языковых войнах Java может за себя постоять

Дженнифер Райф

Мы все выбираем себе любимчиков и принижаем другие варианты (будь то цвета, автомобили, спортивные команды и так далее). Выбор языка программирования здесь не исключение. Возможно, выбранный ЯП для нас удобнее всего или благодаря ему мы устроились на работу. Так или иначе, мы цепляемся за него.

Сегодня сосредоточимся на Java. Есть вполне обоснованные жалобы и похвалы в адрес данного языка. Дальнейший текст основан на моем личном опыте, и другие люди могут иметь иную точку зрения.

Моя история знакомства с Java

Сначала давайте поглядим, через какую призму я рассматриваю этот ЯП.

С программированием приложений я познакомилась в колледже, где использовала — угадайте, что? — Java. До этого я прошла несколько вводных занятий по HTML, Alice и Visual Basic. Ни один из этих языков не разрабатывался для погружения в сложные структуры кода.

Итак, мой первый опыт знакомства с программированием для корпоративных сред и критически важных процессов связан с Java. С тех пор я работала со многими другими языками, но по-прежнему возвращаюсь к Java.

Замысел и предыстория Java

Язык Java, созданный в 1995 году, имеет C-подобный синтаксис и соответствует принципу WORA (написать один раз, запускать где угодно). Его задача состояла в том, чтобы упростить сложное программирование, необходимое для языков семейства C, и добиться независимости от платформы с помощью JVM.

Я думаю, что знание истории ЯП поможет рассмотреть его положительные и отрицательные стороны в контексте. Если изучить, как развивался язык, станет понятно, чем пожертвовали его создатели ради достижения других целей.

Недостатки Java

В основном жалобы сводятся к тому, что развертываемые файлы большие, а синтаксис многословен. Хотя это действительно так, я полагаю, что предыдущий абзац об истории Java объясняет, в чем причина.

Во-первых, дистрибутивы Java в целом больше. Как мы знаем из истории Java, язык создали по концепции «написать один раз, запускать где угодно», чтобы приложение могло запускаться на любой JVM без изменений. Это означает, что все зависимости следует добавлять в схему развертывания, независимо от того, объединены ли они в один JAR или распределены между различными компонентами (WAR файл + сервер приложений + JRE + зависимости). Это влияет на размер дистрибутива.

Во-вторых, язык Java очень многословен. Опять же, я считаю, что так задумывалось. Его создавали, когда C и подобные ему языки управляли памятью, поэтому разработчикам требовалось указывать низкоуровневые детали.

Цель Java состояла в том, чтобы абстрагироваться от некоторых из этих деталей и стать более удобной для пользователя.

Почему мне нравится Java

- Java говорит мне, что я создаю и как. С другими языками я, возможно, смогу написать что-нибудь, затратив меньшее количество строк, но мне не нравится, что у меня нет четкого понимания того, как эти ЯП работают «под капотом».
- Этот навык пригодится где угодно. Работая с Java в различных областях, я приобрела знания, относящиеся как к бизнесу, так и к техническому рынку. Java — не единственный язык, обладающий таким преимуществом, но среди всех аналогов он выглядит наиболее долговечным.
- Java позволяет мне играть с технологиями во всех стеках и областях. Кажется, он объединяет их. Мне нравится экспериментировать и исследовать, и в Java это возможно.

Что это значит для разработчиков?

Рынок разнообразен, на нем множество предложений, отвечающих потребностям бизнеса. Один и тот же вариант не годится для всех (да и не должен), поэтому каждый разработчик обязан выбирать ЯП, лучше всего подходящий для конкретной задачи. Даже если вам не нравится Java как основной язык, я все равно считаю, что владение им — ценный навык.

Встроенное мышление

Патриция Аас

Компьютеры изменились. Они преобразились во многих отношениях, но в рамках этой статьи для нас важен один существенный сдвиг: относительные затраты чтения из оперативной памяти стали чрезвычайно высокими.

Это происходило постепенно, пока доступ к ОЗУ не стал полностью доминировать в показателях производительности приложения. Центральный процессор постоянно ожидал завершения доступа к памяти. И по мере того как затраты на доступ к ОЗУ по сравнению с регистрами росли и росли, производители чипов вводили всё новые и новые уровни кэша и делали их всё больше и больше.

А кэш — это здорово! Если то, что вам нужно, расположено в нем…

Кэши многоплановы, но, как правило, они предсказывают, что последующий доступ к памяти будет близок или приблизительно схож с недавним предыдущим доступом. Это достигается путем извлечения из памяти немного большего объема, чем необходимо, и сохранения этого избытка в кэше, что часто называется предварительной выборкой. Если более поздний доступ может получить свое значение из кэша вместо оперативной памяти, это обозначается как доступ, «дружественный к кэшу».

Представьте, что вам нужно выполнить итерацию по большому массиву относительно небольших объектов, возможно, по куче треугольников. Вообще говоря, сегодня в Java нет массива треугольников. У вас есть массив указателей на объекты «треугольник», потому что обычные объекты в Java относятся к «ссылочным типам». Это означает, что вы получаете к ним доступ через указатели/ссылки Java. Таким образом, даже несмотря на то, что массив, вероятно, представляет собой непрерывный раздел памяти, сами объекты «треугольники» могут находиться в любом месте кучи Java. Цикл по этому массиву окажется «недружественным к кэшу», поскольку мы будем перескакивать в памяти от одного объекта «треугольник» к другому, и предварительная выборка кэша, вероятно, нам не сильно поможет.

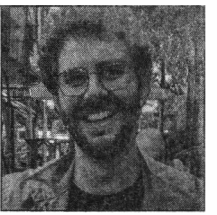
Но представьте, что массив содержит фактические объекты «треугольник», а не указатели на них. Теперь они соседствуют в памяти, и перебор по ним гораздо более «дружественен к кэшу». Допустимо, что следующий треугольник будет ждать нас прямо там, в кэше. Типы объектов, которые можно сохранять непосредственно в подобном массиве, называются «типами значений» или «встроенными типами». В Java уже есть несколько встроенных типов, например `int` и `char`, и скоро появятся определяемые пользователем типы. Вероятно, их назовут «встроенными классами». Они будут похожи на обычные классы, только проще.

Другой способ достичь дружественности к кэшу — хранить объекты в вашей области стека или непосредственно в регистрах. Разница между встроенными типами и ссылочными типами заключается в том, что вам не нужно хранить встроенные типы в куче. Это полезно для объектов, которые существуют только для области действия вызова метода. Поскольку соответствующие части стека, вероятно, находятся в кэше, доступ к объектам в стеке, как правило, будет дружественным к кэшу. Еще одно преимущество — объекты, которые не хранятся в куче Java, не нуждаются в сборе мусора.

Это дружественное к кэшу поведение уже наблюдается в Java при использовании так называемых «примитивных типов» — например, `int` и `char`. Примитивные типы, будучи встроенными типами, обладают всеми их преимуществами. Следовательно, хотя встроенные типы поначалу могут показаться чуждыми, вы уже работали с ними раньше. Возможно, вы просто не думали о них как об объектах. Итак, когда «встроенные классы» покажутся запутанными, не спросить ли вам себя: «Могу ли я использовать здесь `int`?»

Взаимодействие с Kotlin

Себастьяно Поджи



В последние годы Kotlin стал горячей темой в сообществе JVM. Область применения языка постоянно расширяется — от мобильных до серверных проектов. Одно из преимуществ Kotlin — его высокая степень совместимости с Java, не требующая дополнительных усилий.

Вызов любого Java-кода из Kotlin работает без капризов. Kotlin прекрасно понимает Java, но есть одна маленькая неприятность, порой возникающая, если вы не следуете рекомендациям Java в точности: отсутствие типов, которые не могут принимать значение `null` в Java. Если вы не применяете аннотации в Java, явно указывающие, могут ли данные типы быть `null`, Kotlin предполагает, что для всех данных типов неизвестно, способны ли они стать `null`, — это так называемые платформенные типы.

Если вы уверены, что они никогда не будут `null`, то можете принудительно преобразовать их в тип, отличный от `null`, с помощью оператора «`!!`» или путем приведения их к типу, отличному от `null`. В любом случае вы получите сбой, если во время выполнения значение станет равно `null`. Лучший способ избежать такого сценария — добавить в ваши Java-API аннотации, которые явно указывают, может ли этот тип быть `null` (например, `@Nullable` и `@NotNull`). Существует множество поддерживаемых аннотаций (*https://oreil.ly/hKoXx*) содержащихся в: JetBrains, Android, JSR-305, FindBugs и в других модулях. Таким образом, Kotlin будет знать, способен ли тип стать `null`, и при написании кода на Java вы получите дополнительную информацию и предупреждения от IDE о потенциальных `null`. Беспроигрышный вариант!

При вызове кода Kotlin из Java вы заметите, что, хотя большая часть кода работает просто отлично, возникают причуды с некоторыми расширенными функциями языка Kotlin, которые не имеют прямого эквивалента в Java. Компилятор Kotlin должен принять некоторые творческие решения, чтобы реализовать их в байт-коде.

Они скрыты в Kotlin, но Java не знает об этих механизмах и раскрывает их, что порождает приемлемый, но не оптимальный API.

Примером могут служить объявления верхнего уровня. Поскольку байт-код JVM не поддерживает методы и поля вне классов, компилятор Kotlin помещает их в синтетический класс с тем же именем, что и файл, в котором они находятся. Например, все символы верхнего уровня в файле *FluxCapacitor.kt* будут отображаться как статические члены класса `FluxCapacitorKt` из Java. Вы можете изменить имя сгенерированного класса на что-то более приятное, снабдив файл Kotlin комментарием `@file:JvmName("Flux CapacitorFuncs")`.

Возможно, вы ожидаете, что элементы, определенные в объекте (`companion`), будут статическими в байт-коде, но это не так. Kotlin «под капотом» перемещает их в поле с именем `INSTANCE` или искусственный внутренний класс `Companion`. Если вам нужно получить к ним доступ как к статическим членам, просто прокомментируйте их с помощью `@JvmStatic`. Кроме того, вы можете сделать так, чтобы свойства объекта (`companion`) отображались в виде полей в Java, аннотируя их как `@JvmField`.

Наконец, Kotlin предлагает необязательные параметры со значениями по умолчанию. Это очень удобная функция, но, к сожалению, Java ее не поддерживает. В Java вам необходимо указать значения для всех параметров, включая необязательные. Чтобы избежать этого, используйте аннотацию `@JvmOverloads`, которая поручает компилятору генерировать перегрузки для всех необязательных параметров. Порядок параметров важен, поскольку здесь в перегрузке не происходит всех возможных изменений параметров, а совершается одна дополнительная перегрузка для каждого необязательного параметра в том порядке, в котором они отображаются в Kotlin.

Подводя итог, можно сказать, что Kotlin и Java почти полностью совместимы «из коробки», и это одно из преимуществ Kotlin перед другими языками JVM. Кроме того, в некоторых случаях, если вы минутку поработаете над вашими API, их использование станет намного более приятным и в другой среде. Честное слово, нет причин не постараться дополнительно, учитывая, какого эффекта вы сумеете добиться, затратив столь малые усилия!

Дело сделано, но...

Жанна Боярски

Сколько раз вы бывали на стендапах, ежедневных Scrum-летучках или встречах обсуждения статусов задач, где звучала фраза: «Дело сделано, но...»? Когда я слышу это, то сразу думаю: «Значит, еще не сделано». Когда «сделано» говорят о том, что пока не готово, возникают три проблемы.

1. Коммуникация и ясность

В идеале у вашей команды есть *определение* «сделанности» (DoD, definition of done). Но даже если нет, вероятно, есть какие-то взгляды на то, что подразумевается под «сделано». И, что еще лучше, человек, сообщающий о статусе, знает это определение. В ином случае у нас не будет границ ответственности за завершение задачи.

Обычно в список вещей, которые не *сделаны*, входят написание тестов, подготовка документации и описание пограничных условий. Найдите минутку и подумайте, сможете ли вы дополнить список. Точно так же мне не нравится выражение «уже точно сделано». Это неявно поддерживает концепцию, что «сделано» на самом деле не указывает на то, что работа выполнена. Соблюдайте четкость в общении. Если что-то не сделано, не говорите, что оно сделано.

Так вы сможете передавать коллегам больше сведений. Например, «я написал код для оптимистичного сценария, а затем добавлю тесты», или «я закончил весь код — мне осталось только обновить руководство пользователя», или даже «я думал, что закончил, а затем обнаружил, что виджет не работает по вторникам». Все это дает информацию вашей команде.

2. Восприятие

Менеджерам нравится, когда им говорят: «Сделано». Это означает, что вы можете взять на себя другую работу. Или помочь коллеге по команде. Или, по сути, заняться чем угодно, что не относится к выполненной задаче. Если

они слышат: «Сделано», то так и воспринимают статус работы. «…Но» либо забывается, либо становится незначительной мелочью. И теперь вы переходите к следующей задаче, не закончив предыдущую. Вот откуда берется технический долг! Да, иногда вы сами решаете набрать его. Однако сделать такой выбор после обсуждения гораздо лучше, чем позволить кому-то принять решение за вас лишь потому, что вы заявили, будто работа сделана.

Так! Заметка готова, но мне все еще нужно написать последнюю часть. Видите, как это работает? На самом деле я вовсе даже не закончила.

3. Вас не похвалят за полработы

«Сделано» — это двоичное состояние. Либо сделано, либо нет. Нельзя останавливаться на полпути. Предположим, вы изготовляете пару ходулей и говорите, что закончили на 50%. Подумайте о том, что это значит. Возможно, у вас готова одна ходуля. Не очень-то полезно. Что более вероятно, вы *думаете*, будто у вас есть одна ходуля, причем все равно необходимо изготовить вторую, а затем протестировать. Проверка, скорее всего, покажет, что вам нужно вернуться на предыдущую стадию и что-то изменить. А если нужно что-то переделать, значит, вы не выполнили задачу даже на 50%. Вы проявили излишний оптимизм.

Помните: не говорите, что сделали нечто, пока оно не *сделано*!

Сертификаты Java: пробирный камень технологий

Мала Гупта

Представьте, что вам нужно пройти роботизированную операцию. Хирург опытный и квалифицированный, но не имеет опыта работы с таким оборудованием для проведения вмешательств. Будете ли вы по-прежнему готовы на роботизированную операцию с этим врачом? Я бы согласилась только в том случае, если бы убедилась, что хирург умеет работать с данным оборудованием.

Продолжая аналогию: как бы вы определили навыки кандидата перед тем, как вводить его в свои важные проекты? Университетской степени в области компьютерных наук недостаточно. Между умениями, полученными в рамках университетской учебной программы, и требованиями к работнику очень большой разрыв.

Независимые образовательные организации предпринимают шаги, чтобы преодолеть эту пропасть. Но этого недостаточно. Кто и как будет измерять качество их информационных материалов? Вот тут-то и вступает в дело индустрия.

Подходящей метафорой тут будет пробирный камень — чудо-камень, применявшийся в древние времена для измерения чистоты золота и других драгоценных металлов, которые использовались в качестве платежных средств. Металлическую монету терли о темный кремнистый камень, похожий на яшму, и, если появлялся красочный след, металл считался чистым.

Организации вроде Oracle подтверждают уровень умений, выдавая специалистам профессиональные сертификаты. Благодаря этим «пробирным камням» IT-навыки оцениваются стандартизированным образом.

Люди часто спрашивают, необходимы ли эти профессиональные сертификаты для выпускников или аспирантов в области компьютерных наук. Возможно,

их университетская программа уже охватывала эту область знаний? Здесь нужно рассматривать краткосрочные и долгосрочные цели в перспективе. Бакалавриат или аспирантуру в области компьютерных наук в университете можно считать стратегическим решением при выборе долгосрочного карьерного пути, тогда как получение профессиональных сертификатов — тактическое решение. Так вы приобретете ценные технологические навыки, которые необходимо применять в текущих проектах, и достигнете краткосрочных целей.

Профессиональные сертификаты по Java от корпорации Oracle пользуются большим спросом. Они присуждаются кандидатам, которые соответствуют определенным требованиям. В зависимости от типа сертификации от претендента могут потребовать пройти какой-либо курс, завершить проект или сдать экзамен. Цель состоит в том, чтобы подтвердить: данное лицо имеет право занимать определенные должности или работать над определенными проектами. Получая сертифицированные навыки, кандидаты устраняют разрыв между своими существующими умениями и теми, которые необходимы в отрасли, и благодаря этому успешнее работают в проектах. Требования к сертификатам регулярно обновляются.

Oracle предлагает множество вариантов сертификации Java, в которых определяются темы и пути, которым должны следовать разработчики. Кандидаты могут выбрать подходящую сертификацию в соответствии со своими интересами.

Если человек подтверждает свои навыки, потенциальные работодатели убеждаются, что он способен программировать на определенном языке или же разбирается в какой-либо платформе, методологии или практике. С помощью сертификатов профессионалы преодолевают самый первый барьер — рассмотрение резюме и отбор кандидатов для собеседований.

Сертификаты Java помогают программистам успешно развивать карьеру. Когда люди ищут работу, а организации и команды пытаются найти таланты с подтвержденными навыками, эти документы могут стать первым шагом на пути к взаимовыгодному сотрудничеству.

Java — дитя 90-х

Бен Эванс

Есть только два вида языков: те, на которые люди жалуются, и те, которые никто не использует.

Бьерн Страуструп

Я не уверен, о чем в первую очередь говорится в мудром высказывании Страуструпа — о языках программирования или о человеческой природе. Тем не менее это привлекает внимание к часто забываемой прописной истине о том, что ЯП разрабатывают люди. Соответственно, языки всегда несут в себе следы той среды и контекста, в которых их создавали.

Поэтому вас не должно удивлять, что в концепции Java повсюду отыщутся приметы конца 1990-х годов — если вы знаете, где искать.

Например, последовательность байтов для загрузки ссылки на объект из локальной переменной `0` во временный вычислительный стек представляет собой такую двухбайтовую последовательность:

```
19 00 // aload 00
```

Однако набор команд байт-кода JVM предоставляет альтернативную форму, которая на один байт короче:

```
2A // aload_0
```

Возможно, вам покажется, что один сохраненный байт — это не так уж значительно, но такие сохранения могут накапливаться во всем файле класса.

Теперь вспомните, что в конце 90-х классы Java (часто апплеты) загружались через телефонные модемы, потрясающие устройства, чья скорость достигала невероятных значений — 14,4 килобита в секунду. При такой пропускной способности программисты на Java стремились экономить байты везде, где только возможно.

Можно даже заявить, что вся концепция примитивных типов — это комбинация взлома производительности и подачки C++ программистам,

пришедшим в мир Java, порожденная 1990-ми — периодом создания языка.

Даже «магическое число» (первые несколько байтов файла, которые позволяют операционной системе идентифицировать его тип) для всех файлов класса Java кажется устаревшим:

```
CA FE BA BE
```

В нынешние времена Cafe babe («цыпочка из кафе»), пожалуй, не очень здорово смотрится в Java. К сожалению, если смотреть на вещи реалистично, это уже нельзя поменять.

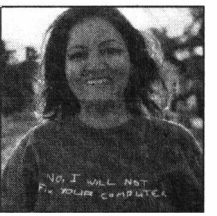
Дело не только в байт-коде: в стандартной библиотеке Java (особенно в ее старых частях) повсюду есть API-интерфейсы, копирующие эквивалентные API-интерфейсы C. Каждый программист, которому приходилось читать содержимое файла, слишком хорошо это знает. Хуже того, простого упоминания `java.util.Date` зачастую достаточно, чтобы многие Java-разработчики пошли красными пятнами.

Через призму 2020 года и последующих лет Java иногда рассматривается как широко распространенный, массовый язык. При этом из вида упускается, что мир программного обеспечения радикально изменился с момента дебюта Java. Ее шикарные идеи вроде виртуальных машин, динамического самоуправления, JIT-компиляции и сборки мусора теперь повсеместно встречаются в ландшафте языков программирования.

Хотя некоторые могут рассматривать Java как «столп общества», на самом деле это просто программирование в целом сместилось туда, где всегда находилась Java. Под маской корпоративной респектабельности она все еще дитя 90-х.

Программирование на Java в аспекте производительности JVM

Моника Беквит



Совет № 1. Не зацикливайтесь на мусоре

Я обнаружила, что иногда разработчики Java зацикливаются на том, сколько мусора производят их приложения. Такого рода одержимость оправдана лишь в очень редких случаях. Сборщик мусора (GC) помогает виртуальной машине Java (JVM) в управлении памятью. GC вместе с динамическим многоуровневым компилятором JIT (клиент (C1) + серверный класс (C2)) и интерпретатором образуют механизм выполнения для OpenJDK HotSpot VM. Существует множество оптимизаций, которые динамический компилятор может выполнить от вашего имени. Например, C2 способен использовать динамическое предсказание ветвей кода и в нем имеется вероятность («всегда» или «никогда») для принятых (или нет) ветвей кода. Точно так же C2 отлично показывает себя в оптимизациях, связанных с константами, циклами, копиями, деоптимизациями и так далее. Доверяйте адаптивному компилятору, но в случае сомнений проверяйте с помощью «исправности», «наблюдаемости», логгирования и прочих подобных инструментов, которые доступны благодаря нашей богатой экосистеме.

Что важно для GC, так это живучесть/возраст объекта, его «распространенность», «размер живого набора» для вашего приложения, длительные переходные значения, скорость распределения, накладные расходы на маркировку, ваш коэффициент продвижения (для коллектора поколений) и так далее.

Совет № 2. Охарактеризуйте и подтвердите свои бенчмарки

Один мой коллега однажды поделился некоторыми наблюдениями за набором бенчмаркинга с различными суббенчмарками. Один из них характеризо-

вался как бенчмарк для запуска приложения и всех связанных с этим процессов. Глядя на показатели производительности и исходя из того, что сравнение проводилось между выпусками OpenJDK 8u и OpenJDK 11u LTS, я поняла, что разница в цифрах могла появиться из-за изменения GC по умолчанию с Parallel GC на G1 GC.

Итак, похоже, что (суб)бенчмарк либо не был должным образом охарактеризован, либо не был пройден. И первое и второе — важные упражнения по бенчмаркингу. Они помогают идентифицировать и изолировать «единицу тестирования» (UoT) от других компонентов тестовой системы, которые могут выступать в качестве отвлекающих факторов.

Совет № 3. Размер и скорость распределения по-прежнему имеют значение

Чтобы как следует разобраться в проблеме, рассмотренной выше, я попросила показать мне логи GC. Через несколько минут стало ясно, что (фиксированный) размер области, основанный на размере кучи приложения, классифицирует «обычные» объекты как «огромные». Для G1 GC огромные объекты — это объекты, занимающие 50% или более области G1. Такие объекты не следуют быстрому пути распределения памяти и выделяются из старого поколения. Следовательно, размер распределения имеет значение для региональных GC.

GC следит за текущими изменениями графа объектов и перемещает объекты из пространства From в пространство To. Если ваше приложение выделяет ресурсы со скоростью, превышающей ту, с которой способен справиться ваш алгоритм (параллельной) разметки GC, это может стать проблемой. Кроме того, GC с поколениями объектов может преждевременно продвигать объекты с коротким сроком службы или не выдерживать переходные процессы должным образом из-за притока распределений памяти. GC G1 от OpenJDK пока только идет к независимости от его резервного, отказоустойчивого, неинкрементного, полностью обходящего кучу (параллельного) сборщика stop-the-world.

Совет № 4. Адаптивная JVM — это ваше право, требуйте его исполнения

Приятно видеть адаптивный JIT и все достижения, направленные на запуск, расширение, доступность JIT и оптимизацию занимаемой области.

Аналогичным образом нам доступна различная алгоритмическая интеллектуальность на уровне GC. Те GC, которых еще не хватает, должны скоро появиться, но это не произойдет без нашей помощи. Будучи разработчиками Java, пожалуйста, предоставьте сообществу отзывы о вашем примере использования и помогите стимулировать инновации в данной области. Кроме того, протестируйте функции, которые постоянно добавляются в JIT.

Java должна приносить радость

Холли Камминс

В начале моей карьеры Java-программиста я пользовалась J2EE 1.2. У меня возникали вопросы. Почему на каждый компонент приходится четыре класса и сотни строк сгенерированного кода? Почему компиляция крошечных проектов занимает полчаса? Это непродуктивно и не радует. Такие характеристики часто ходят парой: нечто нас не радует, поскольку мы знаем, что оно бесполезно. Представьте себе собрания, где ничего не решается, отчеты о состоянии дел, которые никто не читает...

Если безрадостность — это плохо, то что же тогда радость? Она хорошая? И как ее обрести? У нее есть разные обличья:

- исследование (целенаправленные изыскания);
- игра (сама по себе, без цели);
- головоломки (правила и цель);
- состязание (правила и победитель);
- работа (цель приносит удовлетворение).

Java позволяет обрести все это — насчет «работы» все очевидно, а про «головоломки» знает любой, кто отлаживал программу на Java. (Отладка не обязательно доставляет удовольствие, но отыскать решение — это здорово.) Мы учимся посредством «исследования» (когда мы новички в чем-то) и «игры» (когда уже знаем достаточно, чтобы создавать что-нибудь).

Если не касаться радости, которую приносит нам использование Java, действительно ли этот язык увлекателен по своей природе? *Несомненно*, Java многословна по сравнению с более молодыми ЯП. Шаблонность — это не весело, но и тут кое-что поправимо. Например, Lombok аккуратно генерирует геттеры и сеттеры, а также методы `hashCode` и `equals` (в противном случае они трудоемки и подвержены ошибкам). Вручную записывать трассировку входа

и выхода неинтересно, но аспекты или библиотеки трассировки могут работать динамически (и значительно улучшают читаемость кода).

Что порождает интерес при использовании? Отчасти выразительность и понятность, но дело не только в этом. Я не уверена, что лямбды в целом короче или понятнее, чем альтернативы на основе классов. Но они такие забавные!

Когда вышла Java 8, разработчики нырнули в лямбды, словно дети в бассейн с мячиками. Мы хотели узнать, как это работает (исследование), и решить проблему выражения алгоритмов в функциональном стиле (головоломки).

В Java «самое интересное, что можно сделать» часто совпадает с «самым лучшим» (выигрыш). Автоинструментирующая трассировка обходит безрадостные моменты, устраняя ошибки копирования и вставки имени метода и повышая четкость кода. Или возьмите производительность. Для нишевых сценариев необходим странный, сложный код, чтобы наскрести каждую крупицу скорости. Однако в большинстве случаев самый простой код также оказывается *самым быстрым*. (Что не всегда верно для таких языков, как C.) Компилятор Java JIT оптимизирует код по мере его выполнения. Он лучше всего подходит для чистого идиоматического кода. Простой код хорошо читается, поэтому ошибки будут более очевидными.

Код, с которым вы мучаетесь, имеет побочный эффект. Психологические исследования показывают, что карьерный успех напрямую зависит от счастья. Одно исследование (*https://oreil.ly/ pmfaZ*) показало, что люди с позитивным мышлением работали на 31% продуктивнее, чем люди с нейтральным или негативным мышлением. Вы покажете не лучший результат, если станете использовать плохо спроектированные библиотеки, и в дальнейшем будете показывать не лучшие результаты, потому что настрадались с дурным кодом.

А не оправдывает ли концепция «радость — это хорошо» безответственность? Вовсе нет! Убедитесь, что радостно *всем*, включая клиентов, коллег и будущих сопровождающих вашего кода. По сравнению с динамически типизированными языками сценариев, которые бывают быстрыми и нестрогими, Java уже имеет ярлычок «безопасная и ответственная». Но, создавая программы, нам также следует писать код со всей ответственностью.

Хорошая новость здесь в том, что компьютеры способны выполнять почти все скучные задачи быстрее и правильнее, чем люди. Машины не ждут, что им будет весело (пока), так что пользуйтесь ими! Не смиряйтесь со скукой. Если что-то кажется вам неинтересным, поищите способ получше. Если такового нет, придумайте его. Мы же программисты, мы можем пофиксить уныние.

Неуказываемые типы Java

Бен Эванс

Что такое `null`?

Начинающие Java-программисты часто испытывают трудности с пониманием этой концепции. Простой пример показывает истину:

```
String s = null;
Integer i = null;
Object o = null;
```

Следовательно, символ `null` должен быть значением.

Каждое значение Java имеет свой тип, а если так, то `null` должен иметь свой тип. Какой же?

Очевидно, что ни один из тех, с которыми мы обычно сталкиваемся. Переменная типа `String` не может содержать значение типа `Object` — свойства подстановки Liskov просто не работают таким образом.

Интерфейс типа локальной переменной Java 11 также не помогает:

```
jshell> var v = null;
| Error:
| cannot infer type for local variable v
|  (variable initializer is 'null')
| var v = null;
| ^ - - - - - - ^
```

Возможно, прагматичный Java-программист просто почешет в затылке и решит, как уже делали многие, что на самом деле это не очень-то важно. А потом он или она притворится, что «`null` — это просто специальный литерал, который может иметь любой ссылочный тип».

Однако для тех из нас, кто считает такой подход неудовлетворительным, верный ответ отыщется в «Спецификации языка Java» (JLS) в разделе 4.1:

> Существует также специальный тип `null` — тип выражения `null` (§3.10.7, §15.8.1), который не имеет имени. Поскольку тип `null` не имеет имени, невозможно объявить переменную типа `null` или привести к типу `null`.

Вот оно. Java позволяет нам записывать значения, типы которых мы не можем объявлять, как типы переменных. Мы могли бы назвать их «невыразимыми типами» или, формально, *необозначаемыми типами*.

Как показывает пример `null`, мы фактически применяем их постоянно. Есть еще два очевидных места, где встречаются такого рода типы. Первое появилось в Java 7, и JLS есть что сказать о них:

> Параметр исключения может обозначать свой тип либо как отдельный тип класса, либо как объединение двух или более типов классов (называемых альтернативами).

Истинный тип параметра multicatch — это объединение различных возможных перехватываемых типов. На практике будет компилироваться только код, соответствующий контракту API *ближайшего общего супертипа* альтернатив. Реальный тип параметра — это не то, что мы можем использовать в качестве типа переменной.

Каков тип «o» в следующем примере?

```
jshell> var o = new Object() {
...> public void bar() { System.out.println("bar!"); }
...> }
o ==> $0@3bfdc050jshell> o.bar();
bar!
```

Эта переменная — точно не объект (`Object`), потому что для нее можно вызвать `bar()`, а у типа `Object` нет такого метода. Вместо этого истинный тип не обозначается — у него нет имени, которое мы использовали бы в качестве типа переменной в коде Java. Во время выполнения тип представляет собой просто назначенный компилятором заполнитель (`$0` в нашем примере).

Задействуя `var` в качестве «волшебного типа», программист сохраняет информацию о типе для каждого отдельного использования `var` до конца метода.

Нам недоступен перенос типов из одного метода в другой. Чтобы сделать это, нам пришлось бы объявить возвращаемый тип — а ведь именно такое действие нам недоступно!

Соответственно, применимость данных типов ограничена. Система типов Java остается в значительной степени номинальной, и кажется маловероятным, что настоящие структурные типы когда-либо появятся в языке.

Наконец, мы должны отметить, что многие из более продвинутых вариантов использования обобщений (включая загадочные ошибки `capture of ?`) также лучше всего понятны в терминах неуказываемых типов — но это уже другая история.

JVM — мультипарадигмальная платформа. Используйте это, чтобы повысить свой уровень программирования

Рассел Уиндер

Java — это императивный язык: Java-программы сообщают JVM, что делать и когда. Но в вычислениях самое важное — все, что связано с построением абстракций. Java превозносят как объектно-ориентированный язык: к абстракциям в ней относятся объекты, методы и передача сообщений с помощью вызова метода. На протяжении многих лет люди создавали все более и более масштабные системы, используя объекты, методы, обновляемое состояние и явную итерацию, и появились болевые точки. Многие из них «залепляют пластырем» с помощью высококачественного тестирования, но все равно программисты в итоге занимаются «взломом», чтобы обойти различные проблемы.

С появлением Java 8 язык Java претерпел крайне революционные изменения: в него ввели ссылки на методы, лямбда-выражения, методы по умолчанию для интерфейсов, функции более высокого порядка, неявную итерацию и ряд прочих новинок. Java 8 представила совершенно другой набор методик реализации алгоритмов.

Императивное и декларативное мышление — очень разные способы выражения алгоритмов. В 1980-х и 1990-х годах эти позиции считались обособленными и несовместимыми: у нас шла война между сторонниками объектно-ориентированного и функционального программирования. Smalltalk и C++ выступали за объектную ориентацию, а Haskell защищал функциональность. Позже C++ перестал быть ООЯ и позиционировал себя как мультипарадигмальный язык. Java превратилась в лидера объектно-ориентированного лагеря. Однако с появлением Java 8 язык Java тоже стал мультипарадигмальным.

Еще в начале 1990-х годов появилась JVM, разработанная для обеспечения переносимости Java (давайте не затрагивать историю проекта Green и языка

программирования Oak). Первоначально она предназначалась для создания подключаемых модулей веб-браузера, но процесс быстро перешел к созданию серверных систем. Java компилируется в аппаратно-независимый байт-код JVM, и интерпретатор выполняет байт-код. Благодаря компиляторам JIT вся модель интерпретации выполняется намного быстрее, без изменения вычислительной модели JVM.

Поскольку JVM стала чрезвычайно популярной платформой, появились другие языки, которые использовали байт-код в качестве целевой платформы: Groovy, JRuby и Clojure — динамические ЯП, выполняющиеся на JVM; а также статические языки Scala, Ceylon и Kotlin. Scala, в частности, показала в конце 2000-х годов, что объектно-ориентированное и функциональное программирование можно интегрировать в единый мультипарадигмальный язык. Если Clojure — функциональный язык, Groovy и JRuby с самого начала были мультипарадигмальными. Kotlin применяют для создания языков на JVM в 2010-х и 2020-х годах, опираясь на опыт Java, Scala, Groovy и т. д.

Чтобы использовать JVM наилучшим образом, мы должны выбрать правильный язык программирования для решения задачи. Это не обязательно означает, что в работе будет применяться один ЯП: можно использовать разные языки для разных кусочков, и всё благодаря JVM. Так, мы вправе применять Java или Kotlin для частей, которые лучше всего выражаются в виде статического кода, и Clojure или Groovy для частей, которые лучше всего обрабатываются динамическим кодом. Вы измучаетесь, если попробуете написать динамический код на Java, поэтому используйте правильный инструмент для данной работы, ведь все ЯП способны взаимодействовать в JVM.

Держите руку на пульсе

Триша Джи

Я изучала в университете Java версии 1.1 (жаль, это потому, что я стара, а не потому, что мой университет полагался на древние технологии). В то время она была сравнительно маленьким языком, а мне хватало наивности полагать, будто я выучила всю Java, которую следует знать, и впереди у меня беззаботная жизнь Java-программиста.

Когда я впервые поступила на работу — на тот момент я еще училась в университете и писала на Java менее года, — появилась Java 1.2. В ней представили совершенно иную библиотеку пользовательского интерфейса (UI), называемую Swing (*https://oreil.ly/6bJM0*), поэтому я провела то лето, изучая Swing, чтобы улучшить взаимодействие пользователей с нашей программой.

Пару лет спустя, на моей первой работе после окончания университета, я обнаружила, что апплеты устарели, а на сцену вышли сервлеты. Следующие шесть месяцев я потратила на изучение сервлетов и JSP (*https://oreil.ly/G_LNk*), чтобы мы смогли предоставить нашим пользователям регистрационную онлайн-форму.

На своей следующей работе я выяснила, что, по-видимому, мы больше не используем `Vector` (*https://oreil.ly/uFBk4*), но применяем `ArrayList` (*https://oreil.ly/VrWT3*). Это потрясло меня до глубины души. Как могут сами основы языка, сами структуры данных уходить у меня из-под ног? Если первые два открытия были связаны с изучением дополнений к языку, то третье касалось изменений в вещах, которые, как мне думалось, я уже знала. Как же мне разобраться во всем этом, если я больше не учусь в университете, где мне что-то преподавали?

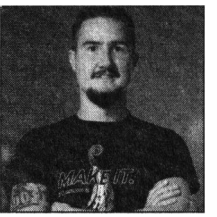
На тех ранних работах мне повезло, что меня окружали люди, осведомленные о технологических изменениях, повлиявших на проекты Java, которыми я занималась. Именно такую роль должны играть старшие члены команды: не просто выполнять то, что им говорят, но и вносить предложения о том, как это сделать, и помогать младшим коллегам совершенствоваться вместе с ними.

Чтобы выжить в качестве Java-программиста, вам нужно признать, что Java — живой и подвижный язык. Он развивается не только через новые версии, но и посредством библиотек, фреймворков и даже новых языков JVM. Возможно, поначалу это напугает и ошеломит вас. Но «быть в курсе событий» не означает, что вы должны изучать всё подряд — вам просто нужно держать руку на пульсе, прислушиваться к ходовым фразам и разбираться в технологических тенденциях. Вам необходимо углубляться только в темы, которые относятся к вашей работе или же интересны для вас лично (а в идеале и то и другое).

Зная, что доступно в текущей версии Java и что планируется для будущих версий, вы сумеете реализовать функции или функциональные возможности, которые помогут вашим пользователям делать то, что им нужно. Это означает, что данный подход повышает вашу продуктивность как разработчика. Java теперь выпускает новую версию каждые шесть месяцев, так что вы действительно облегчите себе жизнь, если будете держать руку на пульсе.

Виды комментариев

Николай Парлог



Предположим, вы хотите добавить некие комментарии в свой Java-код. Что вы используете — «/**», «/*» или «//»? И куда именно вы их поставите? Помимо синтаксиса, существуют устоявшиеся практики, которые придают семантическое значение тому, что где используется.

Комментарии Javadoc для контрактов

Комментарии Javadoc (те, что заключены в `/** ... */`) применяются исключительно для классов, интерфейсов, полей и методов и размещаются непосредственно над ними. Перед вами пример `Map::size`:

```
/**
 * Возвращает количество сопоставлений ключ-значение в этой карте.
 * Если
 * карта содержит больше элементов, чем Integer.MAX_VALUE, то
 * возвращает
 * Integer.MAX_VALUE.
 *
 * @return количество сопоставлений ключ-значение в этой карте
*/
```

Пример демонстрирует синтаксис, а также семантику: комментарий Javadoc — это контракт. Он сообщает пользователям API, чего им следует ожидать, и при этом сохраняет центральную абстракцию типа нетронутой, не говоря о деталях реализации. В то же время он обязывает разработчиков обеспечивать указанное поведение.

Версия Java 8 немного ослабила эту строгость: формализовала различные интерпретации, ввела (нестандартные) теги `@apiNote`, `@implSpec` и `@implNote`. Префиксы `api` или `impl` указывают, кому адресован комментарий — пользователям или разработчикам. Суффиксы `Spec` или `Note` уточняют,

действительно ли это спецификация или только для примера. Обратили внимание, что `@apiSpec` не упомянут? Это потому, что текст комментария без тегов должен выполнять эту роль — указывать API.

Блоки комментариев для контекста

Блоки комментариев заключены в «`/* ... */`». Нет никаких ограничений на то, где их размещать, и инструменты обычно игнорируют их. Как правило, их вставляют в начале класса или даже метода, чтобы дать представление о его реализации. Там могут приводить как технические детали, так и описание контекста, в котором создавался код (знаменитое «почему» из присловья «код говорит вам *что*, комментарии говорят вам *почему*») или неиспользованные пути. Хороший пример предоставления подробных сведений о реализации можно найти в `HashMap`. Начинается он так*:

```
/*
     * Implementation notes.
     *
     * This map usually acts as a binned (bucketed) hash table,
     * but when bins get too large, they are transformed into bins
     * of TreeNodes, each structured similarly to those in
     * java.util.TreeMap.
     * [...]
     */
```

Как правило, когда ваше первое решение не становится финальным, когда вы идете на компромисс или когда ваш код преобразуется из-за странного требования или неудобного API зависимости, стоит подумать о документировании этого контекста. Ваши коллеги и вы сами в будущем поблагодарят вас. (Про себя.)

Строчные комментарии для странностей

Строчные комментарии начинаются с «//», которые должны повторяться в каждой строке. Нет никаких ограничений на то, где их использовать, но обычно их помещают над комментируемой строкой или блоком

* Примечания по внедрению. Эта карта обычно действует как привязанная (добавленная в бакет) хэш-таблица, но когда корзины становятся слишком большими, они превращаются в корзины из TreeNodes, каждый из узлов в которых структурирован аналогично узлам в java.util.TreeMap.

(а не в конце). Инструменты игнорируют их, многие разработчики поступают так же. В строчных комментариях часто описывается, что делает код, и в целом это справедливо считается плохой практикой. Такой подход все же бывает полезен в определенных случаях: например, когда код должен использовать тайные языковые функции или его легко взломать незаметным способом (наглядным примером здесь служит параллелизм).

В заключение

- Убедитесь, что выбрали правильный вид комментария.
- Не обманывайте ожиданий.
- Комментируйте ваш &#!*@$ код!

Знай `flatMap` свой

Дэниел Инохоса

Названия должностей постоянно меняются. По аналогии с медицинским сообществом, где люди переходят от широкой к узкой специализации, некоторые из нас, ранее просто «программисты», теперь занимают другие должности. Одна из новейших специализированных дисциплин — *инженер по обработке данных*. Он управляет ими, создавая конвейеры, фильтруя данные, преобразуя их и превращая в то, что требуется ему или кому-то другому для принятия бизнес-решений в режиме реального времени с помощью потоковой обработки.

Как обычный программист, так и инженер по обработке данных должны овладеть методом `flatMap`, одним из самых важных инструментов для любого функционального, дееспособного языка, такого как наша любимая Java. Также он ценен для фреймворков больших данных и потоковых библиотек. Метод `flatMap`, как и его партнеры `map` и `filter`, применим для всего, что представляет собой «контейнер чего-то» — например, `Stream<T>` и `CompletableFuture<T>`. Если вы хотите выйти за рамки стандартной библиотеки, есть также `Observable<T>` (RXJava) и `Flux<T>` (Project Reactor).

В Java мы задействуем `Stream<T>`. Идея `map` проста: возьмите все элементы потока или коллекции и примените к ним функцию:

```
Stream.of(1, 2, 3, 4).map(x -> x * 2).collect(Collectors.toList())
```

Результат таков:

```
[2, 4, 6, 8]
```

Что произойдет, если мы напишем так?

```
Stream.of(1, 2, 3, 4)
      .map(x -> Stream.of(-x, x, x + 1))
      .collect(Collectors.toList())
```

К сожалению, мы получаем `List` потоков `Stream` конвейера:

```
[java.util.stream.ReferencePipeline$Head@3532ec19,
 java.util.stream.ReferencePipeline$Head@68c4039c,
 java.util.stream.ReferencePipeline$Head@ae45eb6,
 java.util.stream.ReferencePipeline$Head@59f99ea]
```

Но, если подумать, то, конечно, для каждого элемента `Stream` мы создаем другой `Stream`. И еще, загляните поглубже в `map(x -> Stream.of(...))`. Для каждого единичного элемента мы создаем множественное число. Если вы выполняете `map` с множественным числом, пора распечатывать `flatMap`:

```
Stream.of(1, 2, 3, 4)
      .flatMap(x -> Stream.of(-x, x, x+1))
      .collect(Collectors.toList())
```

Это даст то, к чему мы стремились:

```
[-1, 1, 2, -2, 2, 3, -3, 3, 4, -4, 4, 5]
```

Возможности использования `flatMap` необъятны.

Давайте перейдем к чему-то более сложному, подходящему для любой задачи функционального программирования или разработки данных. Рассмотрим следующую взаимосвязь, в которой опущены геттеры, сеттеры и `toString`:

```
class Employee {
   private String firstName, lastName;
   private Integer yearlySalary;
   // геттеры, сеттеры, toString
}

class Manager extends Employee {
   private List<Employee> employeeList;
   // геттеры, сеттеры, toString
}
```

Предположим, что нам даны лишь `Stream<Manager>` и наша цель — определить все зарплаты всех сотрудников, включая `Manager` и `Employee`. У нас может возникнуть соблазн сразу перейти к `forEach` и начать копаться в этих зарплатах. К сожалению, так мы смоделируем наш код в соответствии со структурой данных, что породит ненужную сложность. Более удачное решение — пойти

от обратного и структурировать данные в соответствии с нашим кодом. Вот тут-то и появляется `flatMap`:

```
List.of(manager1, manager2).stream()
    .flatMap(m ->
       Stream.concat(m.getEmployeeList().stream(), Stream.of(m)))
    .distinct()
    .mapToInt(Employee::getYearlySalary)
    .sum();
```

Этот код принимает каждый объект-менеджера и возвращает множественное число — менеджеров и их сотрудников. Затем мы применяем `flatMap` к этим коллекциям, чтобы сделать один `Stream`, и выполняем `distinct`, чтобы отфильтровать дубликаты. Теперь мы можем рассматривать их все как одну коллекцию. Остальное несложно. Сначала выполняем специфичный для Java вызов `mapToInt`, который извлекает их `yearlySalary` и возвращает `IntStream`, специализированный тип `Stream` для целых чисел. Наконец, суммируем `Stream`. Готов лаконичный код.

Если вы используете `Stream` или другой тип `C<T>`, где `C` — любой поток или коллекция, продолжайте обработку данных с использованием `map`, `filter`, `flatMap` или `groupBy`, пока не придете к `forEach` или к любой другой терминальной операции, например `collect`. Если вы начнете работу с терминальной операцией преждевременно, то полностью утратите отлаженность и оптимизацию, предоставленные вам Java `Stream`, потоковыми библиотеками или фреймворками больших данных.

Знайте свои коллекции

Никхил Нанивадекар

Коллекции — основной элемент любого языка программирования. Они обычно составляют один из основных строительных блоков разрабатываемого кода. Язык Java представил фреймворк коллекций давным-давно, еще в JDK 1.2. Многие программисты фактически используют `ArrayList` в качестве своей рабочей коллекции. Однако в коллекциях есть не только `ArrayList`, так что давайте исследуем этот вопрос.

Коллекции можно классифицировать как *упорядоченные* или *неупорядоченные*. Упорядоченные имеют предсказуемый порядок итераций. Неупорядоченные не имеют предсказуемого порядка итераций. Классифицировать можно и по другому параметру: *сортированные* или *несортированные*. Элементы в отсортированной коллекции упорядочиваются от начала до конца в последовательности, определяемой компаратором. Несортированные — не имеют определенной последовательности, определяемой по элементам. Хотя слова «сортировка» и «упорядочение» схожи по смыслу, они не всегда взаимозаменяемы в отношении коллекций. Важное различие состоит в том, что *упорядоченные* коллекции имеют предсказуемый порядок итераций, но не порядок сортировки, тогда как *отсортированные* коллекции имеют предсказуемый порядок сортировки, а значит, и предсказуемый порядок итераций.

Помните: все отсортированные коллекции упорядочены, но не все упорядоченные коллекции отсортированы. В JDK существуют различные упорядоченные, неупорядоченные, отсортированные и несортированные коллекции. Давайте взглянем на некоторые из них.

`List` (список) — это интерфейс для упорядоченных коллекций со стабильным порядком индексации. Списки позволяют вставлять повторяющиеся элементы и обеспечивают предсказуемый порядок итераций. В JDK предложены такие реализации `List`, как `ArrayList` и `LinkedList`. Для поиска конкретного элемента можно использовать метод `contains`. Операция `contains` выполняет обход списка с самого начала, следовательно, сложность поиска элементов в `List` составляет O(*n*).

Map (карта) — это интерфейс, который поддерживает отношения «ключ-значение» и сохраняет только уникальные ключи. Если к карте добавляется тот же ключ и другое значение, старое значение заменяется новым значением. В JDK предложены такие реализации Map, как HashMap, LinkedHashMap и TreeMap.

HashMap неупорядочена, тогда как LinkedHashMap упорядочена. Обе они зависят от hashCode и equals при определении уникальных ключей. TreeMap сортирована: ключи в ней отсортированы согласно компаратору или в порядке сортировки ключей Comparable. TreeMap зависит от compareTo при определении порядка сортировки и уникальности ключей. Map предоставляет методы containsKey и containsValue, чтобы найти определенный элемент. Функция containsKey ищет ключ во внутренней хэш-таблице для HashMap. Если результатом поиска становится ненулевой объект, он проверяется на равенство с объектом, переданным в containsKey. Операция containsValue выполняет обход всех значений с самого начала. Следовательно, поиск ключей в HashMap — это операция O(1), тогда как поиск значений в HashMap — это операция O(n).

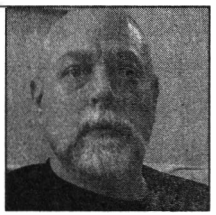
Set — это интерфейс для коллекций уникальных элементов. В JDK наборы (set) поддерживаются картами (map), где ключи являются элементами, а значения равны нулю. В JDK представлены такие реализации Set, как HashSet (поддерживается HashMap), LinkedHashSet (поддерживается LinkedHashMap) и TreeSet (поддерживается TreeMap).

При поиске конкретного элемента для Set можно использовать метод contains. Метод contains для Set применяет метод containsKey из интерфейса Map и, следовательно, представляет собой операцию O(1).

Коллекции — это важная часть головоломки программного обеспечения. Чтобы использовать их эффективно, необходимо понимать их функциональность, их реализацию и, что не менее важно, последствия использования шаблона итерации. Не забывайте читать документацию и писать тесты, применяя эти универсальные базовые строительные блоки кода.

Обратите внимание на Kotlin

Майк Данн



Java, возможно, самый зрелый и проверенный ЯП среди тех, что все еще широко используются, и это вряд ли сильно изменится в обозримом будущем. Чтобы содействовать реализации современных представлений о том, что должен делать язык программирования, кое-какие умные люди решили написать новый ЯП, который выполнял бы все функции Java, а также несколько интересных новых штучек, которые подходили бы для довольно безболезненного изучения и были бы в значительной степени совместимыми. Человек вроде меня, годами работающий над одним и тем же огромным приложением для Android, может, к примеру, написать один класс в Kotlin, не совершая полной миграции.

Kotlin нужен вам для того, чтобы создавать более короткий, чистый и современный код. Хотя текущие и анонсированные версии Java тоже способны решать многие проблемы, с которыми справляется Kotlin, этот ЯП может особенно пригодиться разработчикам на Android, застрявшим где-то между Java 7 и Java 8.

Давайте рассмотрим несколько примеров, таких как шаблон конструктора свойств для моделей в Kotlin. Начнем с простого варианта того, как может выглядеть модель в Java:

```
public class Person {
  private String name;
  private Integer age;
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public Integer getAge() {
    return age;
```

```
  }
  public void setAge(int age) {
    this.age = age;
  }
}
```

Мы можем создать специальный конструктор для получения некоторых начальных значений:

```
public class Person {
  public Person(String name, Integer age) {
    this.name = name;
    this.age = age;
  }
  ...
}
```

Не так уж плохо, но вы, вероятно, уже видите, что после добавления еще нескольких свойств определение для этого довольно простого класса очень быстро раздуется. Давайте взглянем на тот же класс в Kotlin:

```
class Person(val name:String, var age:Int)
```

Вот и все! Еще один отличный пример — делегирование. Делегаты Kotlin позволяют вам предоставлять логику для любого количества операций чтения. Один из примеров здесь — отложенная инициализация, концепция, которая наверняка знакома Java-разработчикам. Это может выглядеть примерно так:

```
public class SomeClass {
  private SomeHeavyInstance someHeavyInstance = null;
  public SomeHeavyInstance getSomeHeavyInstance() {
    if (someHeavyInstance == null) {
      someHeavyInstance = new SomeHeavyInstance();
    }
    return someHeavyInstance;
  }
}
```

Опять же, не так уж страшно, сделано просто и без конфигурации, но, скорее всего, вы будете повторять один и тот же код несколько раз в своем проекте,

нарушая принцип DRY (Don't Repeat Yourself — «Не повторяйся»). Кроме того, это не потокобезопасно. Вот версия Kotlin:

```
val someHeavyInstance by lazy {
  return SomeHeavyInstance()
}
```

Коротко, мило и читабельно. Все шаблонные фрагменты прекрасно спрятаны под капотом. О, а еще это потокобезопасно. Еще одно серьезное улучшение — null-безопасность.

После ссылки на объект, который может быть null в Kotlin, вы увидите множество операторов вопросительных знаков:

```
val something = someObject?.someMember?.anotherMember
```

Вот то же самое в Java:

```
Object something = null;
if (someObject != null) {
  if (someObject.someMember != null) {
    if (someObject.someMember.anotherMember != null) {
      something = someObject.someMember.anotherMember;
    }
  }
}
```

Оператор проверки null-значения (?) немедленно прекратит вычисление и вернет null, как только любой из референтов в цепочке примет значение null.

Давайте завершим еще одной убийственной функцией — сопрограммами (корутинами). В двух словах, сопрограмма выполняет работу асинхронно с вызывающим кодом, хотя это задание может передаваться некоторому количеству потоков. Важно отметить, что, даже если один поток обрабатывает несколько сопрограмм, Kotlin как-то волшебно переключает контекст, и несколько заданий запускаются одновременно. В то время как конкретное поведение настраивается, сопрограммы, естественно, задействуют выделенный пул потоков, но используют переключение контекста в пределах одного потока (очень круто!). Поскольку речь о Kotlin, они также бывают

причудливыми, комплексными и технически переусложненными, но по умолчанию они очень просты:

```
launch {
  println("Hi from another context")
}
```

Однако не забывайте о различиях между потоками и сопрограммами — например, вызов `object.wait()` в одной задаче приведет к приостановке всех других задач, работающих в содержащем ее потоке. Попробуйте Kotlin и составьте свое мнение.

Изучайте идиомы Java и храните их в памяти

Жанна Боярски

Нам, программистам, приходится часто выполнять определенные задачи. Так, просмотр данных и применение условия встречаются повсюду. Вот два способа подсчитать, сколько положительных чисел находится в списке:

```
public int loopImplementation(int[] nums) {
  int count = 0;
  for (int num : nums) {
    if (num > 0) {
      count++;
    }
  }
  return count;
}

public long streamImplementation(int[] nums) {
  return Arrays.stream(nums)
               .filter(n -> n > 0)
               .count();
}
```

Оба они выполняют одно и то же, и оба используют общие идиомы Java. Идиома — это распространенный способ выражения какой-то небольшой части функциональности, в отношении которой сообщество Java пришло к общему согласию. Если вы начнете писать идиомы быстро, не задумываясь о них, то будете создавать код намного проворнее. Когда вы пишете код, ищите шаблоны вроде указанных выше. Вы даже можете попрактиковаться в них, чтобы ускорить процесс, и выучить их наизусть.

Некоторые идиомы, такие как цикл, условия и потоки, применяются всеми Java-программистами. Другие вы используете, только работая над какими-то

особыми типами кода. Например, я часто задействую регулярные выражения и файловый I/O. Приведенную ниже идиому я обычно применяю при вводе-выводе файлов. Она считывает файл, удаляет все пустые строки и записывает его обратно:

```
Path path = Paths.get("words.txt");
List<String> lines = Files.readAllLines(path);
lines.removeIf(t -> t.trim().isEmpty());
Files.write(path, lines);
```

Если бы я работала в команде и наши файлы не помещались бы в память, мне пришлось бы использовать другую идиому программирования. Впрочем, я имею дело с небольшими файлами и такой проблемы у меня не возникает, так что написать четыре строки ради создания чего-то мощного вполне оправданно.

Обратите внимание, что при использовании этих идиом большая часть кода сохраняется вне зависимости от вашей задачи. Если я хочу получить отрицательные числа или нечетные числа, то просто меняю оператор *if* или фильтр. Если мне нужно удалить все строки длиной более 60 символов, то достаточно изменить условие в `removeIf`:

```
lines.removeIf(t -> t.length() <= 60);
```

Так или иначе, здесь я думаю о том, чего хочу достичь, а не ищу способы чтения файла или подсчета значения. Эту идиому я выучила давным-давно.

Интересная особенность идиом заключается в том, что вы не всегда заучиваете их намеренно. Я не говорила себе: «Садись и зубри идиому для чтения/записи файла». Я запомнила ее, потому что часто использовала. Если вы многократно ищете какие-то сведения, то усвоите их. Или, по крайней мере, разберетесь, где их найти. Например, у меня возникают проблемы с запоминанием флагов регулярных выражений. Я знаю, что они делают, но путаю «`?s`» и «`?m`». Эти выражения я искала столько раз, что теперь мне известно: чтобы получить ответ, следует погуглить «шаблон javadoc».

В заключение: пусть ваш мозг служит кэшем. Изучите идиомы и общие вызовы API библиотеки. Знайте, где быстро найти все прочее. Это позволит вам высвободить мозг для работы над чем-то сложным!

Учитесь создавать kata и создавайте kata, чтобы учиться

Дональд Рааб

Каждый разработчик Java должен получать новые навыки и поддерживать усвоенные умения в актуальном состоянии. Экосистема Java огромна и продолжает развиваться. Учитывая, сколь многому вам предстоит научиться, необходимость идти в ногу со временем способна напугать. Мы поможем друг другу сохранить темп на быстро меняющемся ландшафте, если будем сотрудничать как сообщество, делясь знаниями и практикой. Один из возможных способов — получение, создание и совместное использование code kata.

Code kata (ката) — это упражнение по программированию, помогающее оттачивать на практике определенные навыки. Некоторые code kata предоставят вам структуру для проверки того, что навык был приобретен, запуская модульные тесты для вашего решения. Code kata — отличный способ для разработчиков поделиться практическими упражнениями с самим собой в будущем и с другими программистами, у которых можно поучиться.

Вот как создать свое первое code kata:

1. Выберите тему, которую вы хотите изучить.
2. Напишите проходящий модульный тест, который демонстрирует некоторую часть знаний.
3. Повторяйте рефакторинг кода до тех пор, пока не достигнете удовлетворительного финального решения. После каждого рефакторинга проверяйте, проходит ли тест.
4. Удалите решение в упражнении и оставьте один из неудачных тестов.
5. Зафиксируйте неудачный тест с поддерживающим кодом и создайте артефакты в системе управления версиями (VCS).
6. Опубликуйте исходный код, чтобы поделиться им с другими.

Теперь я продемонстрирую, как создать небольшое ката, для чего выполню первые четыре шага:

1. Тема: узнать, как соединять строки в `List`.
2. Напишите работающий тест JUnit, который показывает, как соединять строки в `List`:

```
@Test
public void joinStrings() {
   List<String> names =  Arrays.asList("Sally", "Ted", "Mary");
   StringBuilder builder = new StringBuilder();
   for (int i = 0; i < names.size(); i++) {
     if (i > 0) {
        builder.append(", "); }
        builder.append(names.get(i));
     }
     String joined = builder.toString();
     Assert.assertEquals("Sally, Ted, Mary", joined);
}
```

3. Проведите рефакторинг кода, чтобы использовать `StringJoiner` Java 8. Повторите тест:

```
StringJoiner joiner = new StringJoiner(", ");
for (String name : names) {
   joiner.add(name);
}
String joined = joiner.toString();
```

Проведите рефакторинг кода, чтобы использовать потоки Java 8. Повторите тест:

```
String joined = names.stream().collect(Collectors.joining(", "));
```

Проведите рефакторинг, чтобы использовать `String.join`. Повторите тест:

```
String joined = String.join(", ", names);
```

4. Удалите решение и оставьте неудачный тест с комментарием:

```
@Test
public void joinStrings() {
   List<String> names = Arrays.asList("Sally", "Ted", "Mary");
   // Соедините имена и разделите их с помощью ", "
   String joined = null;
   Assert.assertEquals("Sally, Ted, Mary", joined);
}
```

Сделал сам, помоги другому — шаги 5 и 6 я предоставляю выполнить читателю.

Пожалуй, этот пример достаточно прост, чтобы успешно проиллюстрировать, как создавать собственные ката различной сложности, эффективно используя модульные тесты. Так вы получите опорную структуру для накопления уверенности и способности к осмыслению.

Цените свое обучение и запас сведений. Узнав что-то полезное, запишите это. Весьма уместно сохранять практические упражнения, чтобы затем напоминать себе, как все работает. Фиксируйте свои знания и исследования в code kata. Ката, благодаря которым вы отточили собственные навыки, также способны помочь другим разработчикам.

Нам всем есть чему поучиться, мы все способны чему-то научить. Когда мы делимся полученными сведениями с окружающими, то совершенствуем все сообщество Java. Это жизненно важно для того, чтобы мы сами и наши коллеги-разработчики на Java совместно улучшали наши навыки программирования.

Научитесь любить ваш устаревший код

Уберто Барбини

Что такое устаревшая система? Это старое программное обеспечение, которое очень сложно поддерживать, расширять и улучшать. С другой стороны, подобная система также работает и служит бизнесу, в противном случае она бы не выжила.

Возможно, когда ее только создали, ныне устаревшая система была спроектирована настолько удачно, что люди начали говорить: «Хорошо, а не использовать ли ее также для вот этого, и этого, и этого». Она становится перегруженной техническим долгом, но все равно работает. Такие системы бывают удивительно устойчивыми.

Тем не менее разработчики ненавидят иметь дело с устаревшими системами. Как порой кажется, на них столько технических долгов, что их никто никогда не сумеет погасить. Возможно, нам следует просто объявить о банкротстве и двигаться дальше. Так гораздо проще.

Но что, если вам действительно нужно поддерживать устаревший код? Что вы делаете, когда вам необходимо устранить ошибку?

Решение номер один: клейкая лента. Зажмите нос, исправьте дефект — «хорошо, может, однажды мы об этом пожалеем, но сейчас давайте скопируем-вставим эту часть кода, просто чтобы исправить баг». Дальше будет только хуже. Так, заброшенное здание порой очень долго стоит невредимым, но, если там вышибут хоть одно стекло, вскоре в нем не останется ни единого целого окна. Просто сам вид выбитого стекла побуждает людей к действию. Закон разбитых окон.

Решение номер два: забудьте о старой системе и перепишите ее с нуля. Догадаетесь сами, в чем проблема с этим вариантом? Чаще всего новая система не будет работать или ее не удастся завершить. Причина здесь в ошибке выжившего (*https://oreil.ly/lKSDd*). Вы видите старый системный код и говорите: «О, да ладно, если уж тот, кто написал этот ужасный код, заставил его

работать, то задачка довольно простая». Но это не так. Вы можете счесть код ужасным, однако он ведь уже пережил много сражений. Когда вы начинаете с нуля, то не знаете истории тех битв и утратили множество знаний об этой области.

Так что же нам делать? В Японии есть искусство, называемое *кинцуги* (*https://oreil.ly/F4AZX*). Когда ломается ценный предмет, его не выбрасывают, а восстанавливают, заполняя трещины лаком с золотым порошком. Золото подчеркивает, что вещь разбита, но все равно красива.

Возможно, мы смотрим на устаревший код с неправильной точки зрения? Я не говорю, что мы должны позолотить старый код, но стоит научиться исправлять его так, чтобы потом гордиться результатом.

Паттерн strangler (*https://oreil.ly/SWJFc*) позволяет нам делать именно это. Он назван в честь фикуса-душителя (*https://oreil.ly/jficR*) (а не удушения!), который обвивается вокруг других деревьев. В процессе роста он постепенно опутывает носителя, который увядает, пока не останутся только стебли фикуса вокруг полой сердцевины.

Аналогичным образом мы начинаем с того, что заменяем зловонную строку кода новой, чистой, тщательно протестированной. А затем, отталкиваясь от этого, создаем новое приложение, которое ползет поверх предыдущего, пока не заменит его полностью.

Но даже если мы не завершим работу, гораздо лучше совместить новое и старое, чем просто дать старому сгнить. Это намного безопаснее, чем разработка с нуля, потому что мы будем постоянно проверять новое поведение и всегда сможем откатить последнюю версию, если внесем ошибки.

Устаревший код заслуживает капельку любви.

Научитесь использовать новые функции Java

Гейл С. Андерсон

Версия Java 8 представила лямбды и потоки, две революционные функции, которые дают Java-программистам значимые языковые конструкции. Начиная с Java 9 и далее циклы выпуска происходят каждые шесть месяцев и в каждом релизе появляется все больше новых функций. Вы должны узнавать о них, потому что они помогают вам писать более качественный код. И ваши навыки станут улучшаться по мере того, как вы будете пополнять ваш арсенал программирования новыми языковыми парадигмами.

Уже многое написано о потоках и о том, как они поддерживают функциональный стиль программирования, уменьшают объем громоздкого кода и делают код более читаемым. Итак, давайте рассмотрим пример с потоками: не столько для того, чтобы убедить вас использовать потоки везде, сколько для того, чтобы побудить вас узнать о них и других функциях Java, появившихся с момента выпуска Java 8.

В нашем примере вычисляются максимальное, среднее и минимальное значения систолического, диастолического и пульсового параметров артериального давления на основе показателей, собранных при мониторинге. Мы хотим визуализировать результаты вычислений этих сводных статистических данных с помощью столбчатой диаграммы JavaFX.

Вот часть нашего класса `BPData`, где показаны только геттеры, которые нам нужны:

```
public class BPData {
  ...
  public final Integer getSystolic() {
    return systolic.get();
  }
  public final Integer getDiastolic() {
    return diastolic.get();
```

```
  }
  public final Integer getPulse() {
    return pulse.get();
  }
  ...
}
```

Идеальным решением для этой визуализации служит столбчатая диаграмма JavaFX. Сначала нам нужно построить правильный ряд и передать наши преобразованные данные в объект гистограммы. Поскольку операция повторяется для каждой серии, имеет смысл создать единый метод для параметризации как серии гистограмм, так и конкретного средства получения `BPData`, необходимого для доступа к этим данным. Исходные сведения хранятся в переменной `sortedList`, отсортированной по дате коллекции элементов `BPData`. Перед вами метод `computeStatData`, который строит диаграммы данных:

```
private void computeStatData(
        XYChart.Series<String, Number> targetList,
        Function<BPData, Integer> f) {
    // Задать максимум
    targetList.getData().get(MAX).setYValue(sortedList.stream()
        .mapToInt(f::apply)
        .max()
        .orElse(1));
    // Задать среднее
    targetList.getData().get(AVG).setYValue(sortedList.stream()
        .mapToInt(f::apply)
        .average()
        .orElse(1.0));
    // Задать минимум
    targetList.getData().get(MIN).setYValue(sortedList.stream()
        .mapToInt(f::apply)
        .min()
        .orElse(1));
}
```

Параметр `targetList` — это данные серии гистограмм, которые соответствуют одному из параметров (систолическому, диастолическому или пульсовому давлению). Мы хотим создать столбчатую диаграмму с максимумом,

средним значением и минимумом для каждой из этих серий. Соответственно, мы задаем Y-значения диаграммы по результатам соответствующих вычислений. Второй параметр — это конкретный геттер из `BPData`, передаваемый в качестве ссылки на метод. Мы используем его в потоковом методе `mapToInt` для доступа к конкретным значениям для этого ряда. Каждая последовательность потоков возвращает максимальное, среднее или минимальное значение исходных данных. Каждый метод завершающего потока возвращает `orElse` и объект `Optional`, в результате чего на нашей гистограмме отображается значение-заполнитель, равное 1 (или 1.0), если исходный поток данных пуст.

Вот как вызвать данный метод `computeStatData`. Удобная нотация ссылок на методы позволяет легко указать, какой метод получения `BPData` вызывать для каждого ряда данных:

```
computeStatData(systolicStats, BPData::getSystolic);
computeStatData(diastolicStats, BPData::getDiastolic);
computeStatData(pulseStats, BPData::getPulse);
```

До появления Java 8 написание такого кода отнимало гораздо больше сил. Следовательно, изучение и использование новых функций Java — полезный навык, который стоит освоить, поскольку Java продолжает совершенствоваться.

Почему бы вам, работая над следующим продуктом, не освоить синтаксическую конструкцию Java 14 `record`, функцию предварительного просмотра, чтобы упростить класс `BPData`?

Изучите свою IDE, чтобы уменьшить когнитивную нагрузку

Триша Джи

Я сотрудница фирмы, которая продает IDE, поэтому, конечно, хотела бы сказать, что вы должны знать, как функционирует ваша IDE, и правильно ее использовать. Ранее я 15 лет работала с несколькими IDE, изучая, как они помогают разработчикам создавать нечто полезное и как применять их для легкой автоматизации задач.

Мы все знаем, что IDE обеспечивают подсветку кода и показывают ошибки и потенциальные проблемы, но любая Java IDE способна на гораздо большее. Если вы изучите возможности вашей IDE и будете использовать функции, применимые к повседневной работе, то сумеете повысить производительность. Например, ваша IDE:

- Может сгенерировать код, так что вам не придется его вводить. Самые распространенные примеры — геттеры и сеттеры, `equals` и `hashCode`, а также `toString`.
- Имеет инструменты рефакторинга, которые могут автоматически продвигать ваш код в определенном направлении, не досаждая при этом компилятору.
- Умеет запускать тесты и помогать с отладкой проблемы. Если вы используете для отладки `System.out`, это занимает у вас гораздо больше времени, чем при проверке значений объектов в ходе выполнения.
- Должна интегрироваться с вашей системой управления сборкой и зависимостями, чтобы среда разработки действовала так же, как ваши рабочая и тестовая среды.
- Способна даже помочь вам с инструментами или системами, внешними по отношению к коду вашего приложения, — например, контролем версий, доступом к базе данных или проверкой кода (помните, что *I*

в *IDE* означает *интегрированная*). Вам не нужно покидать окно IDE, чтобы работать со всеми аспектами вашего конвейера развертывания программного обеспечения.

Используя IDE, вы можете перемещаться по коду естественным образом — отыскивая методы, которые вызывают нужный фрагмент кода, или переходя к методу, вызывающему этот код. Вы можете перейти непосредственно к файлам (или даже к определенным фрагментам кода) всего лишь несколькими нажатиями клавиш, а не водя мышью для навигации по файловой структуре.

Инструмент, который вы выбираете для написания кода, обязан помогать сосредоточиться на том, что вы разрабатываете. Вы не должны думать о тонкостях того, как вы это пишете. Перекладывая утомительные задачи на среду IDE, вы снижаете свою когнитивную нагрузку и можете тратить больше умственных сил на бизнес-проблему, которую пытаетесь решить.

Давайте заключим контракт: искусство разработки Java API

Марио Фуско

API — это то, что разработчики используют для решения какой-либо задачи. Точнее, он формирует контракт между ними и проектировщиками программного обеспечения, предоставляя возможности своего ПО через этот API. В этом смысле мы все разработчики API: наше программное обеспечение не функционирует изолированно, а становится полезным только тогда, когда взаимодействует с другим ПО, написанным иными разработчиками. При написании программного обеспечения мы становимся не только потребителями, но и поставщиками одного или нескольких API, поэтому каждый разработчик обязан знать характеристики хороших API и то, как достичь таких показателей.

Во-первых, хороший API должен быть легко понятным и доступным для обнаружения. Нужно, чтобы люди смогли начать использовать его и, в идеале, узнать, как он работает, не читая документацию. Чтобы добиться этого, важно применять согласованные имена и соглашения, что звучит довольно очевидно. Тем не менее даже в стандартном Java API легко найти ситуации, когда этого предложения не придерживались. Например, поскольку вы можете вызвать `skip(n)`, чтобы пропустить первые *n* элементов потока, какое название подойдет методу, пропускающему *все* элементы потока, пока один из них не удовлетворит предикату `p`? Разумно предположить, что `skipWhile(p)`, но на самом деле этот метод именуется `dropWhile(p)`. В названии `dropWhile` как таковом нет ничего плохого, но оно не согласуется со `skip`, выполняющим очень похожую операцию. Не делайте так.

Еще один способ упростить использование вашего API — свести его к минимуму. Это сокращает как изучаемые концепции, так и затраты на их техническое обслуживание. Опять же, примеры нарушения столь простого принципа отыщутся в стандартном Java API. У `Optional` есть фабричный статический метод `of(object)`, который создает `Optional`, оборачивая переданный ему

объект. Кстати, использование фабричных методов вместо конструкторов — еще одна ценная практика, поскольку она обеспечивает большую гибкость: поступая таким образом, вы также можете вернуть экземпляр подкласса или даже `null`, когда метод вызывается с недопустимыми аргументами. К сожалению, `Optional.of` выкинет `NullPointerException` при вызове с аргументом `null`. Вряд ли вы ожидаете чего-то подобного от класса, призванного предотвращать `NullPointerExceptions` (NPEs). Это не только нарушает принцип наименьшего удивления — который также следует учитывать при проектировании вашего API, — но и требует введения второго метода `ofNullable`, возвращающего пустой `Optional` при вызове метода с аргументом `null`. Метод `of` имеет непоследовательное поведение, и, если бы его реализовали правильно, метод `ofNullable` удалось бы исключить.

Вот еще полезные советы, которые помогут вам улучшить ваш API: разбейте большие интерфейсы на более мелкие части; рассмотрите возможность реализации свободного API (на этот раз JavaStreams послужит хорошим примером); никогда не возвращайте `null`, вместо этого применяйте пустые коллекции и `Optional`; ограничьте использование исключений и, по возможности, избегайте проверяемых исключений. Что касается аргументов метода: избегайте длинных списков аргументов, особенно однотипных; используйте самый слабый возможный тип; сохраняйте их в согласованном порядке среди различных перегрузок; рассмотрите varargs. Более того, тот факт, что хороший API не требует пояснений, не означает, что вы не должны четко и подробно документировать его.

Наконец, не ждите, что вы создадите отличный API с первого раза. Разработка API — это итеративный процесс, и создание тестовых версий — единственный способ проверять и улучшать продукт. Напишите тесты и примеры для вашего API и обсудите их с коллегами и пользователями. Повторите несколько раз, чтобы устранить неясные цели, избыточный код и излишнюю абстракцию.

Делайте код простым и читабельным

Эмили Цзян

Я большой поклонник простого и читаемого кода. Каждая строка должна выглядеть как можно понятнее. Каждая строка обязана быть востребованной. Чтобы получить читаемый и незамысловатый код, обратите внимание на два аспекта — формат и содержание. Вот несколько советов, которые помогут вам написать понятный и простой код.

Четко форматируйте ваш код с помощью отступов.

Применяйте их последовательно. Если вы работаете в проекте, там должен иметься шаблон кода. Все члены команды обязаны использовать одно и то же форматирование. Не путайте пробелы с табуляцией. У меня всегда есть IDE, настроенная на отображение пробелов и табуляций, чтобы я могла найти и исправить их. (Лично я люблю пробелы.) Выберите либо пробелы, либо табуляцию и придерживайтесь своего решения.

Применяйте говорящие имена переменных и имена методов.

Код намного проще поддерживать, если он понятен сам по себе. Если в вашем коде есть говорящие идентификаторы, то он сам обо всем расскажет и вам не понадобится отдельная строка комментария, чтобы объяснить его функционал. Избегайте однобуквенных имен переменных. Если названия ваших переменных и методов понятны и значимы, то, как правило, вам не нужно разъяснять в комментариях, что именно делает ваш код.

При необходимости прокомментируйте ваш код.

Если логика очень сложная, например запросы с регулярными выражениями и т. д., поясните в документации, что пытается сделать код. Когда добавляете комментарии, обязательно убедитесь, что они поддерживаются. Неподдерживаемые комментарии вызывают путаницу. Если вам нужно предупредить других программистов о чем-то, убедитесь, что вы задокументировали это и выделили, например, добавив WARNING перед

телом комментария. Иногда ошибку легче обнаружить и исправить, если первоначальный автор кода рассказывает о своем замысле или помещает где-нибудь предупреждение.

Не проверяйте закомментированный код.

Удалите его, чтобы улучшить читаемость. Одним из распространенных аргументов в пользу закомментированного кода является то, что когда-нибудь он может понадобиться. На самом же деле он может висеть там годами, необслуживаемый и вызывающий путаницу. Даже если однажды вы захотите раскомментировать его, возможно, блок кода не скомпилируется или заработает не так, как ожидалось, поскольку остальная кодовая база значительно изменилась. Не сомневайтесь. Просто удалите его.

Не переусердствуйте, когда добавляете код, который может стать полезным в будущем.

Если вам поручено предоставить некоторую функциональность, не переусердствуйте. Не включайте дополнительную спекулятивную логику, не пытайтесь угадывать, что понадобится в будущем. Любой добавочный код чреват ошибками и накладными расходами на техническое обслуживание.

Избегайте написания подробного кода.

Стремитесь к тому, чтобы решить поставленную задачу, написав как можно меньший объем кода. Чем больше строк, тем больше ошибок. Сначала создайте прототип посредством мозгового штурма, чтобы выполнить задачу, а затем отшлифуйте код. Убедитесь, что у каждой строки есть веская причина для существования. Если вы менеджер или архитектор, судите своих разработчиков не по количеству строк, которые они предоставляют, а по тому, насколько чист и удобочитаем их код.

Изучите функциональное программирование, если вы еще этого не сделали.

Одним из преимуществ использования функций, представленных в Java 8 (например, лямбд и потоков), состоит в том, что они способны улучшить читаемость вашего кода.

Внедрите парное программирование.

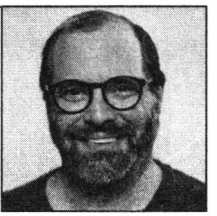
Парное программирование — отличный способ для младшего разработчика поучиться у кого-то более опытного. Также это прекрасно поможет написать осмысленный код, поскольку вам нужно будет объяснить свой выбор и рассуждения другому человеку. Такой замечательный

процесс побуждает вас составлять код тщательно, а не вываливать кучу строк.

В коде будет меньше ошибок, если он прост и удобочитаем. В громоздком коде, скорее всего, будет больше ошибок. В коде, который нелегко понять, скорее всего, будет больше ошибок. Надеюсь, благодаря этим советам вы улучшите ваши навыки написания кода так, что сможете создавать простой и читаемый код!

Добавьте в вашу Java немного Groovy

Кен Коузен



Экран был цвета киберпанковского романа, открытого на первой строке. Я пристально глядел на него, переживая, что сегодня уже ничего не закончу. В стенку моей секции постучали. Моя начальница стояла и ждала.

— Как дела? — спросила она.

— Java так многословна, — вздохнул я. — Мне просто надо загрузить некоторые данные из сервиса и сохранить их в БД. Но я плаваю в конструкторах, фабриках, библиотечном коде, блоках `try/catch`...

— Просто добавь Groovy.

— А? Как это может помочь?

Она присела рядом.

— Не возражаешь, если я порулю?

— Прошу.

— Позволь провести тебе небольшую демонстрацию. — Запустив командную строку, она набрала `groovyConsole`. На экране появился простой графический интерфейс (GUI). — Допустим, мы хотим знать, сколько астронавтов в данный момент находится в космосе. На Open Notify (*https://oreil.ly/oysGk*) есть сервис, который дает такую возможность.

Она ввела в консоли Groovy следующую команду:

```
def jsonTxt = 'http://api.open-notify.org/astros.json'.toURL().text
```

Ответ вернулся в формате JSON с количеством астронавтов, сообщением о состоянии и вложенными объектами, связывающими каждого астронавта с его кораблем.

— Groovy добавляет `toURL` к `String`, чтобы сформировать `java.net.URL`, и `getText` к `URL`, чтобы извлекать данные `text`, к которым ты получаешь доступ.

— Мило, — сказал я. — Теперь я должен сопоставить это с классами Java и использовать какую-нибудь библиотеку вроде Gson или Jackson…

— Не-а. Если все, что тебе нужно, — это количество людей в космосе, просто используй `Json Slurper`.

— Что?

Она напечатала:

```
def number = new JsonSlurper().parseText(jsonTxt).number
```

— Метод `parseText` возвращает `Object`, — продолжила она, — но нас здесь не волнует тип, так что просто переходи к деталям.

Оказалось, что на орбите находятся шесть человек, все на борту Международной космической станции.

— Хорошо, — сказал я. — Допустим, я хочу разобрать ответ на классы. Что тогда? Есть ли порт Gson для Groovy?

Она покачала головой.

— Тебе это не нужно. Это все байт-коды «под капотом». Просто создай экземпляр класса `Gson` и вызывай методы, как обычно:

```
@Canonical
class Assignment { String name; String craft }
@Canonical
class Response { String message; int number; Assignment[] people }
new Gson().fromJson(jsonTxt, Response).people.each { println it }
```

— Аннотация `Canonical` добавляет `toString`, `equals`, `hashCode`, конструктор по умолчанию, именованный конструктор аргументов и конструктор кортежей для каждого класса.

— Потрясающе! А теперь как мне сохранить астронавтов в базе данных?

— Достаточно просто. Давай используем H2 для этого примера:

```
Sql sql = Sql.newInstance(url: 'jdbc:h2:~/astro',
                          driver: 'org.h2.Driver')
sql.execute '''
 create table if not exists ASTRONAUTS(
   id int auto_increment primary key,
   name varchar(50),
   craft varchar(50)
 )
'''
response.people.each {
 sql.execute "insert into ASTRONAUTS(name, craft)" +
             "values ($it.name, $it.craft)"
}
sql.close()
```

— Класс `Sql` в Groovy создает таблицу с использованием многострочной переменной string и вставляет значения с применением интерполированных строк:

```
sql.eachRow('select * from ASTRONAUTS') {
 row -> println "${row.name.padRight(20)} aboard ${row.craft}"
}
```

— Готово, — подытожила она, — с отформатированным выводом и всем прочим.

Я уставился на результат.

— Ты хоть представляешь, сколько строк Java это заняло бы? — спросил я.

Она ухмыльнулась.

— Много. Кстати, все исключения в Groovy непроверяемые, так что тебе даже не нужен блок `try/catch`. Если мы используем `withInstance` вместо `newInstance`, соединение тоже автоматически закроется. Достаточно удобно?

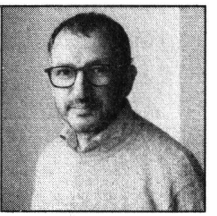
Я кивнул.

— Теперь просто оберни разные части в класс, и можно вызвать их из Java.

Она ушла, а я уже предвкушал, как добавлю Groovy в остальные фрагменты моего кода Java.

Минимизируйте конструкторы

Стив Фримен



Регулярно вижу такую закономерность: в конструкторе выполняется значительная работа — берется набор аргументов, которые преобразовываются в значения для полей. Зачастую это выглядит так:

```
public class Thing {
    private final Fixed fixed;
    private Details details;
    private NotFixed notFixed;
    // еще поля

    public Thing(Fixed fixed,
                 Dependencies dependencies,
                 OtherStuff otherStuff) {
        this.fixed = fixed;
        setup(dependencies, otherStuff);
    }
}
```

Полагаю, `setup` инициализирует остальные поля, основанные на `dependencies` и `otherStuff`, но мне не ясно из сигнатуры конструктора, какие именно значения необходимы для создания нового экземпляра. Также неочевидно, какие поля способны изменяться в течение срока службы объекта, поскольку они не могут быть `final`, если не инициализированы в конструкторе. Наконец, модульное тестирование этого класса неоправданно сложно, поскольку для его инициализации требуется создать правильную структуру в передаваемых аргументах `setup`. Хуже того, иногда мне попадаются такие вот конструкторы:

```
public class Thing {
    private Weather currentWeather;
    public Thing(String weatherServiceHost) {
```

```
        currentWeather = getWeatherFromHost(weatherServiceHost);
    }
}
```

Они требуют сервис и подключение к интернету для создания экземпляра. К счастью, сейчас такие конструкторы встречаются редко.

Все это делалось с благими намерениями, с целью упростить создание экземпляров путем «инкапсуляции» поведения. Я считаю, что этот подход унаследован от C++, где программисты могут творчески использовать конструкторы и деструкторы для управления ресурсами. Классы проще объединять в иерархии наследования, если каждый контролирует собственные внутренние зависимости.

Я предпочитаю использовать подход, основанный на моем опыте работы с Modula-3 (*https:// oreil.ly/t2t4G*). В данном случае конструктор только присваивает значения полям. Его единственная задача — создать действительный экземпляр. Если нужно сделать что-то еще, я использую фабричный метод:

```
public class Thing {
    private final Fixed fixed;
    private final Details details;
    private NotFixed notFixed;

    public Thing(Fixed fixed, Details details, NotFixed notFixed) {
        this.fixed = fixed;
        this.details = details;
        this.notFixed = notFixed;
    }

    public static Thing forInternationalShipment(
            Fixed fixed,
            Dependencies dependencies,
            OtherStuff otherStuff) {
        final var intermediate = convertFrom(dependencies, otherStuff);
        return new Thing(fixed,
                         intermediate.details(),
                         intermediate.initialNotFixed());
    }
```

```
    public static Thing forLocalShipment(Fixed fixed,
                                         Dependencies dependencies) {
        return new Thing(fixed,
                         localShipmentDetails(dependencies),
                         NotFixed.DEFAULT_VALUE);
    }
}

final var internationalShipment =
    Thing.forInternationalShipment(fixed, dependencies, otherStuff);
final var localShipment = Thing.forLocalShipment(fixed, dependencies);
```

Преимущества здесь таковы:

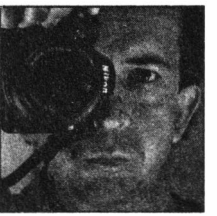
- Теперь я очень четко представляю себе жизненный цикл полей экземпляра.
- Я отделил код для создания экземпляра объекта от его использования.
- Имя фабричного метода описывает само себя, в отличие от конструктора.
- Упрощается отдельное тестирование класса и его создания.

Тут есть недостаток, поскольку теперь не удастся совместно использовать реализацию конструктора в иерархиях наследования, но недочет можно устранить, если сделать доступными поддерживающие вспомогательные методы. Еще более полезно будет, если вы догадаетесь избежать глубокого наследования.

Наконец, все это также побуждает меня проявлять аккуратность при работе с фреймворками внедрения зависимостей. Если создавать объект сложно, то помещать все содержимое в конструктор, потому что так упрощается использование инструментов рефлексии, кажется мне устаревшей практикой. Обычно вместо этого можно зарегистрировать фабричный метод как способ создания новых экземпляров. Аналогично, если вы используете рефлексию, чтобы присвоить значения private полей непосредственно для «инкапсуляции» (или потому, что не хотите писать конструктор), то нарушаете систему типов и усложняете модульное тестирование. Лучше присвоить значения поля с помощью минимального конструктора. Применяйте `@Inject` или `@Autowired` с осторожностью и делайте весь процесс явным.

Назовите дату

Кевлин Хенни



Пока класс `java.util.Date` медленно, но верно катится под горку на Sun-ках, а `java.time` занимает его место на вершине, стоит уделить немного времени тому, чтобы извлечь некоторые уроки из его беспокойной жизни, и лишь затем позволить ему упокоиться с миром.

Самый очевидный урок заключается в том, что обработка даты и времени сложнее, чем предполагают люди, — даже когда они предполагают, что будет сложно.

Общепризнанная истина состоит в том, что, если отдельный программист убежден, будто разбирается в датах и времени, его код нужно проверить. Но здесь я хочу сосредоточиться не на этом, и не на том, как важна неизменяемость для типов значений, и не на том, что делает класс (не)подходящим для создания подклассов, и не на том, как использовать классы, а не целые числа, чтобы выразить обширную область.

Исходный код состоит из интервалов, знаков препинания и имен. Все это передает читателю вложенный смысл, но большая его часть транслируется (или теряется) в названиях. Они очень важны. Очень.

Если у класса `Date` (Дата) такое название, было бы неплохо, если бы он соответствовал календарной дате, то есть определенному дню… но это не так. Он представляет собой момент времени, который можно рассматривать как компонент даты. Это понятие чаще обозначают как *дата-время* или, если вы хотите прибегнуть к коду, `DateTime`. `Time` тоже подойдет, поскольку это всеобъемлющая концепция. Иногда бывает трудно найти правильное название, но в данном случае это не так.

Теперь нам ясно, что мы подразумеваем под датой, датой-временем и `Date`, и встает вопрос — что делает `getDate`? Возвращает ли он полное значение даты и времени? Или, возможно, просто компонент даты? Ни то ни другое: он возвращает день месяца. В кругах программистов это значение чаще и конкретнее обозначается именно как *день месяца*, а не *дата* — термин, обычно зарезервированный для представления календарной даты.

И, не отходя далеко от темы, `getDay` правильнее было бы назвать `getDayOfWeek`. Важно не только правильно выбирать название, но и распознавать и устранять неясности в неоднозначных терминах, таких как *день* (чего — недели, месяца, года?..). Обратите внимание, что лучше решать проблемы с именованием, выбирая более подходящее название, чем прибегать к помощи Javadoc.

Названия привязаны к условностям, а условности привязаны к названиям. Когда дело доходит до соглашений, отдавайте предпочтение одному (не многим), предпочитайте выражаться четко и предпочитайте то, что широко известно и просто в использовании, а не нишево и подвержено ошибкам (да, C, я смотрю на тебя).

Например, «Аполлон-11» прилунился в 20:17 в двадцатый день июля (седьмого месяца) 1969 года (CE, UTC и т. д.).

Но если вы вызовете `getTime`, `getDate`, `getMonth` и `getYear`, ожидая получить эти цифры, вас ждет разочарование: `getTime` возвращает отрицательное число миллисекунд от начала 1970 года; `getDate` вернет 20 (как и ожидается, он считает с 1); `getMonth` вернет 6 (месяцы считаются с 0); `getYear` вернет 69 (точка отсчета лет — 1900, а не 0 или 1970).

Хорошее название — это часть проекта. Оно задает ожидания и разъясняет модель, показывая, как что-либо следует понимать и использовать. Если вы хотите сказать читателю `getMillisSince1970`, не говорите `getTime`. Стремление давать конкретные названия побуждает вас рассматривать альтернативы, спрашивать себя, верно ли вы уловили смысл абстракции и правильна ли она сама. Это не просто наклеивание ярлычков, и дело не только в `java.util.Date` — речь идет о коде, который вы пишете, и коде, который вы используете.

Необходимость технологий промышленной прочности

Пол У. Гомер

Java, возможно, называли «преемником COBOL», но это не обязательно плохо.

COBOL был невероятно успешной технологией. Надежный, последовательный и легко читаемый, он тянул на себе информационную эпоху, управляя основной массой критически важных систем в мире. А если синтаксис требует большого количества дополнительного ввода, значит, тем больше людей участвуют в создании кода и задумываются над тем, что они пишут.

Модные пакеты программного обеспечения — крутая тема (и, поскольку большинство из них довольно незрелые, всегда есть чему поучиться), но для функционирования миру нужно надежное ПО промышленного уровня. Скорее всего, с новой хитроумной идиомой или слегка запутанной парадигмой очень весело играть, но они по определению окутаны неизвестностью. Мы одержимы поиском какого-нибудь магического способа просто щелкнуть пальцами и наворожить следующую систему корпоративного класса, но постоянно забываем, что, как более трех десятилетий назад сказал Фредерик Брукс-младший, подобные волшебные пули, серебряные или еще какие-нибудь, просто невозможны.

Чтобы решать реальные проблемы людей, нам не требуется очередная модная игрушка. Нам нужно подумать и потрудиться, чтобы полностью понять и запрограммировать надежные решения. Если системы работают только в солнечные дни, или их нужно переписывать каждый год или около того, то они не удовлетворяют нашим растущим потребностям в укрощении сложностей современного общества. Уже неважно, как действует система, если при сбое в ней происходит что-то непредсказуемое. Напротив, мы должны полностью инкапсулировать наши знания в надежные, многоразовые, перекомпозируемые компоненты и использовать их как можно дольше, чтобы справляться с натиском хаотичной стихии в наш текущий период истории. Если код не продержится долго, его, вероятно, не стоило писать.

Java как технология отлично подходит для этой цели. Она достаточно новая (то есть содержит современные языковые функции), но и достаточно зрелая (то есть надежная). Мы стали лучше организовывать большие кодовые базы, и, поскольку нам вполне хватает вспомогательных продуктов, инструментов и экосистем, мы можем уделять внимание реальным бизнес-проблемам, не тратя время на чисто технические. Java — мощный стек для отделения систем от их окружения, но при этом достаточно стандартный, что позволит вам найти опытных сотрудников. Даже если о ней не толкуют все подряд, то, по крайней мере, это очень надежная, стабильная платформа, на которой можно создавать системы, служащие десятилетиями. Похоже, она дает нам все то, чего мы все хотим и что требуется нам для текущих стремлений в области разработки ПО.

Инженерные решения не должны диктоваться модой. Разработка программного обеспечения как дисциплина основывается на знаниях и организации. Если вы не знаете, как будут вести себя части, то не можете гарантировать, что целое будет вести себя правильно. Если решение ненадежно, то оно, по сути, только усугубляет проблему, а не решает ее. Бывает забавно просто сколотить какой-нибудь код, который вроде как работает, но настоящие профессионалы создают то, что выживает в реальных условиях и не сбавляет оборотов.

Создавайте только те части, которые изменяются, и повторно используйте остальные

Дженн Стретер

Мы, Java-программисты, тратим много времени на ожидание запуска сборок. Часто это связано с тем, что мы делаем это неэффективно. Мы добьемся небольших улучшений, если изменим свое поведение. Например, можно запускать только подмодуль вместо всего проекта и не выполнять очистку перед каждой сборкой. Чтобы добиться более заметных результатов, нам нужно воспользоваться преимуществами кэширования сборки, предлагаемыми инструментами сборки, а именно Gradle, Maven и Bazel.

Кэширование сборки — это повторное использование результатов предыдущего запуска для минимизации количества шагов сборки (например, задач Gradle, целей Maven, действий Bazel), выполняемых во время текущего запуска. Можно кэшировать любой шаг сборки, который идемпотентен (то есть выдает один и тот же результат для заданного набора входных данных).

Результат компиляции Java, например, — это дерево файлов классов, сгенерированных компилятором Java, а входные данные представляют собой факторы, влияющие на создаваемые файлы классов, такие как сам исходный код, версия Java, операционная система и любые флаги компилятора. Когда условия выполнения и исходный код не меняются, на этапе компиляции Java каждый раз создаются одни и те же файлы классов. Таким образом, вместо выполнения шага компиляции средство сборки может искать в кэше любые предыдущие запуски с теми же входными данными и повторно использовать выходные данные.

Кэширование сборки не ограничивается компиляцией. Инструменты сборки определяют стандартные входные и выходные данные для других распространенных этапов сборки (например, статического анализа

и генерирования документации), а также позволяют нам настраивать входные и выходные данные для любого кэшируемого этапа сборки.

Этот тип кэширования особенно полезен для многомодульных сборок. В проекте с 4 модулями, каждый из которых имеет 5 шагов сборки, чистая сборка должна выполнить 20 шагов. Однако в большинстве случаев мы модифицируем исходный код только в одном модуле. Если никакие другие проекты не зависят от этого модуля, значит, нам нужно выполнить только шаги, следующие после генерации исходного кода. В данном примере их будет только 4, а выходные данные остальных 16 шагов можно извлечь из кэша, экономя время и ресурсы.

Инкрементная сборка Gradle, которую мы видим в выходных данных сборки как `UP-TO-DATE`, реализует кэширование сборки на уровне проекта. Локальный кэш, подобный тому, который встроен в Gradle и доступен как расширение для Maven, работает даже при изменении рабочих пространств, ветвей Git и параметров командной строки.

Дополнительные преимущества дает совместный эффект удаленного кэширования сборки, доступный в Gradle, Maven и Bazel. Один из распространенных вариантов использования удаленного кэширования — первая сборка после извлечения из удаленного репозитория управления версиями. Выполнив это действие, мы должны создать проект на нашей машине, чтобы воспользоваться преимуществами внесенных изменений. Но поскольку мы никогда не создавали эти изменения на нашей машине, они еще не находятся в нашем локальном кэше. Однако система непрерывной интеграции уже внесла эти изменения и загрузила результаты в общий удаленный кэш, поэтому мы получаем доступ к данным из удаленного кэша, экономя время, необходимое для выполнения этих шагов сборки локально.

Используя кэширование сборок в своих сборках Java, мы можем делиться результатами между нашими локальными сборками, агентами сервера CI и всей командой, в результате чего сборки у всех проходят быстрее и уменьшаются траты ресурсов на повторное вычисление одних и тех же операций.

Проекты с открытым кодом — это не волшебство

Дженн Стретер

Один из моих главных раздражителей — речи про то, что какая-нибудь технология, язык, инструмент сборки и т. д. «работают по волшебству». Если разговор о проекте с открытым исходным кодом, то я мысленно поправляю говорящего: «Я слишком ленив, чтобы разобраться в нем», и мне вспоминается Третий закон Кларка, гласящий, что «любая достаточно развитая технология неотличима от магии»*.

В эпоху современного интернета вам проще, чем когда-либо прежде, просмотреть справочные руководства и исходный код, чтобы узнать, как работает эта технология. У многих проектов с открытым исходным кодом — например, языка программирования Apache Groovy — есть сайт (в данном случае groovy-lang.org) со списком ресурсов, на которых вы можете найти документацию, справочники, баг-трекер и даже ссылки на сам исходный код.

Если вы в самом начале пути, руководства и учебные пособия — отличное подспорье для новичка. Если вы больше предпочитаете визуальные или практические уроки, многие платформы онлайн-обучения предлагают вводные курсы для постижения новых языков с помощью лабораторных работ, упражнений и групповой работы. Иногда их даже помещают в свободный доступ, чтобы о технологии узнало как можно больше людей.

Изучив базовый синтаксис и структуры данных и начав использовать их в своих собственных проектах, вы, скорее всего, столкнетесь с неожиданным поведением или даже ошибками. Рано или поздно это обязательно произойдет, какую бы экосистему вы ни выбрали, — просто таков мир, в котором мы живем. Сначала вам следует поискать средство отслеживания проблем

* Артур Кларк. «Черты будущего. Исследование пределов возможного». Лондон: Pan Books, 1973. (Пер. Я. Берлина, В. Колтового). Что ж, в информатике есть формальное определение «магия», которое относится к сокрытию деталей реализации с помощью абстракции, но большинство людей злоупотребляют этим термином, применяя его к любой технологии, которую им трудно понять.

вроде Jira или GitHub issues, чтобы узнать, возникают ли такие неурядицы еще у кого-нибудь. Если да, то, возможно, уже есть обходные пути, появилось исправление в более новой версии или известны сроки, когда эта проблема будет устранена.

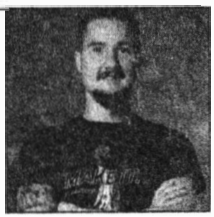
Порой надо немного потрудиться, чтобы выяснить, где взаимодействуют члены сообщества вашей технологии. Иногда это происходит в чатах, форумах или рассылках. В частности, для проектов фонда Apache, как правило, используется инфраструктура Apache, а не коммерческие продукты. Найти такое место — лучший способ забыть о «волшебстве» и прийти к пониманию.

Учтите, обучение — непрерывный процесс: даже после того как вы освоите ту или иную технологию, вам нужно будет и дальше познавать ее. Возможно, в новых версиях добавят новые функции или изменят поведение способами, которые вам потребуется понять. Чтобы узнать, что вам нужно для обновления ваших проектов, подписывайтесь на рассылки или посещайте конференции с разработчиками, вносящими свой вклад в проекты с открытым исходным кодом. Если вы уже стали экспертом в данной области, таким путем вы отлично поспособствуете тому, чтобы раскрыть «магию» для всех остальных.

Наконец, если вы заметите какую-то неясность или лакуну, многие проекты с радостью примут ваше содействие, особенно в части документации. Их руководители нередко заняты обычной повседневной работой и другими приоритетными делами, поэтому не всегда реагируют сразу, но это наилучший способ помочь всем добиться успеха и разъяснить «магию» для следующего поколения пользователей.

Optional — монада, нарушающая закон, но это хороший тип

Николай Парлог



В большинстве языков программирования типы, которые могут быть пустыми или непустыми, являются хорошо управляемыми монадами. (Да, я использовал слово на букву «М» — но не волнуйтесь, никакой математики.) Это означает, что их механика соответствует нескольким определениям и следует ряду законов, которые гарантируют безопасную (де)композицию вычислений.

Методы `Optional` соответствуют этим определениям, *но* нарушают законы. Не без последствий...

Определение монад

Чтобы определить монаду в терминах `Optional`, вам нужны три вещи:

1. Сам тип `Optional<T>`.
2. Метод `ofNullable(T)`, который оборачивает значение `T` в `Optional<T>`.
3. Метод `flatMap(Function<T, Optional<U>>)`, применяющий данную функцию к значению, обернутому `Optional`, для которого этот метод вызывается.

Существует альтернативное определение, использующее `map` вместо `flatMap`, но оно слишком длинное, чтобы уместить его здесь.

Законы монад

Переходим к интересному моменту — монада должна выполнять три закона, чтобы ее приняли в сообщество крутых ребят.

В терминах Optional:

1. Для Function<T, Optional<U>> f и значения v, f.apply(v) должны соответствовать Optional.ofNullable(v).flatMap(f). Благодаря такой *левой идентификации* не имеет значения, применяете вы функцию напрямую или позволяете это делать Optional.
2. Вызов flatMap(Optional::ofNullable) возвращает Optional, равный тому, для которого вы его вызвали. Эта *правая идентификация* гарантирует, что применение пустой операции ничего не изменит.
3. Для Optional<T> o и двух функций Function<T, Optional<U>> f и Function<U, Optional<V>> g результаты o.flatMap(f).flat Map(g) и o.flatMap(v -> f.apply(v).flatMap(g)) должны быть равны. Благодаря этой *ассоциативности* неважно, отображаются функции в виде flat-mapped по отдельности или в виде структуры.

Хотя Optional чаще всего сохраняется, это не относится к конкретному крайнему случаю. Взгляните на реализацию flatMap:

```
public <U> Optional<U> flatMap(Function<T, Optional<U>> f) {
   if (!isPresent()) {
       return empty();
   } else {
       return f.apply(this.value);
   }
}
```

Как видите, он не применяет функцию к пустому Optional, что позволяет легко разбить левую идентификацию:

```
Function<Integer, Optional<String>> f =
   i -> Optional.of(i == null ? "NaN" : i.toString());
// следующие значения не равны
Optional<String> containsNaN = f.apply(null);
Optional<String> isEmpty = Optional.ofNullable(null).flatMap(f);
```

Не очень хорошо, но для map это еще хуже. Здесь ассоциативность означает, что при заданном Optional<T> o и двух функциях Function<T, U> f и Function<U, V> g, результаты o.map(f).map(g) и o.map(f.andThen(g)) должны быть равны:

```
Function<Integer, Integer> f = i -> i == 0 ? null : i;
Function<Integer, String> g = i -> i == null ? "NaN" : i.toString();
// следующие значения не равны
Optional<String> containsNaN = Optional.of(0).map(f.andThen(g));
Optional<String> isEmpty = Optional.of(0).map(f).map(g);
```

Ну так что?

Возможно, примеры покажутся надуманными, а важность законов неясной, но их воздействие осязаемо: в цепочке `Optional` нельзя механически объединять и разделять операции, поскольку это может изменить поведение кода. Это прискорбно, потому что правильные монады позволяют вам игнорировать их, когда требуется сосредоточиться на удобочитаемости или бизнес-логике.

Так почему же `Optional` — неправильная монада? *Потому что* `null`-*безопасность важнее!* Чтобы соблюдать законы, `Optional` следовало бы содержать `null`, будучи непустым. И ему полагалось бы передавать его функциям, заданным `map` и `flatMap`. Представьте, что получилось бы, если бы все, что вы сделали в `map` и `flatMap`, требовалось проверять на `null`! *Такой* класс `Optional` был бы отличной монадой, но совершенно не обеспечивал бы `null`-безопасность.

Нет, я рад, что `Optional` такой, какой он есть.

Упаковка по функциям с модификатором доступа по умолчанию

Марко Билен

Многие бизнес-приложения написаны с применением трехуровневой архитектуры: визуальной части, бизнес-логики и уровня данных, и все объекты модели используются всеми тремя ярусами.

Имеются кодовые базы, где классы для этих приложений организованы по уровням. В отдельных приложениях, которым необходимо регистрировать различных пользователей и компанию, где они работают, структура кода приведет к появлению чего-то вроде:

```
tld.domain.project.model.Company
tld.domain.project.model.User
tld.domain.project.controllers.CompanyController
tld.domain.project.controllers.UserController
tld.domain.project.storage.CompanyRepository
tld.domain.project.storage.UserRepository
tld.domain.project.service.CompanyService
tld.domain.project.service.UserService
```

При использовании такой *упаковки по слоям* для ваших классов требуется, чтобы множество методов были общедоступными. `UserService` должен уметь считывать и записывать `User` в хранилище и, поскольку `UserRepository` находится в другом пакете, почти все методы `UserRepository` следует делать публичными.

Возможно, в организации принята политика, согласно которой пользователю при изменении пароля отправляется электронное письмо с уведомлением. Такую политику возможно реализовать в `UserService`. Однако, поскольку методы в `UserRepository` общедоступны, нет защиты от того, что другая часть приложения вызовет в `UserRepository` метод, который изменит пароль, но не запустит отправку уведомления.

Когда это приложение будут обновлять, чтобы добавить какой-либо модуль обслуживания клиентов или веб-интерфейс обслуживания, некоторые функции в этих модулях могут потребовать сброса пароля. Поскольку данные функции встроены в более поздний момент времени — возможно, после того как в команду вошли новые разработчики, — у этих сотрудников может возникнуть желание получить доступ к `UserRepository` напрямую от `CustomerCareService` вместо того, чтобы вызывать `UserService` и отсылать уведомления. Язык Java предоставляет механизм, позволяющий избежать такой ситуации: модификаторы доступа (МД).

При использовании МД по умолчанию мы явно не объявляем модификатор доступа для класса, поля, метода и т. д. Переменная или метод, объявленные без какого-либо модификатора контроля доступа, доступны только для других классов в том же пакете. Это также называется *package-private (доступные только в текущем пакете)*.

Чтобы извлечь выгоду из этого механизма защиты доступа, базу кода надлежит организовать в иерархию пакетов по функциям. Вышеупомянутые классы теперь будут упакованы следующим образом:

```
tld.domain.project.company.Company
tld.domain.project.company.CompanyController
tld.domain.project.company.CompanyService
tld.domain.project.company.CompanyRepository
tld.domain.project.user.User
tld.domain.project.user.UserController
tld.domain.project.user.UserService
tld.domain.project.user.UserRepository
```

При такой организации ни один из методов в `UserRepository` не должен иметь МД `public`. Все они могут быть *package-private* и по-прежнему доступными для `UserService`. Методы из `UserService` могут иметь модификатор доступа `public`.

Ни один разработчик, строящий `CustomerCareService` в пакете `tld.domain.project.support`, не сумеет вызвать методы для `UserRepository`. Ему придется вызывать методы `UserService`. Таким образом, структура кода и модификаторы доступа помогают гарантировать, что приложение будет придерживаться политики отправки уведомления.

Эта стратегия организации классов в вашей кодовой базе поможет снизить количество связей в ней.

Продакшн — самое радостное место на земле

Джош Лонг

Продакшн — мое самое обожаемое место в интернете. Я люблю продакшн. *Вы* должны любить продакшн. Приходите как можно раньше и как можно чаще. Приведите детей. Приведите родных. Погода потрясающая. Это самое радостное место на земле. Даже лучше, чем Диснейленд!

Попасть туда не всегда легко, но поверьте мне: как только вы туда доберетесь, вам захочется остаться. Это как Маврикий. Вы *полюбите* его! Вот несколько советов, которые сделают ваше путешествие максимально приятным.

Езжайте по шоссе непрерывной поставки.

Нет более короткого пути в продакшн. Непрерывная поставка позволяет быстро и последовательно переходить от последнего коммита в Git к продакшену! В непрерывном конвейере поставки код одним плавным движением автоматически перемещается от разработчика к развертыванию и по каждому промежуточному этапу. Здесь помогают инструменты непрерывной интеграции вроде Travis CI или Jenkins, но старайтесь анализировать информацию, полученную в процессе производства. Метод пробных релизов снижает риск внедрения новой версии ПО в производство, так как в данном случае изменения медленно распространяются на небольшую группу пользователей. Подобную тонкую стратегию развертывания возможно автоматизировать на базе инструментов непрерывной поставки — например, Spinnaker от Netflix.

Продакшн умеет удивлять.

Будьте готовы! Сервисы точно дадут сбой. Не бросайте своих клиентов в беде. Укажите агрессивные тайм-ауты на стороне клиента. Далее, во многих технических дискуссиях основное внимание уделяется соглашениям об уровне обслуживания (SLA). Чтобы соответствовать нормам SLA,

используйте хеджирование сервисов — шаблон, при котором запускаются несколько идемпотентных вызовов к идентично настроенным экземплярам служб на отдельных узлах и отбрасываются все ответы, кроме самого быстрого. Срывы точно случатся. Используйте автоматические выключатели, чтобы вовремя реагировать на отказы системы и изолировать их. Например, в Spring Cloud есть автоматический выключатель Spring Cloud Circuit Breaker — абстракция, которая поддерживает мониторинг работающих и неработающих сервисов.

В продакшене никто не услышит крик вашего приложения.

Добейтесь наблюдаемости с самого начала. Продакшн — это оживленное место! Если все пойдет хорошо, у вас будет больше пользователей и спроса, чем вы сможете обслужить. По мере увеличения спроса расширяйте масштабы. Облачная инфраструктура, такая как Cloud Foundry, Heroku и Kubernetes, уже давно поддерживает горизонтальное масштабирование, предоставляя ансамблю узлов балансировщик нагрузки. Это особенно просто, если вы создаете микросервисы без состояния в 12-факторном стиле. Такая стратегия работает, даже если ваше приложение монополизирует другие ценные ресурсы, такие как потоки.

Ваш код не должен монополизировать потоки.

Потоки обходятся крайне дорого. Наилучшие способы решения этой проблемы — через совместную многопоточность — заключаются в подаче среде выполнения сигналов о том, когда она может выполнять работу с конечным набором реальных потоков ОС. Узнайте о таких подходах, как реактивное программирование, поддерживаемое Project Reactor (заметно распространенное в серверной части), Spring Webflux и RxJava (довольнотаки распространенное на Android). Если вы разобрались, как работает реактивное программирование, следующий закономерный шаг — освоить что-нибудь вроде сопрограмм Kotlin. Совместная многопоточность позволит вам увеличить количество поддерживаемых пользователей или разделить затраты на инфраструктуру.

Автономия — ключ к успеху.

Благодаря микросервисам небольшие команды, сосредоточенные на одном проекте, способны самостоятельно выпускать программное обеспечение в продакшн.

Девяносто процентов вашего приложения прозаичны.

Используйте такие фреймворки, как Spring Boot, чтобы сосредоточиться на конечных результатах продакшена, а не на поддерживающем коде.

Что, язык программирования Java не ваш конек… или кофеек? Экосистема JVM богата производительными альтернативными ЯП вроде Kotlin.

Устраните трения, связанные с переходом к продакшену. Избегайте того, что технический директор Amazon Вернер Фогельс называет «неразделенной тяжелой работой»*. Расчистите путь к продакшену, и люди захотят приходить пораньше и почаще. Их будет тянуть туда, как во фразе Антуана де Сент-Экзюпери про «безбрежное море»**.

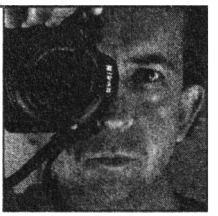
---

* Divina Paredes, "Amazon CTO: Stop Spending Money on 'Undifferentiated Heavy Lifting,'" *CIO*, June 9, 2013.

** «…мой народ… будут единым целым. Корабль строится не потому, что ты научил их шить паруса, ковать гвозди, читать по звездам; корабль строится тогда, когда ты пробудил в них страсть к морю…» Антуан де Сент-Экзюпери. Цитадель. Пер. М. Кожевниковой.

Программируйте с GUT

Кевлин Хенни



Итак, вы пишете модульные тесты? Отлично! Хороши ли они вообще? Заимствуя термин у Алистера Кокберна, надет ли на вас ХоМуТ*? Или из-за вас кому-то (например, вам в будущем) придется с процентами выплачивать технический долг, накопившийся в вашей технической базе?

Что я подразумеваю под *хорошим*? Отличный вопрос. Сложный вопрос. Заслуживает ответа.

Давайте начнем с названий. Отражайте в них то, что вы тестируете. Да-да, ваша схема наименований обойдется без чего-то вроде `test1`, `test2` и `test3`. По сути, вам даже не нужно слово `test` в названии тестов: `@Test` уже позаботился об этом. Скажите читателю, *что* вы тестируете, а не то, что *вы тестируете*.

Нет, я призываю давать названия в честь тестируемого метода. Укажите читателю, какое поведение, свойство, возможности и т. д. вы проверяете. При наличии метода `addItem` вам не требуется соответствующий тест `addItemIsOK`. Это классический тест «с душком». Определите возможные сценарии поведения и протестируйте только один случай для каждого теста. И нет, отсюда не следует, что нужны `addItemSuccess` и `addItemFailure`.

Позвольте спросить вас, в чем цель вашего теста? Проверить, что «это работает»? Тут все только начинается. Самая большая трудность при анализе кода заключается не в том, чтобы определить, «работает ли это», а в том, чтобы определить, что подразумевается под «это работает». У вас есть шанс уловить, в чем смысл, на примере `additionOfItemWithUniqueKeyIsRetained` и `additionOf ItemWithExistingKeyFails`.

Поскольку эти имена длинные, а также не являются кодом, который работает на продакшене, подумайте, не добавить ли символ подчеркивания для улучшения читаемости. Стиль CamelCase («верблюжий регистр») не масштабируется, поэтому получится `Addition_of_item_with_unique_key_is_retained`.

* XMT/GUT — хорошие модульные тесты. — *Прим. ред.*

В библиотеке JUnit 5 вы можете использовать `DisplayNameGenerator.ReplaceUnderscores` с `@DisplayNameGeneration` для красивого отображения имени как Addition of item with unique key is retained (Добавление элемента с уникальным ключом сохраняется корректно).

Как видите, если назвать тест осмысленной фразой-предположением, возникнет одно приятное свойство. Если тест пройден, вы вправе с определенной долей уверенности считать, что предположение верно. Если тест провалился, предположение ошибочно.

И это сильная сторона методики. Прохождение тестов само по себе не гарантирует, что код работает. Но хорошие модульные тесты характеризуются тем, что в них значение сбоя ясно показывает, что код не работает. Как сказал Дейкстра: «Тестирование программ может показать наличие ошибок, но никогда не покажет их отсутствие!»*

На практике это означает, что модульный тест не должен зависеть от того, что нельзя контролировать в рамках теста. Файловая система? Сеть? База данных? Асинхронный порядок? Возможно, вы воздействуете на них, но не контролируете. Тестируемый модуль не должен зависеть от того, что может вызывать сбой даже в случае правильности кода.

Кроме того, опасайтесь чрезмерно подробных тестов. Вы знаете, о чем речь: хрупкие утверждения о деталях реализации, а не о требуемых функциях. Вы что-то обновляете — орфографию, магическое значение, качественный результат, — и тесты проваливаются. Но происходит это потому, что неисправны тесты, а не рабочий код.

О, и еще берегитесь недостаточно подробных тестов. Они расплывчаты и проходят «по щелчку», даже если код невообразимо и откровенно ошибочен. Вот вы успешно добавили первый элемент. Не надо проверять, что теперь их количество просто «больше нуля». Все будет правильно лишь в том случае, если количество = 1. Многие целые числа больше нуля, поэтому миллиарды результатов будут ошибочны.

Кстати, о результатах. Возможно, вы обратили внимание, что многие тесты играют простую пьесу в трех актах: *организовать — действовать — утверждать*, иначе говоря, *дано — когда — тогда (given–when–then)*. Помня об этом, вы сумеете сосредоточиться на истории, которую пытается рассказать тест. Это поможет сохранить его связным, подкинет идеи о других тестах

* Эдсгер В. Дейкстра. «Заметки о структурном программировании» (в работе «Структурное программирование», O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, eds. (London and New York: Academic Press, 1972), 6).

и поспособствует выбору названия. О, и раз уж мы вернулись к названиям, то порой заметно, что имена становятся повторяющимися. Учитывайте повторение. Используйте его для группировки тестов во внутренние классы с помощью `@Nested`. Так вы сможете уместить `with_unique_key_is_retained` и `with_existing_key_fails` внутри `Addition_of_item`.

Надеюсь, заметка вам помогла. Решили пересмотреть какие-то тесты? Ладно, увидимся позже.

Ежедневно читайте OpenJDK

Хайнц М. Кабуц

OpenJDK содержит в себе миллионы строк Java-кода. Почти каждый класс нарушает некоторые рекомендации по «чистому коду». Реальный мир полон беспорядка. Фактически, «чистого кода» не существует — трудно даже дать определение, что это такое.

Опытные Java-программисты способны читать код, невзирая на то, каков его стиль. У OpenJDK более тысячи авторов. Несмотря на некоторую согласованность в форматировании, они пишут код по-разному.

Возьмем для примера метод `Vector.writeObject`:

```
private void writeObject(java.io.ObjectOutputStream s)
        throws java.io.IOException {
    final java.io.ObjectOutputStream.PutField fields = s.putFields();
    final Object[] data;
    synchronized (this) {
        fields.put("capacityIncrement", capacityIncrement);
        fields.put("elementCount", elementCount);
        data = elementData.clone();
    }
    fields.put("elementData", data);
    s.writeFields();
}
```

Почему программист пометил локальные переменные `fields` и `data` как `final`? В этом нет никакой необходимости. Он принял решение, исходя из стиля написания кода. Хорошие программисты успешно читают код вне зависимости от того, являются ли локальные переменные `final` или нет. Ни в том, ни в другом случае это их не беспокоит.

Почему `fields.put("elementData", data)` находится вне блока `synchronized`? Вероятно, дело в преждевременной оптимизации, желании сократить

дублирующую секцию кода. Или, возможно, программист проявил небрежность? Легко захотеть оптимизировать все, что мы видим, но нужно сопротивляться этому желанию.

Вот еще один метод из `Spliterator` внутри `ArrayList`:

```
public Spliterator<E> trySplit() {
    int hi = getFence(), lo = index, mid = (lo + hi) >>> 1;
    return (lo >= mid) ? null : // разделить диапазон пополам, если он не
слишком мал
            new RandomAccessSpliterator<>(this, lo, index = mid);
}
```

При виде такого метода поборники «чистого кода» наверняка забили бы во все колокола. Те, кто влюблен в `final`, будут жаловаться, что `hi`, `lo` и `mid` могли бы быть `final`. Да, могли бы. Но это не так. В OpenJDK локальные переменные обычно не помечаются как `final`.

А почему у нас вот это неясное «`(lo + hi) >>> 1`»? Не лучше ли указать «`(lo + hi) / 2`»? (Ответ: это не совсем одно и то же.)

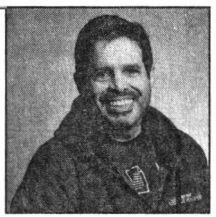
И почему все три локальные переменные объявлены в одной строке? Разве это не преступление против всего хорошего и правильного?

Оказывается, согласно исследованиям, количество ошибок пропорционально количеству строк кода (СК). Если вы распределите свой метод так, как просил ваш университетский профессор, вы получите больше СК. А чем больше у вас СК, тем больше ошибок при той же функциональности. Кроме того, иногда программисты-новички склонны разбрасывать свой код по многим классам. Эксперты пишут плотный, компактный код.

Нам нужно научиться читать множество различных стилей написания кода, и для этого я рекомендую OpenJDK. Читайте классы `java.util`, `java.io` и огромное множество других.

По-настоящему заглянуть «под капот»

Рафаэль Беневидес



Java — это полноценная платформа, и к ней следует относиться именно так. За свою карьеру Java-разработчика я встречал сотни коллег, хорошо знакомых с синтаксисом языка. Они понимают лямбды и потоки и назубок знают каждый API от `String` до `nio`. Но они станут всесторонними профессионалами, если также изучат следующие темы:

Алгоритмы сборки мусора

JVM GC значительно улучшился с момента его первых версий. Он автоматически настраивается на оптимальные параметры для обнаруженной среды. Если вы хорошо понимаете, как все работает, то иногда сможете еще больше повысить производительность JVM.

Профилировщики JVM

Настройка JVM — это не игра в угадайку. Вы должны разобраться, как ведет себя приложение, прежде чем вносить какие-либо изменения. Знания, как подключать и интерпретировать данные профилировщика, помогут вам настроить JVM для повышения производительности, обнаружить утечки памяти или понять, почему выполнение метода занимает так много времени.

Очевидно, что благодаря облачным приложениям код может выполняться на нескольких подключенных к сети компьютерах в разных операционных системах. Вероятно, профессионалам в Java, желающим разработать устойчивое и переносимое приложение, стоит изучить вот что:

Кодировка символов

В разных ОС могут применяться различные кодировки символов. Если вы понимаете, что это такое и как нужно настроить приложение, оно не будет выводить странные символы.

Сеть TCP/IP

Облачные приложения — это распределенные системы. В мире облачных вычислений, интернета и сетей важно понимать, как маршрутизировать таблицы, разбираться в задержках, брандмауэрах и всем, что связано с сетями TCP/IP, особенно когда что-то работает не так, как ожидалось.

Протокол HTTP

В мире, где браузер является клиентом, вы лучше спроектируете ваше приложение, если изучите, как действует HTTP. Вам может очень пригодиться знание того, что происходит, когда вы храните свои данные в сеансе HTTP, особенно в многокластерной среде.

Полезно знать даже то, что фреймворки делают «под капотом». Рассмотрим в качестве примеров фреймворки объектно-реляционного сопоставления (ORM), такие как JPA и Hibernate:

Включить вывод SQL во время разработки

При включенном выводе SQL вы увидите, что за команды отправляются в базу данных, еще до того, как заметите, что какой-то странный вызов SQL плохо себя ведет.

Размер выборки запроса

Большинство реализаций JPA/Hibernate имеют размер выборки по умолчанию, равный единице (1). Это означает, что, если ваш запрос возвращает 1000 объектов из базы данных, для заполнения этих объектов будет выполнено 1000 команд SQL. Настройте размер выборки, чтобы уменьшить количество выполняемых инструкций SQL. Эту проблему вы сможете выявить, если включите вывод SQL (см. предыдущий пункт).

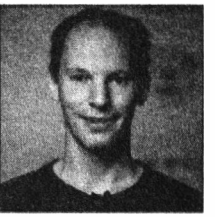
Отношения «один ко многим» и «многие к одному»

Хотя отношения «один ко многим» по умолчанию загружаются по требованию, некоторые разработчики совершают ошибку, меняя это поведение на обязательную загрузку или инициализируя их вручную перед возвратом коллекции объектов. Будьте осторожны, поскольку любая обязательно загруженная сущность также *может* создать *связь «многие к одному»*, что приведет к извлечению почти каждой таблицы/сущности из базы данных. Выявить эту проблему тоже проще, если у вас включен вывод SQL (опять же, см. первый пункт).

Короче говоря, не позволяйте себя контролировать — контролируйте сами!

Возрождение Java

Сандер Мак



Похоже, язык Java объявляли мертвым чаще, чем любой другой ЯП. Пожалуй, неудивительно, что сообщения о его смерти сильно преувеличены. Java занимает огромное место в серверной разработке, и большинство предприятий создают системы на Java. Тем не менее в каждом слухе есть доля правды — так, Java медленно развивается в эпоху динамических языков вроде Ruby и JavaScript. Традиционно основные релизы Java разрабатывались на протяжении трех-четырех лет. В таком темпе трудно угнаться за другими платформами.

Все изменилось в 2017 году. Компания Oracle, управляющая Java, объявила, что платформа Java будет выпускаться два раза в год. Java 9, появившаяся ближе к концу 2017 года, стала последним масштабным релизом, которого пришлось ждать долго. После девятой версии каждый год в марте и сентябре выходит новый значительный релиз Java. Как по часам.

Переход на календарный график выпуска повлек за собой много последствий. Релизы больше не могут ждать функций, которые еще не завершены. Кроме того, поскольку между выпусками проходит меньше времени, а команда, разрабатывающая Java, не увеличивается, в релиз попадает меньше функций. Но это нормально, ведь мы получим еще один релиз всего через шесть месяцев. Теперь мы можем рассчитывать на постоянный поток новых функций и улучшений.

Интересно, что новые языковые функции теперь также предоставляются постепенно. Язык Java в настоящее время развивается в духе agile. Например, Java 12 представила выражения Switch в качестве тестового релиза с возможностью оценить новый функционал: разработчики явно намерены позже расширить эту функцию для поддержки полного сопоставления с образцом.

Одна из причин того, что любой выпуск Java отнимал так много времени и усилий, заключается в том, что платформа несколько закоснела за 20 с лишним лет существования. В Java 9 она стала полностью модульной. Каждая часть платформы теперь помещается в собственный модуль с явными

зависимостями от других частей. Модульная система, представленная в Java 9, гарантирует, что отныне эта архитектура платформы будет соблюдаться.

Внутренние компоненты платформы теперь безопасно инкапсулированы внутри модулей, что предотвращает (чрезмерное) использование прикладного и библиотечного кода. Ранее многие приложения и библиотеки зависели от этих внутренних компонентов платформы, что затрудняло разработку Java без изменения большого количества существующего кода. Также модульную систему возможно использовать для ваших собственных приложений. Кроме того, с ней ваша кодовая база станет более удобной в обслуживании, гибкой и долговременно надежной.

Переход от длительного и непредсказуемого цикла выпуска к регулярным выпускам, основанным на календаре, — большое достижение команды Java. Безусловно, мы как сообщество разработчиков не сразу адаптировались к новому положению дел. К счастью, изменения в Java теперь менее масштабны и более постепенны. Такие более частые и регулярные выпуски легче принять, к ним проще адаптироваться.

Для тех, кто не торопится с переходами, каждые шесть выпусков, начиная с Java 11, выходит версия Java, помеченная как «поддерживаемая на длительный срок» (LTS — Long-Term Supported). Это означает, что вы можете переключаться между выпусками LTS лишь раз в три года, если хотите. Важно понимать, что обязательства LTS предлагаются такими поставщиками, как Oracle, Red Hat или даже Amazon, и не всегда бесплатны. В любом случае независимый от поставщика проект OpenJDK продолжает создавать поддерживаемые сборки для самой свежей разрабатываемой версии Java. Однако в релизах, появляющихся между выпусками LTS-версий, может поменяться многое. Если вам ничего не мешает, примите концепцию частых релизов и наслаждайтесь постоянным потоком улучшенной Java. Это не так страшно, как кажется.

Заново откройте для себя JVM с помощью Clojure

Джеймс Эллиотт

Где-то в 2007 году в моем офисном книжном клубе прочли «Параллелизм Java на практике» Брайана Гетца (издательство Addison-Wesley). Едва заглянув дальше предисловия к этой важной книге, мы запаниковали из-за того, как сильно заблуждались в нашем наивном понимании модели памяти Java и как легко ошибки внедряются в многопоточный код. Зазвучали шумные вздохи, и как минимум один читатель заявил, что ему привиделся кошмар.

При разработке облачного сервиса, способного обслуживать огромное количество одновременных запросов, нам требовался язык, который не засеивал бы нашу кодовую базу минами общего изменяемого состояния. Мы выбрали Clojure, так как он обладает надежными решениями для параллелизма и способствует функциональному и эффективному преобразованию неизменяемых данных. Он работает на привычной JVM, плавно взаимодействуя с огромной экосистемой библиотек Java. Хотя некоторые сомневались в незнакомом синтаксисе Lisp и не очень хотели заново учиться программировать без изменения переменных, решение оказалось отличным.

Мы обнаружили преимущества потока задач, ориентированного на REPL (read-eval-print loop — цикл чтения — оценки — печати):

- Никаких перестроек или перезапусков для тестирования изменений.
- Изучение работающей системы и мгновенное тестирование вариантов.
- Концепции можно создавать и совершенствовать постепенно.

Нам понравились склонность Clojure к работе с данными с использованием стандартных структур и его обильная, неприлично объемная базовая библиотека. Если вы хотите что-либо смоделировать, вам не нужно создавать множество классов, каждый с собственным несовместимым API-интерфейсом.

Я вновь начал программировать с радостью и энтузиазмом. Выступление на конференции Strange Loop о «написании кода на лету» для музыкальных

выступлений в Clojure с использованием Overtone (*https://oreil.ly/VcM79*) навело меня на такую мысль: если Clojure достаточно быстр, чтобы создавать музыку, наверняка у него получится управлять освещением сцены? Из подобных раздумий родился Afterglow (*https:// oreil.ly/L9wjF*), проект, поглотивший меня на некоторое время. Хотя пришлось потрудиться, чтобы понять, как описать световые спецэффекты в функциональном стиле, я нашел вдохновение для моих функций эффектов в функциональном метрономе Overtone, сопоставляя музыкальный размер с положением прожекторов, цветами и интенсивностью.

Я заново вызубрил тригонометрию и линейную алгебру, чтобы направлять разные источники света в одну и ту же точку пространства. Изучил, как создать желаемый цвет, используя светодиоды разных оттенков в прожекторе. Освещение сцены с «написанием кода на лету» — это очень весело.

Затем я захотел синхронизировать метроном Afterglow с треками, воспроизводимыми на CDJs (*https://oreil.ly/utaDV*) (современные цифровые вертушки для диджеев), которые я использую для микширования музыки. У них проприетарные и недокументированные протоколы, но это меня не остановило. Я настроил сетевой анализатор и разобрался в нем (*https://oreil.ly/FIIIk*). Успех раннего варианта обернулся восторженными отзывами и дополнениями со всего мира, поэтому я создал библиотеку Beat Link (*https://oreil.ly/fhvT2)*, чтобы облегчить применение полученных нами результатов. Я использовал Java, чтобы ее поняло как можно больше людей, но обнаружил, что при работе с Clojure программирование на Java ощущается трудоемким.

Люди развили мой проект, портировали его на другие языки. Я создал для одного театрального продюсера короткую демонстрацию использования Beat Link для запуска MIDI-событий, на которые могли бы реагировать его ПО для видео и консоль управления освещением. Этот проект популярнее всего прочего, что я делал, потому что он полезен для непрограммистов. Люди искусства по-прежнему непрерывно создают новые крутые штуки на Beat Link Trigger (*https://oreil.ly/JEK1H*), и я видел результаты, посещая в качестве гостя музыкальные фестивали и гастрольные концерты. Поскольку он создан на Clojure, пользователи могут расширять его, и их код компилируется в байтах и загружается в JVM, как будто входил в проект с самого начала. Вот еще одно секретное оружие, которое дает вам Clojure.

Я призываю всех, кто работает на Java, внимательно рассмотреть Clojure и понять, как он способен изменить ваши ощущения от жизни с JVM.

Преобразование логических значений в перечисления

Питер Хилтон

Вы бы не стали использовать «магические числа» в своем коде, так что и магические логические значения не применяйте! Логические литералы хуже жестко закодированных чисел: возможно, вы поймете, откуда появилось число `42`, но `false` может оказаться чем угодно, и что угодно может оказаться ложным.

Когда две переменные имеют значение `true`, вы не знаете, совпадение это или же они обе «истинны» по одной и той же причине и должны изменяться вместе. Это затрудняет чтение кода и приводит к ошибкам, когда вы читаете его неправильно. Как и в случае с магическими числами, вы обязаны провести рефакторинг и преобразовать их в перечисления.

Рефакторинг `42` к `ASCII_ASTERISK` или `EVERYTHING` улучшает читаемость кода, как и рефакторинг `true` к логической константе `AVAILABLE` в классе `Product`, например. Однако вам, пожалуй, не стоит иметь никаких логических полей в вашей модели предметной области, ведь некоторые логические значения на самом деле не являются логическими.

Предположим, что ваша сущность `Product` имеет логическое поле `available` (доступно), которое указывает, продается ли продукт в данный момент. Но на самом деле это не логическое значение, а необязательное значение «доступно», что не одно и то же, потому что «недоступно» в действительности означает что-то другое, например «нет в наличии».

Когда тип имеет два возможных значения, это совпадение, и его можно изменить — например, добавив вариант «снято с производства». Однако существующее логическое поле не может вместить дополнительное значение.

Будьте осторожны: использование `null` для обозначения чего-либо — наихудший из возможных способов реализации третьего значения. В итоге вам понадобится комментарий к коду, например такой: «true, когда продукт доступен, false, когда его нет в наличии, null, когда он снят с производства». Не делайте так.

Наиболее очевидная модель для отображения продуктов, которые вы больше не продаете, — логическое поле `discontinued` (снято с производства) в дополнение к полю `available`. Такой вариант работает, но его сложнее поддерживать, потому что нет никаких намеков на то, что эти поля связаны. К счастью, в Java есть способ группировать такие значения.

Преобразуйте связанные логические поля, подобные этим, в тип перечисления Java (*enum type*):

```
enum ProductAvailability {
  AVAILABLE, OUT_OF_STOCK, DISCONTINUED, BANNED
}
```

Перечисления хороши, потому что с ними вы можете назвать больше концепций. Кроме того, значения более удобочитаемы, чем `true`, которое на самом деле соответствует чему-то иному, например `AVAILABLE`. Кроме того, перечисления удобнее, чем вам, возможно, кажется, поэтому лень — слабое оправданием для отказа от рефакторинга.

Тип enum все же может включать в себя логические удобные методы, которые могут вам понадобиться, если в исходном коде встречалось много условных проверок доступных продуктов. На самом деле перечисления не ограничиваются группировкой констант: они распространяются на поля, конструкторы и методы. Менее очевидное, но более важное преимущество состоит в том, что теперь у вас есть место назначения для других рефакторингов, которые перемещают логику, связанную с доступностью, в тип `ProductAvailability`.

Для сериализации перечисляемого типа нужно поработать больше, чем при использовании JSON или базы данных, но меньше, чем вы ожидаете. Вероятно, вы уже применяете библиотеку, которая прекрасно справляется с этим и позволяет вам выбирать, как сериализовать представление объекта с одним значением.

Модели предметной области часто страдают от *одержимости примитивами* — чрезмерного использования примитивных типов Java. Рефакторинг чисел и дат в классы бизнес-логики улучшает выразительность и читаемость вашего кода, а новые типы обеспечивают связанный код более уютным домом, выложенным проверками и сравнениями.

Если говорить на языке проблемной области, логические типы ложны, а перечислимые типы истинны.

Рефакторинг для ускорения чтения

Бенджамин Мушкала

Обычный читатель достигает уровня в 150–200 сл./мин. (слов в минуту) с хорошим коэффициентом понимания. Люди, увлекающиеся скорочтением, могут легко развивать темп до 700 сл./мин. Но не волнуйтесь, нам не нужно устанавливать новый мировой рекорд, чтобы изучить основные концепции и применить их к нашему коду. Мы рассмотрим три области, которые особенно полезны, когда дело доходит до чтения кода: беглый просмотр, метаруководство и визуальная фиксация.

Как же добиться столь быстрого скорочтения? Один из первых этапов — подавить субвокализацию. Что-что, «субвокализация»? Именно. Тот голос в вашей голове, который только что пытался верно проговорить это слово. И да, теперь вы осознаёте, что он звучит. Но не волнуйтесь, это скоро пройдет! Отучившись от субвокализации, вы сделаете важный первый шаг к тому, чтобы серьезно повысить скорость чтения.

Давайте рассмотрим некий метод с тремя параметрами, все из которых нуждаются в проверке. Один из способов прочитать код — проследить, где и как используются входные параметры:

```
public void printReport(Header header, Body body, Footer footer) {
  checkNotNull(header, "header must not be null");
  validate(body);
  checkNotNull(footer, "footer must not be null");
}
```

После определения места `header` нам необходимо найти следующий параметр `body`, для чего требуется посмотреть вниз и налево. Можно начать с простого рефакторинга, чтобы поставить рядом первую и третью проверки. Теперь горизонтальный поток прерывается только один раз:

```
public void printReport(Header header, Body body, Footer footer) {
  checkNotNull(header, "header must not be null");
  checkNotNull(footer, "footer must not be null");
  validate(body);
}
```

Поскольку проверка на *null* — тоже проверка параметра, в качестве альтернативы мы могли бы извлечь вызовы метода `checkNotNull` в их собственные методы с правильными именами, чтобы помочь читателю. Являются ли они одной и той же или перегруженной версией метода, зависит от используемого кода:

```
public void printReport(Header header, Body body, Footer footer) {
  validateReportElement(header);
  validateReportElement(body);
  validateReportElement(footer);
}
```

Еще один метод скорочтения — метанаведение. Вместо того чтобы пытаться читать в книге одно слово за другим, пробуйте охватить всю строку сразу. Дети обычно ведут по тексту пальцем, чтобы отслеживать, на каком они месте. Применяя какие-либо ориентиры, мы сможем постоянно двигаться вперед, не отступая на пару слов назад. Как ни странно, сам код способен выступать в качестве такого механизма, поскольку имеет внутреннюю структуру, с помощью которой мы указываем путь нашему взгляду:

```
List<String> items = new ArrayList<>(zeros);
items.add("one");
items.add("two");
items.add("three");
```

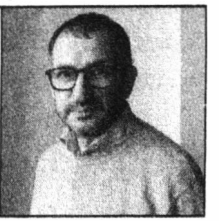
Сколько элементов в списке? Один, два, три!.. На самом деле четыре. А может, и больше. Ой, тоже пропустили аргумент `zeros`? Структура, которая должна нам помогать, в действительности встала у нас на пути. Предложив читателю двигаться взглядом по выравниванию методов `add`, мы устроили зрительную иллюзию и помешали заметить аргумент конструктора. Но читатель с легкостью пройдет по проложенному пути и не упустит никакой важной информации, если переписать код так:

```
List<String> items = new ArrayList<>();
items.addAll(zeros);
items.add("one");
items.add("two");
items.add("three");
```

В следующий раз, когда будете писать фрагмент кода, посмотрите, сможете ли вы ускорить его чтение. Учтите основы визуальной фиксации и метанаведения. Попытайтесь найти структуру, которая окажется логически правильной и при этом будет направлять взгляд так, чтобы необходимая информация не осталась незамеченной. Так вы со временем научитесь быстрее читать код, а в настоящий момент не выпадете из «потока».

Простые объекты значений

Стив Фримен



Классы, представляющие объекты значений, не нуждаются в геттерах или сеттерах. Разработчиков Java обычно учат использовать геттеры для доступа к полям, например:

```
public class Coordinate {
    private Latitude latitude;
    private Longitude longitude;

    public Coordinate(Latitude latitude, Longitude longitude) {
        this.latitude = latitude;
        this.longitude = longitude;
    }

    /**
     * @return широту Coordinate
     */
    public Latitude getLatitude() {
        return latitude;
    }

    /**
     * @return долготу Coordinate
     */
    public Longitude getLongitude() {
        return longitude;
    }
}

System.out.println(thing.getLatitude());
```

Идея заключается в том, что геттеры инкапсулируют способ представления значений в объекте, обеспечивая согласованный подход во всем коде. Это также гарантирует защиту от алиасинга — ситуаций, когда на один и тот же объект существует несколько ссылок — например путем клонирования коллекций перед их возвратом.

Этот стиль зародился в ранние годы существования JavaBeans, когда большой интерес вызывали графические инструменты, использующие рефлексию. Возможно, также имело место некоторое влияние Smalltalk (классический объектно-ориентированный язык), в котором все поля закрыты, если не доступны *через* акцессор. Поля, доступные только для чтения, имеют геттеры, но у них нет сеттеров.

На практике не все классы играют одинаковую роль, и, не имея альтернативной структуры в языке, многие программисты пишут классы Java, которые на самом деле являются *объектами значений*: простыми, никогда не меняющимися наборами полей, где равенство основано на значении, а не на идентичности. В нашем примере два объекта `Coordinate`, которые имеют одинаковую широту и долготу, фактически равнозначны. Я могу использовать экземпляры `Coordinate` в качестве констант во всем своем коде, потому что они неизменяемы.

Несколько лет назад я, как и многие мои коллеги, начал уставать от шаблонного дублирования, которого требуют геттеры, и упростил свой стиль для объектов значений. Я сделал все поля `public final`, по аналогии со `struct` из C:

```
public class Coordinate {
    public final Latitude latitude;
    public final Longitude longitude;

    public Coordinate(Latitude latitude, Longitude longitude) {
        this.latitude = latitude;
        this.longitude = longitude;
    }
}

System.out.println(coordinate.latitude);
```

Это допустимо, потому что объект неизменяем (опять же, нужно проявлять аккуратность с алиасингом, если какое-либо из значений структурировано), и я склонен избегать наследования или реализации большого поведенческого

функционала. Здесь мы видим изменение в подходе по сравнению с более ранними днями Java. Например, `java.awt.Point` является изменяемым, и методы `move` обновляют данные полей `x` и `y` на месте. В настоящее время, после двадцати лет улучшений в JVM и более широкого внедрения функционального программирования, такие переходные объекты настолько незатратны, что мы можем ожидать от `move` возвращения новой неизменяемой копии с новым местоположением. Вот пример для нашей `Coordinate`:

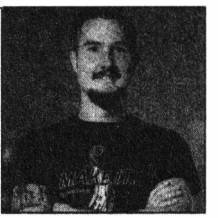
```
public class Coordinate {
    public Coordinate atLatitude(Latitude latitude) {
        return new Coordinate(latitude, this.longitude);
    }
}
```

Я обнаружил, что упрощенные объекты значений — ценное соглашение для уточнения роли типа, с меньшим объемом отвлекающего шума в коде. Они легко поддаются рефакторингу и часто выступают полезной целью для накопления методов, которые лучше выражают предметную область кода. Иногда поведенческие особенности объекта значений становятся более важными, и я понимаю, что могу выразить то, что мне нужно, с помощью методов и сделать поля закрытыми.

Кроме того, оказывается, что команда разработчиков языка Java тоже осознала это и ввела структуру record в Java 14. Пока эта структура не получит широкого распространения, нам придется полагаться на условности.

Позаботьтесь о своих объявлениях модулей

Николай Парлог



Если вы создаете модули Java, ваши объявления модулей (файлы module-info.java) легко могут стать самыми важными из ваших исходных файлов. Каждый из них представляет собой целый архив JAR и определяет, как он взаимодействует с другими JAR, так что внимательно следите за своими объявлениями! Вот несколько моментов, на которые необходимо обратить внимание.

Сохраняйте ясное объявление модулей

Объявления модулей являются кодом и должны рассматриваться как таковые, поэтому убедитесь, что применили свое форматирование кода. Кроме того, не размещайте директивы случайным образом, а структурируйте объявления своих модулей. Вот порядок, который использует JDK:

1. Requires, включая статические и переходные.
2. Exports.
3. Exports to.
4. Opens.
5. Opens to.
6. Uses.
7. Provides.

Что бы вы ни выбрали, если у вас имеется документ, определяющий стиль кода, запишите там решение. Если у вас есть IDE, инструмент сборки или анализатор кода, проверяющий такие вещи, это еще лучше. Попробуйте усовершенствовать его, чтобы он мог автоматически проверять — или даже применять — выбранный вами стиль.

Комментируйте объявление модулей

Мнения о документации по коду, такой как Javadoc или встроенные комментарии, сильно различаются, но, какую бы позицию ни занимала ваша команда в отношении комментариев, распространите ее на объявления модулей. Если вы хотите, чтобы абстракции содержали одно или два предложения, объясняющих их значение и важность, добавьте такой комментарий Javadoc к каждому модулю. Даже если это не в вашем стиле, большинство людей согласятся, что документировать то, *почему* вы приняли конкретное решение, — это хорошо. Возможно, при объявлении модуля вы теперь будете добавлять встроенный комментарий к:

- необязательной зависимости (объяснение, почему модуль может отсутствовать);
- квалифицированному экспорту (объяснение, почему модуль не является общедоступным API, но частично доступен);
- открытому пакету (объяснение, какие фреймворки должны получить к нему доступ).

Объявления модулей предоставляют новую возможность: теперь стало как никогда просто документировать взаимосвязи артефактов вашего проекта в коде.

Оцените объявление модулей

Объявления модулей — центральное представление вашей модульной структуры, их обязательно нужно изучать при проводимой вами проверке кода любого вида. Неважно, просматриваете ли вы свои изменения перед коммитом или перед открытием пул-реквеста, завершаете работу после сеанса парного программирования или ведете формальную проверку. Когда бы вы ни проверяли текст кода, уделяйте особое внимание *module-info.java*:

- Необходимы ли новые зависимости модулей (рассмотрите возможность замены сервисами) и соответствуют ли они архитектуре проекта?
- Подготовлен ли код для обработки отсутствия необязательных зависимостей?
- Необходим ли экспорт нового пакета? Все ли общедоступные классы там готовы к использованию? Можете ли вы уменьшить «площадь поверхности» API?

- Обоснована ли необходимость экспорта или это отговорка, чтобы получить доступ к API, который не готов стать общедоступным?
- Вносились ли изменения, которые не являются частью процесса сборки, но способны вызвать проблемы для потребителей?

Может показаться, что тщательный анализ дескрипторов модулей — пустая трата времени, но я рассматриваю это как возможность, ведь *никогда еще не было так легко анализировать и оценивать взаимосвязи артефактов вашего проекта и его структуры*. И речь не о фотографии схемы на белой доске, загруженной в вашу вики несколько лет назад, а о том, что надо, о фактических отношениях между вашими артефактами. Объявления модулей показывают неприкрытую реальность вместо устаревших благих намерений.

Заботьтесь о создаваемых зависимостях

Брайан Вермеер

В наше время разработка на Java сильно зависит от сторонних библиотек. Используя Maven или Gradle, мы получаем простые механизмы для импорта и применения опубликованных пакетов. Поскольку разработчики не хотят создавать и поддерживать шаблонную функциональность, а скорее сосредотачиваются на конкретной бизнес-логике, использование фреймворков и библиотек может стать разумным выбором.

Если взять типичный проект, то, возможно, объем написанного вами кода составляет всего 1%, а прочее приходится на импортированные библиотеки и фреймворки. Большая часть кода, который отправляется в продакшн, просто-напросто не принадлежит нам, но мы сильно зависим от него.

Когда мы оцениваем свой код и то, как относимся к вкладам членов команды, в качестве меры обеспечения качества первого запуска мы часто обращаемся к таким процессам, как проверка кода, прежде чем объединить новый код с основной ветвью. Есть другой вариант: процесс контроля качества также можно обеспечить посредством методики парного программирования. Однако то, как мы относимся к созданным зависимостям, сильно отличается от того, как мы относимся к своему собственному коду. Зависимости зачастую просто используются без какой-либо формы проверки. Важно отметить, что во многих случаях зависимости верхнего уровня, в свою очередь, приводят к вложенным зависимостям, которые могут находиться на много уровней глубже. Например, 200-строчное приложение Spring с пятью прямыми зависимостями может использовать в общей сложности 60 зависимостей, поэтому в продакшн отправляются почти полмиллиона строк кода.

Применяя только эти зависимости, мы слепо доверяем коду других людей, что странно на фоне того, как мы обращаемся с собственным кодом.

Уязвимые зависимости

С точки зрения безопасности вам следует проверить свои зависимости на наличие известных уязвимостей. Если обнаружена и раскрыта уязвимость

в какой-либо зависимости, вы должны узнать об этом и заменить или обновить ее. Взглянув на некоторые примеры из прошлого, вы поймете, что использование устаревших зависимостей с известными проблемами уязвимости может привести к катастрофическим последствиям.

Просматривая свои зависимости на каждом этапе процесса разработки, вы можете предотвратить неожиданное появление уязвимой зависимости до того, как отправите код в продакшн.

Также вы должны на постоянной основе сканировать свой рабочий снапшот, так как новые уязвимости могут проявить себя, когда вы уже используете его в вашей рабочей среде.

Обновление зависимостей

Вы должны разумно выбирать зависимости. Посмотрите, насколько хорошо поддерживается библиотека или фреймворк и сколько программистов работают над ними. Зависимость от устаревших или плохо обслуживаемых библиотек представляет собой большой риск. Если вы хотите оставаться в курсе последних событий, то полагайтесь на свой менеджер пакетов, помогающий определить, доступны ли более новые версии. Для подключаемого модуля версии Maven или Gradle вам пригодятся следующие команды, запускающие проверку наличия более новых версий:

- Maven: `mvn versions:display-dependency-updates`
- Gradle: `gradle dependencyUpdates -Drevision=release`

Стратегия для ваших зависимостей

Обрабатывая зависимости в системе, вы должны следовать определенной стратегии. Необходимо четко сформулировать причину, по которой используется та или иная зависимость, и критерии ее работоспособности. Далее, вы также должны тщательно продумать стратегию обновления. В целом считается, что частые обновления не столь болезненны. И последнее, но не менее важное: вы обязаны иметь инструмент, который сканирует ваши библиотеки на наличие известных уязвимостей, чтобы предотвратить их взлом.

В любом случае вы должны хорошо позаботиться о своих зависимостях и выбрать правильную библиотеку с правильной версией по правильной причине.

Принимайте разделение ответственности всерьез

Дэйв Фарли

Если вы изучали информатику, то, возможно, знаете о концепции, называемой *разделением ответственности**. Она лучше всего характеризуется эффектной строкой: «Один класс — одна вещь, один метод — одна вещь». Идея заключается в том, что ваши классы и методы (и функции) всегда должны быть сосредоточены только на своем результате.

Тщательно продумайте обязанности ваших классов и методов. Иногда я провожу занятия по разработке через тестирование (TDD), где использую сложение дробей в качестве простого ката программирования для изучения TDD. Я смотрю, какой первый тест пишут люди, и чаще всего он выглядит примерно так:

```
assertEquals("2/3", Fractions.addFraction("1/3", "1/3"));
```

Мне кажется, этот тест кричит: «Я плохо продуман!» Для начала, где находится дробь? Она существует только неявно, предположительно, внутри функции `addFraction`.

Хуже того, давайте подумаем о том, что здесь происходит. Как бы мы описали поведение функции `addFraction`? Возможно, что-то вроде: «Она берет две строки, анализирует их и вычисляет их сумму». Как только вы видите (или думаете про) «и» в описании функции, метода или класса, вы должны услышать тревожный звоночек в голове.

Здесь есть две проблемы: одна — в аспекте синтаксического анализа строк, а другая — со сложением дробей. Что, если вместо этого мы напишем тест следующим образом:

---

* «Разделение ответственности» впервые упомянул Эдсгер Вибе Дейкстра (*https://oreil.ly/Hyfse*) в своей статье 1974 года «О роли научной мысли», опубликованной в «Избранных работах по вычислительной технике: личная точка зрения» (New York: Springer-Verlag, 1982), 60–66.

```
Fraction fraction = new Fraction(1, 3);
assertEquals(new Fraction(2,3), fraction.add(new Fraction(1, 3)));
```

Как бы мы описали метод add в этом втором примере? Возможно, «он возвращает сумму двух дробей». Это второе решение проще в реализации, проще в тестировании, а код внутри будет проще для понимания. Оно также отличается значительной гибкостью, поскольку является более модульным и, следовательно, более составным. В частности, если бы нам захотелось сложить три дроби вместо двух, как бы мы добились этого? В первом примере нам пришлось бы добавить второй метод или реорганизовать первый, чтобы вызвать нечто вроде:

```
assertEquals("5/6", Fractions.addFractio("1/3", "1/3", "1/6"));
```

Во втором случае никаких изменений в коде не требуется:

```
Fraction fraction1 = new Fraction(1, 3);
Fraction fraction2 = new Fraction(1, 3);
Fraction fraction3 = new Fraction(1, 6);

assertEquals(new Fraction(5,6),
             fraction1.add(fraction2).add(fraction3));
```

Давайте предположим, что мы все-таки хотели начать со строкового представления. Мы могли бы добавить новый, второй класс, называемый как-то похоже на FractionParser или StringToFraction:

```
assertEquals(new Fraction(1, 3),
             StringFractionTranslator.createFraction("1/3"));
```

StringFractionTranslator.createFraction преобразует строковое представление дроби в Fraction. Допустимо применить в этом классе и другие методы, которые принимают Fraction и отображают String. Теперь можно протестировать этот код более тщательно, причем сделать это, не касаясь сложностей процесса сложения дробей, их умножения или чего-либо еще.

Разработка, основанная на тестировании, очень полезна в этом отношении, поскольку она четко выявляет проблемы плохого разделения ответственности (РО). Часто бывает так, что если вам трудно написать тест, то причиной тому либо плохая связь в вашем проекте, либо неудачное РО.

Разделение ответственности — это очень эффективная стратегия проектирования, которую можно применять в любом коде. Код с хорошим РО по определению более модульный и обычно обладает гораздо более высокими показателями составляемости, гибкости, тестируемости и удобочитаемости.

Всегда стремитесь к тому, чтобы каждый отдельный метод, класс и функция были сосредоточены только на своем результате. Как только вы заметите, что код пробует заниматься двумя вещами сразу, извлеките новый класс или метод, чтобы сделать его проще и понятнее.

Техническое интервьюирование — это навык, который стоит развивать

Триша Джи

Я открою вам секрет: наша индустрия *ужасна* в том, что касается собеседований с разработчиками. Самое нелепое здесь то, что мы почти никогда не поручаем кандидату писать реальный код в реальной среде, в которой он будет разрабатывать. Это все равно что проверять музыканта на теории, но никогда не слушать, как он играет.

Но есть и хорошая новость: прохождение собеседования — такой же навык, как и любой другой, а значит, ему можно научиться. Как и в тех случаях, когда вы приобретаете иные навыки, изучите, что от вас требуется, и практикуйтесь, практикуйтесь, практикуйтесь. Если потенциальные наниматели вам отказывают, это не значит, что вы плохой разработчик. Возможно, вы просто не очень хороши в собеседованиях. Вы можете улучшить этот навык, и каждая новая попытка — это еще одна возможность собрать больше данных и попрактиковаться.

На собеседованиях часто задают близкие по сути вопросы. Вот три из сравнительно типичных:

Подводные камни многопоточности

По-прежнему часто просят проверить код с помощью `synchronized`, разбросанных повсюду, и найти «состояние гонки потоков» или взаимоблокировку. У организаций с таким кодом есть проблемы посерьезнее, чем с наймом разработчиков (хотя, если они покажут такой код на собеседованиях, у них определенно возникнут проблемы с наймом разработчиков), так что, возможно, вы все равно не захотите там работать. Но если вы достаточно хорошо понимаете параллелизм в Java (*https://oreil.ly/n54xA*), то разберетесь и в большинстве таких задачек для собеседования. Если же вы не знакомы с Java-параллелизмом старой школы, расскажите, как

современная Java абстрагировалась от этих проблем, и объясните, как вы можете использовать в подобной ситуации Fork/Join (*https://oreil.ly/CEQjL*) или параллельные потоки (*https://oreil.ly/epUKa*).

Подводные камни компилятора

«Компилируется ли этот код?» Ну, не знаю, вообще-то у нас есть компьютеры и IDE — инструменты вам ответят, а я пока позабочусь о чем-нибудь другом. Но, если на собеседованиях вам задают подобные вопросы, воспользуйтесь некоторыми учебными материалами по сертификации Java (например, книгами), чтобы научиться отвечать на них.

Структуры данных

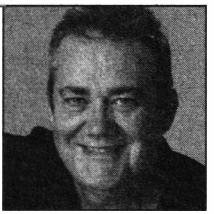
Структуры данных Java довольно просты: для начала будет здорово, если вы уясните разницу между `List` (*https://oreil.ly/tc6p4*), `Set` (*https://oreil.ly/KP1BA*) и `Map` (*https://oreil.ly/37mGa*). Не помешают и знания о хэш-коде (*https://oreil.ly/DvSYa*) и о том, как `equals` (*https://oreil.ly/QvlLo*) используется в контекстах коллекций.

Быстрый поиск в интернете по фразе «частые вопросы на собеседовании по java» также даст вам хороший набор тем для исследования.

А это не жульничество? Возможно, вы изучите достаточно материалов, чтобы пройти собеседование, но действительно ли вы будете знать достаточно, чтобы справляться с настоящими заданиями? Помните: наша индустрия ужасна в том, что касается собеседований с разработчиками. То, с чем вы столкнетесь там, зачастую очень далеко от того, что ждет вас на работе. Задавайте побольше вопросов, чтобы получить хоть какое-то представление о том, чем вам предстоит заниматься в действительности. Вы достаточно легко сумеете освоить новые технологии, ведь мы занимаемся этим постоянно, но то, добьетесь ли вы успеха, часто зависит от *межчеловеческих отношений*. Впрочем, это тема для другой статьи.

Разработка на основе тестирования

Дэйв Фарли



Как правило, разработку на основе тестирования (TDD) понимают неправильно. До TDD высокое качество ПО формировалось только знаниями, опытом и целеустремленностью программиста. После появления TDD добавилось кое-что еще.

Широко распространено мнение, что высокое качество программного обеспечения характеризуется следующими свойствами кода:

- модульность;
- слабая связанность;
- целостность;
- хорошее разделение ответственности;
- сокрытие информации.

Поддающийся тестированию код обладает этими свойствами. TDD — это разработка (проектирование), основанная на тестах. В TDD мы создаем тест до того, как написать код, обеспечивающий прохождение теста. TDD — нечто гораздо большее, чем «хорошее модульное тестирование».

Важно написать тест раньше прочего. Тогда у нас в любом случае будет «поддающийся тестированию» код. Это также означает, что покрытие никогда не станет проблемой. Если мы сначала напишем тест, у нас всегда будет отличное покрытие, и нам не нужно беспокоиться о нем как о метрике — а это плохая метрика.

TDD развивает талант разработчика ПО. Хотя данный метод не сделает плохих программистов великими, с его помощью любой программист станет лучше.

TDD очень прост — процесс делится на стадии, обозначаемые как «красный», «зеленый», «рефакторинг»:

- Пишем тест и видим, что он проваливается (красный).
- Пишем код, минимально необходимый для прохождения, и видим успешный результат (зеленый).
- Проводим рефакторинг кода и теста, чтобы сделать их максимально чистыми, выразительными, элегантными и простыми (рефакторинг).

Эти шаги представляют собой три отдельных этапа в разработке нашего кода. На каждом из них мы должны думать по-разному.

Красный

Сосредоточьтесь на том, чтобы выразить поведенческие намерения вашего кода. Сконцентрируйтесь только на его общедоступном интерфейсе. На данный момент мы разрабатываем только это и ничего более.

Думайте лишь о том, как написать хороший, понятный тест, который фиксирует именно то, чего вы ждете от вашего кода при выполнении.

Сосредоточьтесь на разработке общедоступного интерфейса, сделав тест простым в написании. Если ваши идеи легко выразить в тесте, их также будет легко выразить, когда кто-то станет использовать ваш код.

Зеленый

Найдите самое простое решение, которое позволит пройти тест, даже если оно покажется безыскусным. Пока тест проваливается, ваш код содержит ошибки и вы находитесь на нестабильной стадии разработки. Возвращайтесь к безопасному варианту (зеленому) так быстро и незатейливо, как только можете.

Ваши тесты должны развиваться, формируя «поведенческую спецификацию» для вашего кода. Если вы возьмете себе за правило писать код только тогда, когда у вас есть неудачный тест, это поможет вам лучше разработать и развить данную спецификацию.

Рефакторинг

Вернувшись к «зеленому» этапу, вы можете безопасно провести рефакторинг. Так вы не дадите себе расслабиться, заблудиться в сорняках и потеряться!

Сделайте небольшие простые шаги, а затем повторно запустите тесты, чтобы убедиться, что все по-прежнему работает.

Рефакторинг — это не запоздалые правки, а возможность более стратегически продумать свой проект. Если ваши тесты слишком сложно настраивать, ваш код, вероятно, имеет плохое разделение ответственности и, возможно, чрезмерно тесно связан с другими фрагментами. Если для тестирования вашего кода вам нужно включить чересчур много дополнительных классов, вероятно, он недостаточно изолирован.

Практикуйте перерыв для рефакторинга всякий раз после того, как добьетесь прохождения теста. Смотрите на код и размышляйте: «А нельзя ли это улучшить?» Три фазы TDD различны, и вам в каждой из них также следует сосредотачивать умственные усилия на разных моментах, чтобы добиться максимальной пользы.

В вашем каталоге bin/ отличные инструменты

Род Хилтон

Каждый разработчик Java знаком с инструментами `javac` для компиляции, `java` для запуска и, возможно, `jar` для упаковки приложений Java. Однако вместе с JDK устанавливается множество других полезных инструментов. Они уже находятся на вашем компьютере в каталоге *bin/* вашего JDK, и их можно вызвать через PATH. Вам полезно будет ознакомиться с некоторыми из этих инструментов, чтобы понять, чем вы располагаете.

`jps`

Если вы когда-либо замечали, что запускаете `ps aux | grep java`, чтобы найти запущенные JVM, вам, вероятно, достаточно будет просто запустить `jps`. В этом специальном инструменте перечислены все запущенные JVM, но вместо того, чтобы показывать вам длинную команду с `CLASSPATH` и аргументами, `jps` просто перечисляет идентификатор процесса и имя основного класса приложения, что значительно упрощает определение процессов — какой из них какой. `jps -l` показывает полное имя главного класса, `jps -m` показывает список аргументов, передаваемых в метод main, и `jps -v` показывает все аргументы, переданные в саму JVM.

`javap`

JDK поставляется с дизассемблером файлов классов Java. Запустив `javap <class-файл>`, вы увидите поля и методы этого файла класса, что часто бывает очень полезно для понимания того, во что «под капотом» превращается код, написанный на языках, доступных в JVM (в частности, Scala, Clojure или Groovy). Запустите `javap -c <class-файл>`, чтобы просмотреть полный байт-код этих классов.

`jmap`

Запуск `jmap -heap <id процесса>` выведет сводную информацию о пространстве памяти процесса JVM, например, о том, сколько памяти используется в каждом поколении памяти JVM, а также о конфигурации кучи и типе применяемого GC. `jmap -histo <id процесса>` выведет гистограмму

каждого класса в куче, количество экземпляров этого класса и количество потребляемых байтов памяти. Что наиболее важно, запуск `jmap -dump:format=b,file=<имя файла> <id процесса>` сбросит снапшот всей кучи в файл.

`jhat`

Запуск `jhat <файл heap dump>` примет файл, сгенерированный `jmap`, и запустит локальный веб-сервер. Вы можете подключиться к этому серверу в браузере, чтобы интерактивно исследовать пространство кучи, сгруппированное по имени пакетов. Есть ссылка «Показать количество экземпляров для всех классов (исключая платформу)», считающая только экземпляры классов за пределами самой Java. Также существует возможность запускать запросы OQL, которые позволяют опрашивать пространство кучи с помощью синтаксиса в стиле SQL.

`jinfo`

Запустите `jinfo <id процесса>`, чтобы просмотреть все системные свойства, загруженные JVM, и флаги командной строки JVM.

`jstack`

Запуск `jstack <id процесса>` выведет трассировки стека для всех текущих потоков Java, запущенных в JVM.

`jconsole` *и* `jvisualvm`

Эти графические инструменты позволяют подключаться к нескольким JVM и интерактивно отслеживать запущенные JVM. Они предоставляют вам визуальные графики и гистограммы различных аспектов запущенного процесса и служат альтернативой для многих из перечисленных выше инструментов в том плане, что в них удобно пользоваться мышью.

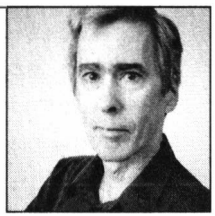
`jshell`

Начиная с Java 9 в Java есть полноценный REPL — отличный инструмент для проверки синтаксиса, быстрого запуска команд на основе Java или тестирования кода и экспериментов, не требующих создания целых программ.

Многие из этих инструментов могут выполняться не только локально, но и применимо к процессам JVM, запущенным на удаленных машинах. Я перечислил лишь часть полезных программ, которые вы уже установили. Уделите какое-то время тому, чтобы разобраться, какие еще штуки есть в каталоге исполняемых файлов вашего JDK, и прочитать их справочные странички. Всегда ведь полезно знать, что за инструменты висят у вас на поясе.

Вылезайте из песочницы Java

Иэн Ф. Дарвин



«Java — лучший язык на свете, какую цель ни возьми». Если вы так считаете, вам нужно почаще выходить из дома. Конечно, Java — отличный ЯП, но не единственный хороший язык и не лучший для любой возможной цели. На самом деле вам, как профессиональному разработчику, следует регулярно находить время для того, чтобы изучить и применить какой-нибудь новый язык, будь то по работе или для самостоятельных начинаний. Заройтесь достаточно глубоко, чтобы понять, чем альтернатива фундаментально отличается от того, к чему вы привыкли, и может ли она принести пользу в ваших проектах. Другими словами: попробуйте, вам должно понравиться. Вот несколько ЯП, которые вы, возможно, захотите выучить:

- JavaScript — это язык браузера. Хотя они похожи названиями и дюжиной или около того ключевых слов, JavaScript и Java очень разные. JavaScript поставляется с сотнями различных веб-фреймворков, некоторые из них выходят за рамки интерфейса. Например, Node.js (*https://nodejs.org*) позволяет запускать JavaScript на стороне сервера, что открывает множество новых возможностей.

- Kotlin (*https://kotlinlang.org*) — язык JVM, который, как и большинство аналогичных ЯП, имеет менее многословный синтаксис, чем Java, наряду с другими функциями, также способными дать ему преимущество над Java. Google применяет Kotlin для большей части того, что сам делает с Android, и поощряет его использование в приложениях для Android. Этим все сказано!

- Dart (*https://dartlang.org*) и Flutter (*https://flutter.dev*): Dart — это компилируемый скриптовый язык от Google. Первоначально предназначенный для веб-программирования, он не процветал до тех пор, пока фреймворк Flutter не начал использовать Dart для компиляции приложений под платформы Android и iOS (а когда-нибудь дойдет и до стороны браузера) из одной кодовой базы.

- Python (*https://www.python.org*), Ruby (*https://oreil.ly/jtdUQ*) и Perl (*https://www.perl.org*) существуют уже несколько десятилетий и остаются одними

из самых популярных языков. Первые два имеют реализации JVM, JRuby и Jython (хотя у него нет активной поддержки).

- Scala (*https://oreil.ly/iJX8Q*), Clojure (*http://clojure.org*) и Frege (*https://oreil.ly/vXlmZ*) (реализация Haskell (*https://www.haskell.org*)) — языки функционального программирования (FP) (*https://oreil.ly/ u0BQX*) в JVM. FP (functional programming) имеет долгую историю ограниченной распространенности, но в последние годы приобретает массовую популярность. На момент написания этой статьи многие языки FP, например Idris (*https://oreil.ly/ YS0vJ*) и Agda (*https://oreil.ly/X8wti*), не работали в JVM. Если вам не очень комфортно в среде Java 8+, вероятно, изучение FP поможет вам использовать ее функциональные возможности.

- R (*https://oreil.ly/eh0Tw*) — это интерпретируемый язык для управления данными. R, клонированный (*https://oreil.ly/PbWQW*) из S (*https://oreil.ly/ yDxJZ*), который Лаборатории Белла разработали для статистиков, сейчас популярен среди специалистов по обработке данных или тех, кому нужны более мощные средства, чем электронные таблицы. Содержит множество встроенных и дополнительных функций статистики, математики и графики.

- Rust (*https://oreil.ly/Shxzu*) — компилируемый язык, предназначенный для разработки систем с функциями параллелизма и строгой типизации.

- Go (*https://golang.org*) («Golang») — компилируемый язык, изобретенный в Google Робертом Гриземером, Робом Пайком и Кеном Томпсоном (соавтором Unix). Существует несколько компиляторов, изначально ориентированных на разные операционные системы и веб-разработку путем компиляции до JavaScript и WebAssembly.

- C — предок C++, Objective-C и, в некоторой степени, Java, C# и Rust. (C дал этим языкам базовый синтаксис встроенных типов, синтаксис методов и фигурные скобки для блоков кода, и именно этот язык виноват в том, что `int i = 077;` имеет значение 63 в Java.) Если вы не изучали язык ассемблера, то, побывав на «уровне C», начнете понимать модели памяти и испытаете благодарность к Java за то, как она работает.

- JShell (*https://oreil.ly/vkgl3*) — это не язык как таковой, а другой способ работы с Java. Вместо того чтобы писать `public class Mumble { public static void main(String[] args) {` лишь для того, чтобы опробовать выражение или какой-то новый API, просто забудьте все церемонии и шаблоны и используйте JShell.

Так что вперед, выйдите за пределы Java!

Мысли о сопрограммах

Доун и Дэвид Гриффитс

Сопрограммы — это функции или методы, которые можно приостанавливать и возобновлять. В Kotlin их допустимо использовать вместо потоков для асинхронной работы, поскольку многие сопрограммы могут эффективно выполняться в одном потоке.

Чтобы показать, как они функционируют, мы сейчас напишем пример программы, которая параллельно воспроизводит следующие последовательности барабанного боя:

Инструмент	Последовательность
Барабаны-альты	`x-x-x-x-x-x-x-x-`
Педальная тарелка	`x-x-x---x-x-x---`
Подвесная тарелка	`----------------x----`

Мы *могли бы* использовать здесь потоки, но в большинстве систем звук воспроизводится звуковой подсистемой, в то время как выполнение кода приостанавливается до тех пор, пока она не сможет воспроизвести следующий звук. Расточительно блокировать подобным образом такой ценный ресурс, как поток.

Вместо этого мы сейчас создадим набор сопрограмм, по одной для каждого из инструментов. У нас будет метод под названием `playBeats`, который принимает последовательность барабанов и имя звукового файла. Полный текст кода находится здесь: *https://oreil.ly/6x0GK*, а упрощенная версия выглядит так:

```
suspend fun playBeats(beats: String, file: String) {
  for (...) { // для каждого такта
    ...
    playSound(file)
```

```
    ...
    delay(<time in milliseconds>)
    ...
  }
}
```

Вызовем это с помощью `playBeats("x-x-x---x-x-x---", "high_hat.aiff")`, и код воспроизведет последовательность, используя звуковой файл *high_hat.aiff*. В этом коде есть две особенности, которые встретятся вам в любой сопрограмме Kotlin:

- Он начинается с ключевого слова `suspend`, а значит, в какой-то момент функция может приостановить свою работу до тех пор, пока какой-либо внешний код не перезапустит ее.
- Он включает в себя неблокирующий вызов функции `delay`.

Функция `delay` аналогична чему-то вроде `Thread.sleep`, за исключением того, что она работает путем передачи управления обратно во внешний мир с запросом на повторное возобновление после указанной паузы.

Если сопрограмма выглядит так, как же ее вызывать? Что вызывает сопрограмму, обеспечивает приостановку, а затем обращается к ней снова, когда ее нужно перезапустить? Функция `launch` позаботится обо всем за нас. Основной метод запуска сопрограмм выглядит следующим образом:

```
fun main() {
  runBlocking {
    launch { playBeats("x-x-x-x-x-x-x-x-", "toms.aiff") }
    launch { playBeats("x-x-x---x-x-x---", "high_hat.aiff") }
    launch { playBeats("---------------x----", "crash_cymbal.aiff") }
  }
}
```

Каждый вызов `launch` принимает блок кода, который вызывает сопрограмму. Блок кода в Kotlin подобен лямбде в Java. Функция `launch` регистрирует вызов сопрограммы областью действия, предоставляемой функцией `runBlocking`.

`runBlocking` запускает цикл планирования в главном потоке, который координирует вызовы каждой из сопрограмм. Она по очереди вызывает каждую из сопрограмм `playBeats` и ожидает ее приостановки, вызывая `delay`. Затем `runBlocking` ожидает, пока не потребуется возобновить какую-либо другую

сопрограмму `playBeats`. Функция `runBlocking` выполняет это до тех пор, пока не завершатся все сопрограммы.

Вам стоит рассматривать сопрограммы как легкие потоки: они позволяют вам мысленно разбивать работу на отдельные простые задачи, которые, по-видимому, выполняются одновременно, причем в одном и том же потоке.

Сопрограммы неоценимы при написании кода для Android, который применяет строгую потоковую модель, где отдельные операции должны выполняться в основном потоке пользовательского интерфейса. Но они также полезны для создания масштабируемых серверных приложений, обязанных эффективно использовать существующие потоки.

Потоки — это инфраструктура, относитесь к ним соответственно

Рассел Уиндер

Сколько Java-разработчиков управляют использованием стека во время процесса программирования или хотя бы думают об этом? Примерно нисколько. Подавляющее большинство Java-программистов предоставляют управление стеком компилятору и системе выполнения.

Сколько Java-разработчиков управляют использованием кучи во время программирования или хотя бы думают об этом? Очень немного. Большинство Java-программистов предполагают, что система сбора мусора целиком возьмет управление кучей на себя.

Так почему же множество Java-программистов управляют всеми своими потоками вручную? Потому что их так учили. С самого начала Java поддерживала многопоточность с общей памятью, что можно уверенно считать большой ошибкой.

Почти все сведения об управлении потоками и параллелизме, имеющиеся у большинства Java-программистов, основаны на теории построения операционных систем 1960-х годов. Если вы пишете ОС, то все замечательно, но действительно ли большинство программ на Java представляют собой ОС? Нет. Так что переосмысление необходимо.

Если в вашем коде есть какие-либо синхронизированные инструкции, блокировки, мьютексы — всё атрибуты операционных систем, — то вы почти наверняка делаете все неправильно. Здесь большинство Java-программистов выбирает неверный уровень абстракции. Точно так же, как пространство стека и пространство кучи являются управляемыми ресурсами, потоки следует рассматривать как управляемые ресурсы. Вместо того чтобы явно создавать потоки и управлять ими, создавайте задачи и отправляйте их в пул

потоков. Задачи должны быть однопоточными — это очевидно! Если у вас много задач, которым необходимо взаимодействовать друг с другом, то вместо использования общей памяти применяйте потокобезопасную очередь.

Все это стало ясно еще в 1970-х годах, в результате чего сэр Чарльз Энтони (Тони) Ричард Хоар создал теорию коммуникационных последовательных процессов (CSP) в качестве алгебры для описания одновременных и параллельных вычислений.

К сожалению, большинство программистов проигнорировали это, стремясь использовать многопоточность с общей памятью. И, по сути, каждая их программа представляла собой новую операционную систему. Однако в 2000-е годы многие стремились вернуться к последовательным процессам взаимодействия. Возможно, самым статусным сторонником этого в недавние годы стал язык программирования Go. Все дело во взаимодействии последовательных процессов, выполняемых через базовый пул потоков.

Используемые многими термины «акторы», «поток данных», «CSP» или «активные объекты», представляют собой вариации на тему последовательного процесса и коммуникации. Akka, Quasar и GPars — все это фреймворки, предоставляющие различные формы задач через пул потоков. Платформа Java поставляется с фреймворком Fork/Join, который можно использовать явно, а также поддерживает библиотеку Streams — революционное обновление Java, представленное в Java 8.

Потоки как управляемый ресурс — это правильный уровень абстракции для подавляющего большинства Java-разработчиков. Такие абстракции, как акторы, поток данных, CSP и активные объекты, используются подавляющим большинством программистов. Отказ от ручного управления потоками снижает нагрузку на Java-разработчиков, что помогает им писать более простые, понятные и удобные в обслуживании системы.

Три черты по-настоящему отличных разработчиков

Джанна Патчей

Получив степень бакалавра в области компьютерных наук и математики, первые несколько лет карьеры я провела в качестве разработчика на Java. Такая работа приносила мне искреннее удовольствие. Как и многие другие математики, я как одержимая старалась писать чистый и элегантный код и целую вечность рефакторила строки, пока результат не приближался к совершенству настолько, насколько это вообще возможно. Я знала, что где-то есть конечные пользователи, но только в том смысле, что они предоставляли требования, создающие трудности, с которыми мне затем приходилось справляться.

Перенесемся на 20 лет вперед с момента окончания университета. Сейчас я нахожусь на совершенно ином поприще — даю консультации по вопросам регулирования финансовых рынков и структуры рынка, проявляя особый интерес к финансовым инновациям, что помогает мне не отрываться от моих корней технаря. Долгие годы я сотрудничала со многими разработчиками уже «с другой стороны забора» — как человек, который предоставляет и разъясняет требования. И со временем я стала лучше ценить определенные черты, которыми обладают по-настоящему отличные разработчики. Такие особенности выходят за рамки их технических возможностей.

Первая и самая важная черта — любознательность, то самое стремление, которое побуждает вас искать решение проблемы, разбираться, как все работает, и создавать нечто новое. На нее можно и нужно опираться при взаимодействии с клиентами и заинтересованными сторонами. Здорово, когда разработчики задают много вопросов о бизнес-сфере, поскольку это показывает, что они действительно хотят понимать и учиться. Это также помогает им лучше разобраться в бизнес-сфере и более эффективно решать проблемы конечных пользователей. Я сталкивалась со множеством менеджеров по развитию, которые активно советовали своим командам «не беспокоить» бизнес лишними вопросами. Они ошибались.

Вторая и третья черты — эмпатия и воображение. Речь о том, способны ли вы поставить себя на место конечного пользователя, чтобы попытаться понять его приоритеты и впечатления от работы с вашим ПО. И о том, сумеете ли вы затем, основываясь на своем техническом опыте, предложить творческие решения проблем, с которыми они сталкиваются. Разработчики часто склонны считать несущественным многое из того, о чем здесь говорится, или предполагать, что этим должен заниматься кто-то другой. Но, если вы станете напрямую общаться с бизнесом, так будет гораздо эффективнее, а вы повысите свою квалификацию.

Все это может показаться очевидным, но такие черты весьма важны. Недавно я посетила конференцию по технологиям и инновациям, на которой основное внимание уделялось тому, как важно наладить сотрудничество между технологической сферой и бизнесом, чтобы наилучшим образом использовать новые процессы и средства — например, облачную среду, технологию распределенных реестров и искусственный интеллект / машинное обучение. Многие выступавшие подчеркивали, насколько важно разрушать барьеры между разработчиками и конечными пользователями. Теперь случается, что программистов вводят в бизнес-команды и ожидают, что они будут знать о бизнес-сфере на том же уровне. Так что особенности, о которых речь, еще касаются будущего и того, как работать умнее. Если вы сможете развить их у себя, перед вами откроются новые двери.

Компромиссы в архитектуре микросервисов

Кенни Бустани

Существует ли оптимальная архитектура программного обеспечения? На что она похожа? Как мы измеряем «оптимальность», когда дело касается создания и эксплуатации ПО? Итак, оптимальная архитектура программного обеспечения обладает максимальной гибкостью для внесения изменений при минимально возможных затратах. Здесь стоимость измеряется с точки зрения определенных качеств, относящихся к проектированию и реализации архитектуры ПО, и сюда добавляются расходы на инфраструктуру для ее эксплуатации. Определяющая черта любого «качества ПО» заключается в том, что его можно объективно измерить и оно влияет на другие качества.

Например, если архитектура ПО требует строгих гарантий согласованности, это воздействует на такие качества, как производительность и доступность. Эрик Брюер создал теорему CAP для описания набора измеримых компромиссов, согласно которой вы можете выбрать только два из трех видов гарантий для работы базы данных (*согласованности*, *доступности* и *терпимости к разделению*). Согласно теореме, когда приложения совместно используют состояние через границы сети, вы должны выбирать между согласованностью или доступностью, поскольку не можете иметь и то и другое.

Одна из главных проблем, связанных с микросервисами, заключается в том, что для них не существует единого всеобъемлющего определения. Более того, микросервисы представляют собой набор концепций и идей, основанных на комплексе ограничений для предоставления архитектуры сервисов. Всякий микросервис или любое ПО, создаваемое вами, пополняют собой историю принятых решений, влияющую на выбор, который вам нужно сделать сегодня.

Хотя для микросервисов, пожалуй, нет единого определения, чаще всего они обладают следующими характеристиками:

- Возможность независимого развертывания
- Организованы вокруг бизнес-возможностей

- Отдельная база данных для каждого сервиса
- Одно приложение — одна команда
- API-first архитектура
- Непрерывная поставка

Погружаясь в мир разработки ПО, вы рано или поздно обнаружите, что там не существует такого понятия, как верный выбор. Вообще говоря, большинство разработчиков или операторов полагают, что существует «лучшее решение из возможных», и вы, наверное, заметите, что они твердо выступают в его поддержку. Сталкиваясь со все новыми ситуациями выбора из нескольких вариантов (например, какую базу данных использовать), вы в конечном счете обнаружите, что все доступные решения предполагают определенные компромиссы. То есть вам, как правило, придется что-то потерять ради того, чтобы получить нечто иное.

Вот краткий список взаимных увязок, с которыми вы можете столкнуться при принятии решения о включении зависимости в ваш микросервис:

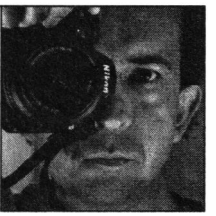
Доступность	Как часто моя система доступна для пользователей?
Производительность	Какова общая производительность моей системы?
Согласованность	Какие гарантии обеспечивает моя система в аспекте согласованности?
Скорость	Как быстро я могу развернуть изменение одной строки кода в рабочей среде?
Композиционность	Какой процент архитектуры и кодовой базы можно использовать повторно, а не дублировать?
Вычисления	Какова стоимость вычислений моей системы при пиковой нагрузке?
Масштабируемость	Какова стоимость увеличения мощности, если пиковая нагрузка продолжает увеличиваться?
Маржинальность	Каково среднее уменьшение маржинальности от добавления разработчиков в мою команду?
Терпимость к разделению	Если раздел в сети вызовет сбой или задержку, будет ли мое приложение работать или вызовет каскадный сбой?

Как ответ на один из этих вопросов повлияет на ответы на прочие?

Вы заметите, что каждый из них обычно имеет какое-то отношение к другим. Если вам когда-нибудь потребуется принять трудное решение по части программной архитектуры, использующей микросервисы, вернитесь к этому списку вопросов.

Не проверяйте свои исключения

Кевлин Хенни



Если вы когда-нибудь захотите отправиться в ад, то уж точно не собьете ноги. Дорога ведь очень хорошо вымощена благими намерениями, куда ни кинь взгляд. И по крайней мере один из этих брусчатых камней — это модель проверяемых исключений (ПИ) Java.

Проверяемым называют исключение, которое, если оно не обрабатывается внутри метода, должно появиться в сигнатуре метода в виде ключевого слова `throws`.

Любые классы, произошедшие от `Throwable`, *могут* перечисляться после `throws`, но необработанные ПИ (не произошедшие от `RuntimeException` или `Error`) *необходимо* там перечислять. Это особенность языка Java, но она не имеет никакого значения для JVM и не является обязательным требованием для языков JVM.

Благое намерение здесь заключается в том, что неудачные сценарии метода наделяются той же значимостью, что и его входные и выходные данные для успешного сценария. На первый взгляд, это кажется разумным. Действительно, в небольшой и закрытой кодовой базе легко достичь на уровне типа уверенности в том, что некоторые исключения не будут пропущены. И, как только данная цель достигнута, появляется некая (очень) хрупкая уверенность в полноте кода.

Однако методы, которые могут работать в небольших форматах, не обязаны масштабироваться. Проверяемые исключения Java представляли собой эксперимент по объединению потока управления с типами, а эксперименты всегда дают результаты. Вот что узнали на своем опыте создатели C# *https://oreil.ly/rCT18*):

> C# не требует и не допускает таких спецификаций исключений. Изучение небольших программ показывает, что требование спецификаций исключений может как повысить производительность разработчиков,

так и улучшить качество кода, но опыт работы с крупными программными проектами предполагает другой результат — снижение производительности и незначительное или нулевое повышение качества кода.

Так говорят разработчики C#, других языков JVM, других не-JVM языков… В чем бы ни состояло первоначальное намерение, на практике ПИ воспринимаются как препятствия. А если уж программисты в чем-то искусны, так это в умении обходить барьеры.

Компилятор жалуется на необработанное проверяемое исключение? Один раз нажали на подсказку в IDE, и препятствие исчезло! Вместо него вы получили постоянно удлиняющееся ключевое слово `throws`, которое помещает случайную информацию в сигнатуры методов, часто обнажая детали, которые следовало бы инкапсулировать.

Или, возможно, вы добавите `throws Exception` или `throws OurCompanyException` к каждому методу, безжалостно разделавшись со стремлением к определенности каждого сбоя?

А как насчет «перехватить и убить»? Если вы спешите закоммитить свой код, пустой блок `catch` исправит все что нужно! Проверяемое исключение теперь как Балрог, а вы в роли Гэндальфа кричите: «Ты не пройдешь!»

ПИ привносят и провоцируют синтаксический багаж. Но проблемы лежат глубже. Вопрос не только в самодисциплине программиста или терпимости к многословию: для фреймворков и расширяемого кода применение проверяемых исключений изначально ошибочно.

При публикации интерфейса вы доверяетесь контракту, подписанному с помощью сигнатур методов. Как признавал Толстой в «Анне Карениной», сценарии черного дня не так просты, не так определенны и не так предсказуемы, как сценарии счастливого дня:

> Все счастливые семьи похожи друг на друга, каждая несчастливая семья несчастлива по-своему.

Стабильность интерфейса — это сложно. Эволюция интерфейса — это сложно. Добавление `throws` все усложняет.

Если другие люди вставляют свой код в ваш и используют ваш код в своем приложении, они знают, какие исключения им можно выбрасывать, но вы не знаете, и вас это не волнует. Ваш код должен позволять исключениям передаваться из их присоединенного кода в обработчики в основном коде

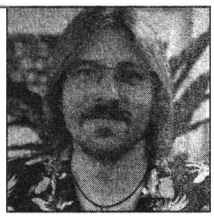
приложения. Открытая инверсия контроля требует прозрачности исключений.

Однако, если другие люди используют проверяемые исключения, они не смогут применять ваши интерфейсы, если только вы не добавите к каждому методу `throws Exception` (шум, который создает нагрузку на весь зависимый код) — или если они не станут пробрасывать свои исключения, обернутые в `RuntimeException`... Или если они вместо всего этого не изменят свой подход и не начнут стандартизировать непроверяемые исключения.

Последний вариант — самый легкий, наиболее стабильный и открытый подход из всех возможных.

Как высвободить потенциал интеграционного тестирования с использованием контейнеров

Кевин Виттек



Вероятно, большинство разработчиков Java сталкивались с пирамидой тестирования на каком-либо этапе своей карьеры — например, в рамках учебной программы. Или, возможно, упоминания о ней встречались им в выступлениях на конференциях, в статьях или сообщениях в блогах. Можно отыскать множество историй происхождения и вариаций этой метафоры (и углубиться в те, что достойны отдельной статьи), но в целом идея сводится вот к чему. У вас есть значительная база модульных тестов, над ней — не такая крупная часть, интеграционные тесты, и еще выше — слой сквозных тестов пользовательского интерфейса, как бы наконечник.

Данную форму продвигают как оптимальное соотношение различных классов тестов. Однако эти рекомендации, как и любые концепции, касающиеся программного обеспечения и компьютеров, следует оценивать в контексте. То есть исходите из того, что интеграционные тесты окажутся медленными и хрупкими. И это предположение, вероятно, верно, если ожидается, что интеграционные тесты будут выполняться в общей среде тестирования или потребуется обширная настройка локальных зависимостей. Но останется ли идеальная форма пирамидальной, если мы поставим такие предположения под сомнение?

С появлением все более мощных компьютеров мы можем либо использовать виртуальные машины (ВМ) для полноценного размещения всей среды разработки, либо применять их для управления и запуска внешних зависимостей, необходимых для интеграционного тестирования (например, баз данных или брокеров сообщений). Но, поскольку большинство реализаций ВМ не освобождены от накладных расходов, это значительно увеличит

нагрузку и потребление ресурсов рабочей станцией разработчика. Кроме того, временны́е затраты на запуск и создание виртуальных машин так велики, что нет смысла специально настраивать требуемую среду в рамках выполнения теста.

С другой стороны, появление удобной для пользователя контейнерной технологии позволяет перейти к новым парадигмам тестирования. Эти реализации контейнеров с низкими накладными расходами (по сути, изолированные процессы с собственной автономной файловой системой) дают возможность создавать и настраивать необходимые службы по требованию, а также использовать единый инструментарий. Впрочем, до сих пор настройка в основном трудоемко выполнялась вручную и возникала нужда в дополнительных сотрудниках, что замедляло внедрение новых разработчиков и порождало вероятность канцелярских ошибок.

На мой взгляд, мы как сообщество должны стремиться к тому, чтобы сделать подготовку и настройку тестовой среды неотъемлемой частью выполнения теста и даже самого тестового кода. В случае Java это означает, что выполнение набора тестов JUnit, независимо от того, выполняется ли он IDE или инструментом сборки, неявно приведет к созданию и настройке набора контейнеров, необходимых для тестов. И эта цель достижима с помощью современных технологий!

Мы можем напрямую взаимодействовать с контейнерным движком, используя существующие API или инструменты командной строки, — по сути, писать наш собственный «драйвер контейнера». Но обратите внимание на то, что есть разница между запуском контейнера и готовностью сервиса внутри контейнера к тестированию. В качестве альтернативы также можно изучить экосистему Java для существующих проектов, которые предоставляют эти функциональные возможности на более высоком уровне абстракции. В любом случае пришло время высвободить мощь хороших интеграционных тестов и сбить с них кандалы прошлого!

Необъяснимая эффективность фаззинга

Нэт Прайс

Неважно, ведется разработка на основе тестирования или нет, программисты, пишущие автоматизированные тесты, страдают от положительной предвзятости тестирования*,**: более вероятно, что они решат проверить, правильно ли ведет себя ПО при вводе действительных данных, чем надежность ПО при неверном вводе. Так и получается, что наши наборы тестов не могут обнаружить целые классы дефектов. *Fuzz testing**** (фаззинг) — это необъяснимо эффективный метод отрицательного тестирования, который легко включить в существующие автоматизированные наборы тестов. Если вы добавите фаззинг в процесс разработки на основе тестирования, это поможет вам создать более устойчивую систему.

Например, мы расширяли ПО повсеместно используемого потребительского продукта для извлечения данных из веб-сервисов. Хотя мы очень старались писать надежный сетевой код и тестировали как отрицательные, так и положительные случаи, фаззинг сразу же выявил удивительное количество входных данных, которые заставили бы ПО выдавать неожиданные исключения. Многие стандартные Java API, которые анализируют данные, выдают непроверяемые исключения, поэтому ранее средство проверки типов не сумело удостовериться, что приложение обработало все возможные ошибки

* Аднан Каушевич, Ракеш Шукла, Сасикумар Пуннеккат и Даниэль Сундмарк. Влияние отрицательного тестирования на TDD: Промышленный эксперимент / Хьюберт Баумайстер, Барбара Вебер. Гибкие процессы в разработке программного обеспечения и экстремальном программировании: 14-я Международная конференция, XP 2013. Вена, Австрия, 3–7 июня 2013 г. — Berlin: Springer, 2013. — С. 91–105, *https://oreil.ly/qX_4n*.

** Лаура Мари Левенталь, Барби М. Тизли, Дайан С. Ролман, Кит Инстон. Положительная предвзятость при тестировании программного обеспечения среди профессионалов. Обзор / Леонард Дж. Басс, Юрий Горностаев, Клаус Унгер. Взаимодействие человека и компьютера EWHCI 1993 Конспекты лекций по информатике, том 753. — Berlin: Springer, 1993. — С. 210–218, *https://oreil.ly/FTecF*.

*** Майкл Саттон, Адам Грин, Педрам Амини. Фаззинг: обнаружение уязвимостей методом перебора. — Upper Saddle River, NJ: Addison-Wesley Professional, 2007.

парсинга. Такие неожиданные исключения могут привести к тому, что устройство окажется в неизвестном состоянии. Если речь о потребительском устройстве, даже таком, которое можно обновлять удаленно, это способно вылиться в значительный рост расходов из-за звонков в службу поддержки клиентов или вызовов инженеров.

Нечеткий тест генерирует множество случайных входных данных, вводит их в тестируемое программное обеспечение и проверяет, продолжает ли ПО демонстрировать приемлемое поведение. Чтобы обеспечить полезное покрытие, фаззер должен генерировать входные данные, которые достаточно действительны, чтобы ПО не отклонило их сразу же, но достаточно недействительны, чтобы выявить скрытые случаи, не охваченные ранее, или указать дефекты в логике обработки ошибок.

Добиться этого можно двумя способами:

- *Фаззеры, основанные на изменениях*, преобразуют примеры хороших входных данных, чтобы создать, возможно, недопустимые тестовые входные данные.
- *Фаззеры на основе генерации* вырабатывают входные данные из формальной модели (например, грамматики), которая определяет структуру допустимых входных данных.

Фаззеры, основанные на изменениях, считаются непрактичными для тестирования «черного ящика», поскольку в таких условиях трудно получить достаточное количество образцов достоверных входных данных*. Однако, когда мы проводим тест-драйв нашего кода, положительные тестовые примеры предоставляют готовый набор допустимых входных данных, которые используют многие пути проверки в программном обеспечении. Фаззинг на основе изменений становится не просто практичным, но и легким в применении.

Иногда запуск тысяч случайных входных данных через всю систему занимает много времени. Опять же, если мы используем фаззинг во время разработки, то можем применить данный метод к определенным функциям нашей системы и спроектировать их таким образом, чтобы их удавалось тестировать изолированно. Затем используем фаззинг для проверки правильности поведения этих блоков и проверки типов, чтобы убедиться, что они верно сочетаются с остальной частью системы.

---

* Чарли Миллер, Закари Н. Дж. Питерсон. Анализ мутаций и фаззинга на основе генерации. — DefCon 15, 2007. — С. 1–7.

Вот пример фаззинга, который вместе с проверкой типов гарантирует, что анализатор сообщений JSON будет выдавать только проверенные исключения, объявленные в его сигнатуре:

```
@Test public void
only_throws_declared_exceptions_on_unexpected_json() {
  JsonMutator mutator = new JsonMutator();
  mutator.mutate(validJsonMessages(), 1000)
    .forEach(possiblyInvalidJsonMessage -> {
      try {
        // нас не волнует результат парсинга в этом тесте
        parseJsonMessage(possiblyInvalidJsonMessage);
      }
      catch (FormatException e) {
        // допустимо
      }
      catch (RuntimeException t) {
        fail("unexpected exception: " + t +
          " for input: " + possiblyInvalidJsonMessage);
      }
    });
}
```

Фаззинг стал неотъемлемой частью моего инструментария для разработки на основе тестирования. Это помогает устранить дефекты и делает конструкцию системы более композиционной.

Простая библиотека для проведения фаззинга на основе изменений в проектах Java и Kotlin доступна на GitHub (*https://oreil.ly/nxVuC*).

Используйте покрытие, чтобы улучшить ваши модульные тесты

Эмили Бач

Измерить охват ваших тестов стало проще, чем когда-либо. В современной среде IDE кнопка для запуска тестов с покрытием находится совсем рядом с кнопками для их запуска или отладки.

Результаты покрытия представляются для каждого класса отдельно, с небольшими графическими диаграммами, а соответствующие строчки в исходном коде выделяются цветом.

Данные о покрытии легко получить. Но как их лучше всего использовать?

Когда вы пишете новый код

Большинство людей согласны, что нужно предоставлять модульные тесты вместе со всем кодом, который вы пишете. Можно обсуждать то, в каком порядке что-то делать, но, судя по моему опыту, лучше всего работают короткие циклы обратной связи.

Напишите немного тестового кода, напишите немного кода в продакшн и создайте функциональность вместе с тестами. Когда я работаю таким образом, то время от времени запускаю тесты с покрытием в качестве дополнительной страховки, чтобы не забыть протестировать какой-нибудь новый код, который я только что написала.

Главная опасность здесь заключается в том, что вы, будучи очень довольны высоким охватом, не заметите, что вам не хватает как кода, так и тестов для важной части функциональности. Возможно, вы забыли добавить обработку ошибок. Или же упустили какое-то бизнес-правило. Если вы изначально *не* написали продакшн-код, то измерения покрытия не обнаружат это вместо вас.

Когда вам необходимо изменить код, который написан не вами

Порой бывает сложно изменить код с плохими или отсутствующими тестами, который вы писали не сами, — особенно если вы не вполне понимаете, что он делает, но вам все равно нужно его изменить. Когда я сталкиваюсь с такой ситуацией, помогает покрытие тестированием — один из способов узнать, насколько хороши тесты и какие части кода получится рефакторить более уверенно.

Я также могу полагаться на данные о покрытии, чтобы находить новые тестовые примеры и увеличивать покрываемые области. Однако тут нужна осторожность. Если я пишу тесты исключительно для увеличения покрытия, то есть риск довольно тесно связать тесты с реализацией.

Когда вы работаете в команде

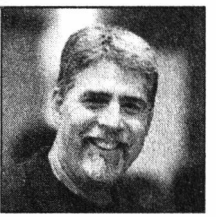
Команда характеризуется тем, что у нее, помимо прочего, есть «нормы» или принятые модели поведения, с которыми все согласны, и неважно, оговорены они явно или нет. Вероятно, вам следует принять в качестве одной из командных норм то, что в процесс проверки кода и тестирования необходимо включать измерения покрытия. Возможно, так вы заметите, где отсутствуют тесты, и поймете, что некоторым коллегам нужно получить дополнительную поддержку и пройти обучение, чтобы писать более качественные тесты. Кроме того, вы наверняка здорово воодушевитесь, если увидите, что новая сложная функциональность хорошо покрыта тестами.

Если вы регулярно измеряете покрытие тестированием всей вашей кодовой базы, я бы посоветовала вам больше обращать внимание на тенденции, чем на цифры как таковые. Я видела, как из-за произвольно выбранных целевых показателей покрытия работники предпочитали тестировать только то, что легко поддавалось тестированию. Порой люди избегают рефакторинга, потому что тогда у них появятся новые строки кода и снизится доля покрытия в целом. Я видела тесты с отсутствующими или очень слабыми утверждениями, написанные лишь ради того, чтобы улучшить показатели покрытия.

Предполагается, что покрытие поможет вам улучшить ваши модульные тесты, а модульные тесты должны облегчить рефакторинг. Измерения покрытия — это инструмент, с помощью которого вы улучшите модульные тесты и облегчите себе жизнь.

Свободно применяйте нестандартные идентификационные аннотации

Марк Ричардс



Аннотации в Java легки в написании, просты в использовании и очень эффективны — по крайней мере, некоторые из них.

Традиционно аннотации в Java предоставляли удобный способ реализации аспектно-ориентированного программирования (АОП), метода, предназначенного для выделения общих поведенческих задач путем внедрения поведения в определенные участки кода. Однако большинство разработчиков решительно отказались от АОП из-за нежелательных побочных эффектов, а также потому, что желали иметь весь код в одном месте — файле класса.

Идентификационные аннотации совершенно особенны в том смысле, что не содержат никакой функциональности. Вместо этого они предоставляют только программную информацию, которая применяется для управления, анализа или документирования отдельных аспектов кода или архитектуры.

Идентификационные аннотации пригодны для определения границ транзакций, домена или поддомена, описания классификации и систематизации служб, обозначения кода фреймворка и еще десятков различных вариантов использования.

Например, нередко бывает важно определять классы, которые представляют собой часть базовой структуры (или шаблонного кода в микросервисах), чтобы в дальнейшем тщательно отслеживать или защищать изменения. Именно это и делает следующая аннотация:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
```

```
public @interface Framework {}

@Framework
public class Logger {...}
```

Подождите... Аннотация же ничего не делает! Или делает? Она обозначает этот класс как класс, связанный с фреймворком. Следовательно, изменения в данном классе могут повлиять почти на все другие классы. Вы можете написать автоматические тесты, чтобы в том случае, если какой-либо код фреймворка изменит эту версию, отправлялось уведомление. Кроме того, разработчики будут знать, что они изменяют класс, который относится к базовому коду фреймворка.

Ниже приведен список других стандартных идентификационных аннотаций, которые я использую на регулярной основе (все они указаны на уровне класса):

`public @interface ServiceEntrypoint {}`

Определяет точку входа микросервиса. Она также используется в качестве заполнителя для других аннотаций описания сервиса, перечисленных ниже.

Применение: `@ServiceEntrypoint`

`public @interface Saga {public Transaction[] value()...}`

Определяет службы, участвующие в распределенной транзакции. В значении `Transaction` перечислены транзакции, охватывающие несколько служб. Добавляется к классам, которые содержат аннотацию `@ServiceEntrypoint`.

Применение: `@Saga({Transaction.CANCEL_ORDER})`

`public @interface ServiceDomain {public Domain value()...}`

Определяет логический домен (например, Платеж, Доставка, Эмитент и т. д.), к которому принадлежит сервис (определяется значением `Domain`). Добавляется к классам, содержащим аннотацию `@ServiceEntrypoint`.

Применение: `@ServiceDomain(Domain.PAYMENT)`

`public @interface ServiceType {public Type value()...}`

Определяет классификацию сервиса. Значение `Type` перечисляет определенные типы сервисов (классификация). Добавляется к классам, которые содержат аннотацию `@ServiceEntrypoint`.

Применение: `@ServiceType(Type.ORCHESTRATION)`

```
public @interface SharedService {}
```

Определяет класс как класс, содержащий общий (совместный) код для всего приложения (например, форматеры, калькуляторы, логгирование, безопасность и т. д.).

Применение: `@SharedService`

Идентификационные аннотации — это форма *программной документации*. В отличие от неструктурированных комментариев к классам, идентификационные аннотации предоставляют согласованные средства для обеспечения соответствия или выполнения аналитики. Также их применяют для информирования разработчика о контексте класса или службы. Например, когда вы пишете приспособленческие функции с помощью ArchUnit (*https://* www.archunit.org), то можете использовать аннотации, чтобы убедиться, что все общие классы находятся на уровне сервисов приложения:

```
@Test
public void shared_services_should_reside_in_services_layer() {
  classes().that().areAnnotatedWith(SharedService.class)
  .should().resideInAPackage("..services..").check(myClasses);
}
```

Подумайте над тем, чтобы применять идентификационные аннотации вместо комментариев. Используйте их не скупясь, чтобы получать информацию, аналитику и программный контроль над вашими сервисами или приложениями.

Тестируйте, чтобы разрабатывать более качественное ПО быстрее

Марит ван Дейк

Тестируя код, вы проверяете, выполняет ли ваш код то, чего вы от него ждете. Тесты также помогают добавлять, изменять или удалять функциональность, ничего не нарушая. Но у тестирования бывают и другие преимущества.

Просто поразмыслив о том, *что именно* тестировать, вы можете определить различные способы будущего использования ПО, обнаружить аспекты, которые еще не ясны, и лучше понять, что должен (и не должен) делать код. Рассчитав, *как именно* протестировать все это еще до начала реализации, вы также сумеете улучшить тестируемость и архитектуру вашего приложения. Так вы придете к наилучшему решению еще до того, как будут написаны тесты и код.

В рамках архитектуры вашей системы подумайте не только о том, что тестировать, но и о том, *где именно*. Бизнес-логику следует тестировать как можно ближе к ее реализации в коде: модульные тесты подойдут для небольших модулей (методов и классов), интеграционные тесты — для проверки интеграции между различными компонентами, контрактные тесты — для предотвращения взлома вашего API и т. д.

Подумайте, как взаимодействовать с вашим приложением в контексте теста, и используйте инструменты, разработанные для этого конкретного уровня: от модульного тестирования (например, JUnit, TestNG) до API (например, Postman, REST-assured, RestTemplate) и пользовательского интерфейса (например, Selenium, Cypress).

Помните о цели конкретного типа тестирования и применяйте соответствующие инструменты: в частности, Gatling или JMeter для тестов производительности, Spring Cloud Contract testing или Pact для контрактного тестирования и PITest для тестирования изменений.

Но эти инструменты следует использовать не просто так, а по назначению. Конечно, шуруп можно забить молотком, но тогда и дерево, и шуруп окажутся в проигрыше.

Автоматизация тестирования является частью вашей системы, и ее необходимо поддерживать вместе с продакшн-кодом. Убедитесь, что эти тесты приносят пользу, и рассмотрите затратность их запуска и обслуживания.

Тесты должны быть надежными и повышать доверие. Если тест не работает, либо исправьте его, либо удалите. Не игнорируйте его — позже вы потратите время впустую, задаваясь вопросом, почему этот тест игнорируется. Удалите тесты (и код), которые больше не представляют ценности. Неудачный тест обязан *быстро* сообщить вам, *что именно* не так, не тратя много времени на анализ сбоя. Следовательно:

- Каждый тест должен проверять что-то одно.
- Используйте осмысленные описательные имена. Также не просто описывайте, что делает тест (мы и сами можем прочитать код), а расскажите нам, *зачем* он это делает. Так мы сможем решить, следует ли обновить тест в соответствии с измененной функциональностью или обнаружен фактический сбой, который необходимо исправить.
- Библиотеки сопоставления, такие как Hamcrest, помогут предоставить подробную информацию о разнице между ожидаемыми и фактическими результатами.
- Никогда не доверяйте тесту, который ни разу не выявил ошибку.

Но невозможно (и не нужно) автоматизировать всё. Ни один инструмент не скажет вам, как ощущается ваше приложение в использовании. Не бойтесь запускать свое приложение и изучать его — люди гораздо лучше машин замечают то, что немного «не в порядке». И, кроме того, усилия по автоматизации не всегда окупаются.

Тестирование должно давать вам правильную обратную связь в нужный момент, чтобы вы с достаточной уверенностью переходили на следующий этап жизненного цикла разработки ПО — от коммита и объединения кода до развертывания и разблокировки функционала приложения.

Если вы сделаете всё правильно, то сможете быстрее создавать более качественное ПО.

Применение объектно-ориентированных принципов в тестовом коде

Энджи Джонс

При написании тестового кода важно проявлять ту же осторожность, что и при разработке продакшн-кода. Ниже рассмотрены общие способы использования объектно-ориентированных (ОО) принципов при реализации тестового кода.

Инкапсуляция

При автоматизации тестирования обширно применяется шаблон проектирования объектной модели страницы (*https://oreil.ly/guEVi*). Он предписывает создание класса для взаимодействия со страницей тестируемого приложения. Внутри этого класса находятся объекты локатора для элементов веб-страницы и методы для взаимодействия с этими элементами. Лучше всего как следует инкапсулировать — ограничить доступ к самим локаторам и предоставить доступ только к их соответствующим методам:

```
public class SearchPage {
    private WebDriver driver;
    private By searchButton = By.id("searchButton");
    private By queryField = By.id("query");

    public SearchPage(WebDriver driver){
        this.driver = driver;
    }

    public void search(String query) {
        driver.findElement(queryField).sendKeys(query);
        driver.findElement(searchButton).click();
    }
}
```

Наследование

Хотя наследованием не следует злоупотреблять, оно, безусловно, бывает полезно в тестовом коде. Например, поскольку на каждой странице существуют компоненты верхнего и нижнего колонтитулов, излишне создавать поля и методы для взаимодействия с этими компонентами в каждом классе объектов страницы. Вместо этого создайте базовый класс `Page`, содержащий общие элементы, которые существуют на каждой странице, и пусть ваши классы объектов страницы наследуются от этого класса. Теперь ваш тестовый код будет иметь доступ ко всему, что находится в верхнем и нижнем колонтитулах, независимо от того, с каким объектом страницы они в данный момент взаимодействуют.

Другой хороший способ использовать наследование в тестовом коде — различные реализации конкретной страницы. Допустим, ваше приложение содержит страницу профиля пользователя, которая имеет различные функциональные возможности в зависимости от ролей (например, Администратор, Участник). Они в чем-то различны, но также могут пересекаться. Дублирование кода между двумя классами — не идеальное решение. Вместо этого создайте класс `ProfilePage`, содержащий общие элементы/взаимодействия, а также подклассы (например, `AdminProfilePage`, `MemberProfilePage`), которые реализуют уникальные взаимодействия и наследуют общие.

Полиморфизм

Предположим, у нас есть удобный метод, ведущий на страницу профиля пользователя. Этот метод не знает, к какому типу относится страница профиля — Администратора или Участника.

Здесь от вас потребуется принять архитектурное решение. Вы создадите два метода — по одному для каждого из типов профилей? По-моему, избыточно, поскольку они оба будут делать одно и то же, просто имея разный тип возвращаемого значения.

Вместо этого возвращайте родительский класс (`ProfilePage`), пока `AdminProfilePage` и `MemberProfilePage` являются классами-наследниками `ProfilePage`. Тестовый метод, вызывающий этот удобный метод, имеет больше контекста и может соответственно выдавать:

```
@Test
public void badge_exists_on_admin_profile() {
```

```
    var adminProfile = (AdminProfilePage)page.goToProfile("@admin");
    ...
}
```

Абстракция

Абстракция редко применяется в тестовом коде, но существуют допустимые варианты использования. Рассмотрим тип виджета, настроенный для различных применений в приложении. При разработке классов, которые взаимодействуют с конкретными реализациями этого виджета, полезно создать абстрактный класс, определяющий ожидаемое поведение:

```
public abstract class ListWidget {
    protected abstract List<WebElement> getItems();
    int getNumberOfItems() {
        return getItems().size();
    }
}

public class ProductList extends ListWidget {
    private By productLocator = By.cssSelector(".product-item");
    @Override
    protected List<WebElement> getItems() {
        return driver.findElements(productLocator);
    }
}
```

Тестовый код — самый настоящий код, а значит, его необходимо поддерживать, улучшать и масштабировать. Поэтому, когда вы разрабатываете его, в ваших интересах следовать передовым методикам программирования, в том числе основополагающим принципам ООП.

Как развивать карьеру, опираясь на силы сообщества

Сэм Хепберн

Сейчас уже недостаточно просто быть отличным Java-разработчиком. Если вы хотите продвинуться по карьерной лестнице, вам нужно вести блог, выступать на конференциях, активничать в социальных сетях, использовать открытый исходный код… Список можно продолжать. Порой такая задача кажется сложной и вы, вероятно, спрашиваете себя: «Почему? Ну почему моих технических способностей недостаточно?» Что ж, краткий ответ заключается в том, что в большинстве случаев люди, от чьих решений зависит ваша карьера, никогда не увидят ваш код. Вам нужно добиться того, чтобы они услышали и заметили ваше имя.

Луч надежды

Вам не нужно делать *все* вышеописанное, и есть сообщества, которые подставят вам плечо. Если сама мысль о том, чтобы толкать речь перед 10, 50, 100 (или более) людьми буквально вызывает у вас приступ паники, не выходите на сцену.

С другой стороны, если вы просто нервничаете и чувствуете, что вам нечего сказать, тут сообщество способно помочь. Вы когда-нибудь исправляли проблему, с которой боролись, и думали про себя: «Если бы только я мог поучиться у кого-то, кто уже это делал»? У каждого возникали такие мысли, и они превращаются в отличные темы для обсуждения в беседе или сообщении в блоге.

Если вы боитесь говорить со сцены, то начните с малого: перед тем как подавать заявку на участие в локальной группе пользователей Java (JUG) или конференции, расскажите о чем-нибудь своей команде.

Как может помочь сообщество?

Помимо того, что вы станете более известным, участие в сообществе так ценно еще и потому, что там люди беседуют и делятся сведениями. Технологии развиваются так быстро, что если вы принадлежите к сообществу, то вам

не надо ждать публикации какой-либо книги, чтобы получить доступ к отличным материалам. Их авторы, исследующие новейшие методики и средства, обсуждают свои идеи на форумах, делятся ими на общественных мероприятиях и в блогах.

Люди в сообществах, с которыми вы, вероятно, уже знакомы, могут помочь вам в саморазвитии. Докладчики вы или участники, то, чему вы учитесь друг у друга, иногда более ценно, чем основная тема мероприятия. Не бойтесь задавать вопросы всем присутствующим в помещении. Лидерство можно разделить многими способами, и у тех, кто сидит рядом с вами, найдутся нужные вам ответы.

Если в ваших краях нет процветающего сообщества Java, не паникуйте — ознакомьтесь с Virtual JUG (*https://virtualjug.com*).

Ищете себе следующую задачу?

Если вас интересуют новые головоломки, то сообщество способно серьезно помочь вам в поиске работы. Если менеджер по найму знает, что ему не обязательно просматривать сотни заявлений на своем столе, потому что можно сразу нанять кандидата с нужными навыками, который впишется в команду, он так и поступит.

Каков наилучший способ попасть на вершину кучи? Найдите варианты взаимодействия вне процесса подачи заявки. В ходе личных встреч в локальных группах пользователей вы также поймете, каково на самом деле трудиться именно в этой команде. Никаких собеседований, где все сладенько улыбаются, но после которых вы в первый же рабочий день обнаруживаете, что попали в неподходящую для вас среду.

Вот мы и вернулись к тому, с чего начали: люди, от чьих решений зависит ваша карьера, не всегда видят ваш код!

Что такое программа JCP и как в ней участвовать

Хизер Ванчура

Программа Java Community Process (JCP) (*https://oreil.ly/t6agC*) — это процесс, посредством которого международное сообщество Java стандартизирует и ратифицирует спецификации (*https://oreil.ly/vzEzX*) для Java-технологий. Программа JCP гарантирует, что высококачественные спецификации разрабатываются с использованием всеобъемлющего, основанного на консенсусе подхода. Спецификации, утвержденные программой JCP, должны сопровождаться эталонной реализацией (доказывает реализуемость спецификации) и комплектом совместимости технологий (набор тестов, инструментов и документации, используемых для тестирования реализаций на соответствие спецификации).

Опыт показал, что лучший способ подготовить технологическую спецификацию — использовать открытый и инклюзивный процесс разработки спецификации и внедрения, сопровождаемый группой отраслевых экспертов с различными точками зрения. Процесс включает в себя предоставление сообществу возможностей для рассмотрения и комментариев, а также мощное техническое руководство, призванное обеспечить достижение технических целей и интеграцию данной спецификации с другими соответствующими спецификациями.

За утверждение прохождения спецификаций через различные этапы программы JCP и согласование расхождений между спецификациями и связанными с ними наборами тестов отвечает Исполнительный комитет (ИК) (*https://oreil.ly/J7Sng*), состоящий из ключевых представителей заинтересованных сторон (поставщиков Java, крупных финансовых учреждений, использующих Java для ведения бизнеса, групп с открытым исходным кодом и других членов сообщества Java, включая отдельных лиц и группы пользователей).

Программа JCP, внедренная в 1999 году, развивалась с течением времени, сама используя данный процесс. Проект ее развития, условно обозначенный

JCP.next, открыто реализуется Исполнительным комитетом JCP (*https://oreil.ly/8Xg8c*). JCP.next — серия запросов от спецификации Java (JSR), призванных сосредоточить внимание на прозрачности и оптимизации программы JCP, а также расширении ее членского состава. Эти JSR трансформируют процессы JCP, изменяя документ процесса JCP. Как только изменения завершатся, их будут применять ко всем новым JSR и к будущим сервисным выпускам для существующих JSR на платформе Java.

Например, JSR 364 (*https://oreil.ly/q3X1U*) «Расширение членства в JCP» был введен в действие как версия JCP 2.10. Этот JSR расширил возможности участия в JCP: определил новые классы членства, гарантировал более широкую вовлеченность сообщества и помог обеспечить соответствующие обязательства членов JCP в области интеллектуальной собственности. Любой разработчик Java вправе присоединиться к программе JCP в качестве либо ведущего специалиста по спецификации JSR, либо члена экспертной группы, либо рядового участника.

JSR 387 (*https://oreil.ly/ce2ag*) «Оптимизация программы JCP» был введен в действие как версия 2.11. Данный JSR упрощает процесс жизненного цикла JSR, приводя его в соответствие с современными методами разработки технологии Java, что, в частности, позволяет финализировать любые JSR в рамках шестимесячного цикла выпуска платформы Java. С помощью этого JSR мы также изменили размер ИК JCP.

Несмотря на многочисленные преобразования в сообществе Java, программа JCP неизменно продолжает существовать. Любой желающий может подать заявку на вступление (*https://oreil.ly/eSzdV*) и участвовать в программе JCP в качестве либо действительного члена (корпорация или НКО), члена-партнера (группа пользователей Java) или ассоциированного члена (частное лицо). Стабильность программы JCP и участие членов сообщества (*https://oreil.ly/z8rot*) обеспечивают дальнейший успех платформы Java и ее будущее. Стандарты гарантируют реализацию технических стратегий, а JCP — сотрудничество индустрии и участие сообщества разработчиков.

Совместимость важна. Программа JCP требует использовать спецификации, RI и TCK, что позволяет создать экосистему вокруг технологий Java. Программа JCP обеспечивает для этого основу и структуру: права и обязанности интеллектуальной собственности защищены, а возможность выбора реализаций, которые проходят TCK, приносит пользу экосистеме. Вот ключ к успеху и дальнейшей популярности технологии Java.

Почему я не придаю никакого значения сертификации

Колин Випурс

Некоторое время назад — пожалуй, где-то в середине нулевых, — один мой друг добился на экзамене на Java Certified Programmer впечатляющего результата: 98%. Стремясь не отставать, я сдал один из тренировочных тестов во время обеденного перерыва и, хотя набрал не так много баллов, получил проходной результат. В голове у меня навсегда засел один вопрос из билета, что-то связанное с иерархией наследования в приложениях Swing. Ответил я без проблем, поскольку тогда работал со Swing на постоянной основе, но мне показалось странным спрашивать о чем-то, что можно легко найти в своей IDE. Сдать экзамен я так и не удосужился, в основном из-за того, что в то время был на полпути к получению степени магистра.

Перенесемся на несколько лет вперед — я только что приступил к участию в новом проекте. В течение первой недели один из моих новых коллег спросил, есть ли у меня сертификат Java 5. «Нет, — ответил я, — но я пользовался этой версией в течение последнего года». Оказывается, он-то сертифицированный специалист... Ну, прекрасно, что кто-то в моей команде обладает *базовым* уровнем знаний и навыков. Менее чем через две недели он спросил, почему мы должны утруждать себя переопределением `hashCode`, когда переопределяем `equals`. Он искренне не понимал взаимосвязи между этими двумя методами. Это лишь малая толика того, чего он *не* знал, и все же его сертифицировали!

Перенесемся еще на несколько лет вперед. Я подписываю контракт с компанией, политика которой заключается в том, что каждый постоянный сотрудник должен иметь сертификат хотя бы на тогдашнем уровне Java Certified Programmer. Я все-таки встретил там нескольких хороших разработчиков, и такие специалисты удостаивались повышений, но попадались и по-настоящему ужасные программисты — все с сертификатами.

Бегло взглянув на сайт Oracle для сертификации Java, вы узнаете, что получение сертификата «...поможет вам обоснованно позиционировать себя как

обладателя полного набора навыков и знаний, необходимых для того, чтобы стать профессиональным разработчиком Java» и «...к вам будут относиться с бóльшим доверием, сертификация поможет вам лучше выполнять вашу повседневную работу, руководить и продвигать свою команду и компанию». Ерунда. Навыки «профессионального разработчика» и способность «лучше выполнять вашу повседневную работу» имеют мало общего с тем, что вам нужно для получения сертификата. Вы можете так изучить теорию, что сдадите экзамены, не написав ни строчки кода. Мы (отрасль в целом) неспособны даже определенно сказать вам, что такое «хорошо» и «плохо», поэтому клочок бумаги, утверждающий, что ему-то можно доверять, ничего не стоит.

Конечно, из каждого правила есть исключения. Я встречал нескольких людей — ну, по крайней мере, одного, — которые воспользовались сертификацией Java как способом укрепить собственные знания. Они решили сдать экзамен, чтобы выучить то, с чем им иначе не довелось бы встретиться в рамках своей повседневной работы, и перед этими людьми я снимаю шляпу.

Я уже более двадцати лет профессионально пишу ПО, и за это время неизменным оставалось одно: хорошим разработчикам сертификация не нужна, но плохие могут легко ее получить.

Пишите к документации комментарии в одно предложение

Питер Хилтон

Согласно одному распространенному заблуждению, авторы непонятного кода якобы способны каким-то образом ясно и понятно выражать свои мысли в комментариях.

Кевлин Хенни

Вероятно, вы либо пишете *слишком много* комментариев в своем коде, либо *вообще ничего* не пишете. «Слишком много» обычно означает «слишком много для поддержания», что может привести к появлению опасно неточных комментариев, которые стоило бы удалить. «Слишком много» также, вероятно, указывает, что они плохо составлены и не улучшены, потому что трудно писать «доходчиво и ясно».

В случае «вообще ничего» вы полагаетесь на то, что у вас идеальное именование, структура кода и тесты, а достичь этого еще сложнее, чем кажется.

Мы все много раз видели код, авторы которого вообще не писали никаких комментариев — то ли для экономии времени, то ли потому, что не хотели, то ли потому, что считали свой код самодокументируемым.

Иногда код действительно так хорош: например, первая тысяча строк нового проекта, хобби-проект, искусно написанный вручную, и, возможно, зрелый, достойно поддерживаемый библиотечный проект узкой направленности с небольшой кодовой базой.

С крупными приложениями дела обстоят иначе, особенно с корпоративными бизнес-приложениями. Комментарии становятся проблемой, когда вам нужно поддерживать 100 000 строк кода, причем его написали (и продолжают добавлять) другие люди. Этот код не всегда идеален, а потому нуждается в разъяснениях. Встает неудобный вопрос: а *сколько* там объяснений, сколько комментариев?

Решение проблемы комментирования больших кодовых баз приложений состоит в том, что к документации нужно писать комментарии «в одно предложение» следующим образом:

1. Напишите самый лучший код, на какой вы только способны.
2. Напишите комментарий в одно предложение к документации для каждого общедоступного класса и метода/функции.
3. Проведите рефакторинг кода.
4. Удалите ненужные комментарии.
5. Перепишите плохие комментарии (потому что для хорошего результата всегда требуется поработать).
6. Добавьте подробности — только там, где это совершенно необходимо.

Такой подход поможет определить, какие комментарии точно нужны. Они требуются либо там, где фрагмент кода сам по себе не может объяснить, зачем он здесь вообще, либо там, где у вас еще не дошли руки до рефакторинга. Создавая комментарии в одно предложение, вы поймете: если на написание хорошего комментария тратится несколько минут, значит, он необходим и в будущем сэкономит время вам и другим читателям.

Если же вы написали хороший комментарий так быстро, как только можете набирать текст, значит, вы определили «очевидный» код, который не нуждается в комментарии. Такой комментарий стоит удалить.

Хитрость в том, что для такого открытия все равно нужно *напечатать* комментарий, каким бы очевидным вы ни считали код (и особенно если вы написали его сами). Не пропускайте этот шаг!

Вам всегда нужно минимальное количество строчек, которые «комментируют только то, что не может сказать код»*, отвечая на вопросы «почему?», на которые вы не сумели ответить в коде. Ограничив их длину одним предложением на каждый общедоступный интерфейс, вы получите реалистичный объем работ по написанию, проверке кода и техническому обслуживанию и сможете сосредоточиться на качестве и краткости.

Не пишите больше одного предложения, если только иначе никак. Возможно, вам нужно ответить на больше вопросов «зачем?», встретился необычно сложный фрагмент или попался непонятный жаргон предметной области,

* Кевлин Хенни. 97 вещей, которые должен знать каждый Java-программист (O'Reilly).

требующий объяснения (прежде всего аббревиатуры). Делегируйте, где получится: в бизнес-доменах часто есть страницы «Википедии», на которые вы можете ссылаться.

Комментарии удивительно полезны, если они хороши. Главным образом потому, что мы тратим больше времени на чтение кода, чем на его написание. Кроме того, комментарии — единственная функция, общая для всех основных ЯП. При программировании используйте язык, лучше всего подходящий для конкретной задачи. Иногда это ваш родной язык.

Пишите «читаемый код»

Дэйв Фарли

Мы все слышали, что хороший код «читаем», но что же это означает?

Первый принцип удобочитаемости: пусть код будет простым. Избегайте длинных методов и функций. Вместо этого разбейте их на более мелкие части и дайте им названия, основанные на том, что они делают.

Автоматизируйте ваши стандарты программирования, чтобы вам удавалось тестировать их в конвейере непрерывной поставки. Например, вы можете завершить сборку с ошибкой, если отыщется метод, содержащий более 20–30 строк кода, или списки параметров, где больше 5 или 6 параметров.

Еще один способ улучшить читаемость — понимать слово «читаемый» буквально. Не интерпретируйте это как: «Сумею ли я понять свой код через пять минут после того, как напишу его?» Лучше попробуйте создать код, в котором разобрался бы непрограммист. Вот простая функция:

```
void function(X x, String a, double b, double c) {
    double r = method1(a, b);
    x.function1(a, r);
}
```

Что она делает? Независимо от того, программист вы или нет, вы не сможете ответить, если не заглянете внутрь реализации `X` и `method1`.

Но если бы я написал так...

```
void displayPercentage(Display display, String message,
                       double value, double percentage) {
  double result = calculatePercentage(value, percentage);
  display.show(message, result);
}
```

...стало бы понятно, что к чему. Даже непрограммист, вероятно, догадался бы по названиям, что здесь происходит. Да, кое-что все еще скрыто — мы не знаем, как работает `display` или как рассчитывается процент, — но это хорошо. Мы можем понять, что пытается сделать этот код.

Конечно, в подобных легких примерах внесенные изменения выглядят так тривиально, что их и обсуждать не стоит, но как часто настолько же элементарный код попадается вам на работе?

Если серьезнее относиться к именованию и добавить к этому простые методы рефакторинга, вы быстро получите более глубокое представление о том, что происходит в вашем коде.

Вот еще один пример кода, в данном случае из какого-то реального проекта:

```
if (unlikely(!ci)) {
    // 361 строка кода
} else {
    // 45 строк 'else'
}
```

Выделите `unlikely(!ci)` и создайте новый метод с названием `noConnection`.

Выделите 361 строку в инструкции `if` и назовите это `createConnection`, и в итоге вы получите:

```
if (noConnection(ci)) {
    ci = createConnection();
} else {
    // 45 строк 'else'
}
```

Разумное присвоение названий (даже если это речь об извлечении функции, которая используется только один раз, и то чтобы дать ей название) придает коду ясность, отсутствующую в противном случае. Также оно часто подчеркивает тот факт, что существуют значительные возможности для упрощения кода.

В приведенном примере в том же файле нашлось еще пять мест, где получилось бы повторно использовать новый метод `createConnection`. Я бы пошел дальше и переименовал `ci` в `connection` или что-то более подходящее.

Поскольку мы улучшили модульность кода, этот подход также дает больше возможностей для дальнейших изменений. Например, теперь мы можем решить, что нужно скрыть некоторые дополнительные сложности в этом методе и просто использовать соединение, независимо от того, создано ли оно здесь в первый раз или нет:

```
ci = createConnection(ci);
// 45 строк кода
```

Упростите функции и методы. В контексте проблемы, которую вы решаете, сделайте значимыми все имена — функций, методов, переменных, параметров, констант, полей, чего угодно!

Представьте, что код читают ваши дедушка с бабушкой (не технари). Догадаются ли они, что происходит? Если нет, сделайте код проще посредством рефакторинга и более выразительным с помощью выбора подходящих имен.

Молодые, старые и мусор

Мария Ариас де Рейна

Одним из главных преимуществ Java является то, что разработчикам не приходится (сильно) задумываться о памяти. В отличие от многих других языков, существовавших на момент ее запуска, Java с самого начала автоматически освобождала неиспользуемую память. Но это не значит, что Java-программистам не нужно знать основы того, как она обрабатывает память. Вы все равно можете столкнуться с утечками памяти и узкими местами.

Java делит свою память на два сегмента:

Heap	Экземпляры, переменные… ваши данные
Nonheap/perm	Код, метаданные… для JVM

Если мы хотим разобраться с памятью в Java, нужно сосредоточиться на Heap (куче). Она делится на два поколения, которые классифицируют по продолжительности жизни: молодое и старое. *Молодое поколение* (оно же *детское*) содержит недолговечные объекты. *Старое поколение* содержит структуры, которые сохранились дольше.

Молодое поколение разделено на две части:

Eden	Где создаются объекты
Survivor	Промежуточное, неопределенное состояние, через которое экземпляр будет переходить из *молодого* в *старое* поколение

Сборщик мусора

Сборщик мусора (GC) — это система, очищающая память. Существуют разные реализации, но в целом он выполняет две задачи:

Minor-сборка	Просматривает молодое поколение
Major-сборка	Просматривает всю память, молодую и старую

GC выполняется одновременно с обычным выполнением приложения. При каждом выполнении GC все запущенные потоки становятся на паузу (как правило, измеряемую миллисекундами). Пока ваше приложение

сохраняет работоспособность, GC обычно ограничивает свои действия второстепенными коллекциями, чтобы не мешать ему.

Стратегии GC

Для правильной работы и очистки памяти нам требуются небольшие, недолговечные объекты, а не долгоживущие. Временные объекты будут оставаться в Eden, чтобы GC удалял их раньше и быстрее.

Наличие неиспользуемых объектов в памяти не нарушает выполнение приложения, но может повлиять на производительность вашего оборудования, а также замедлить выполнение GC, так как он будет обрабатывать их снова и снова при каждом выполнении.

Идея принудительно выполнить GC, вызвав `System.gc`, может показаться заманчивой, однако это приведет к принудительному сбору данных, нарушит эвристику и остановит ваше приложение до окончания текущего сбора.

Ссылки

GC удаляет объекты, на которые больше нет ссылок. Это означает, что, если вы создадите объект с атрибутом, ссылающимся на второй объект, оба объекта будут удалены либо одновременно, либо никогда. Чем больше объектов с перекрестными ссылками, тем более сложна и подвержена ошибкам задача GC. Вы можете помочь GC, обнулив атрибуты объектов, чтобы разорвать связи между ними.

Все статические объекты живут вечно. Это означает, что все их атрибуты, на которые они ссылаются, также будут жить вечно.

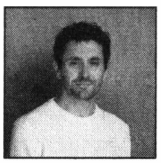
Чтобы помочь GC собирать нежелательные объекты, применяйте специальные типы ссылок, соответствующие классы которых можно найти в `java.lang.ref`:

Weak-ссылка	Не считается ссылкой для очистки. Например, можно использовать WeakHashMap (*https://oreil.ly/6PGRj*), работающую как HashMap (*https://oreil.ly/B_6ss*), но использующую weak-ссылку. Таким образом, если map содержит объект, на который есть только ссылка в самом map, его можно удалить.
Soft-ссылка	К такой ссылке GC относится с уважением и удаляет объект в зависимости от потребности в памяти.
Phantom-ссылка	Всегда возвращает null. Ссылка на самом деле не указывает на объект. Используется для очистки объектов перед извлечением объекта, который его связывает.

> Помните, что сборщик мусора — ваш друг. Он пытается облегчить вам жизнь. Вы можете отплатить ему тем же, упростив его работу.

Об авторах

Авраам Марин-Перез



Авраам Марин-Перез — программист на Java, консультант, писатель и оратор с более чем десятилетним опытом работы в различных отраслях, от финансов до издательского дела и государственного сектора. Окончив факультет компьютерных наук в Университете Валенсии, Испания, Авраам переехал в Лондон, чтобы работать в J.P. Morgan, одновременно получая степень бакалавра в области телекоммуникаций.

После трех лет работы в сфере финансов он переключился на букмекерские онлайн-конторы, которым посвятил еще три года, а затем стал независимым подрядчиком. Авраму очень помогло участие в лондонском сообществе программистов. Решив поделиться своим опытом, чтобы «вернуть долг», он стал редактором новостей Java в InfoQ, выступал на таких конференциях, как Devoxx и CodeOne (ранее JavaOne), написал книги *Real-World Maintainable Software* (издательство O'Reilly) и, в соавторстве, «Непрерывную поставку в Java» (O'Reilly). Авраам никогда не прекращает учиться и в настоящее время идет к диплому физика. Он также помогает руководить Лондонским сообществом Java и консультирует по вопросам карьеры в группе Meet a Mentor.

Реставратор кода, стр. 49

Адам Биен

Адам Биен (*adambien.blog*) — разработчик, консультант, автор, подкастер и энтузиаст Java. Он использует Java со времен JDK 1.0, а JavaScript — со времен LiveScript, но до сих пор любит писать код. Адам регулярно организует семинары по Java EE, WebStandards и JavaScript в аэропорту Мюнхена (*airhacks.com*) и проводит ежемесячное шоу вопросов и ответов в прямом эфире на *airhacks.tv*.

Следуйте скучным стандартам, стр. 78

Алексей Сошин

Алексей Сошин — архитектор программного обеспечения с 15-летним опытом работы в отрасли. Он написал книгу «Практические шаблоны проектирования с Kotlin» (издательство Packt) и создал видеокурс «Веб-разработка с помощью Kotlin». Алексей — энтузиаст Kotlin и Vert.x и опытный докладчик на конференциях.

CountDownLatch — друг или враг? — стр. 53

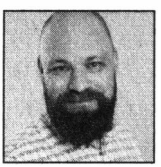
А. Махди Абдель-Азиз

А. Махди Абдель-Азиз — технический тренер и оратор. У него более чем 12-летний опыт работы в области программного обеспечения, в том числе в Google, Oracle и трех стартапах. А. Махди является соучредителем @ExtraVerd и интересуется современными технологиями, такими как PWA, автономный дизайн, машинное обучение и облачный стек.

Если он не говорит в микрофон или не сидит в самолете, вы можете застать его играющим в баскетбол. Свяжитесь с ним в Twitter: *@__amahdy* или на GitHub: *@amahdy*.

События между компонентами Java, стр. 71

Андерс Норас



Хотя изначально Андерс получил образование в области искусства и дизайна, последние 20 лет он провел за написанием кода. В настоящее время работает в Itera на должности технического директора. Он выступал с многочисленными речами и докладами на таких конференциях, как JavaZone, NDC, J-Fall, Øredev и многих других. Норас провел более 100 выступлений для самых разных аудиторий, включая представителей СМИ, дизайнеров и матерых ученых-компьютерщиков. Он известен энергичными и очень увлекательными презентациями. Помещенная здесь статья — его второй вклад в серию книг «97 вещей».

Всё, что вам нужно, — это Java, стр. 21

Энджи Джонс

Энджи Джонс — старший евангелист, специализирующийся на стратегиях и методах автоматизации тестирования. У нее завидный багаж знаний, которыми она делится, выступая с лекциями и уроками на конференциях по программному обеспечению по всему миру. Также она пишет учебные пособия и технические статьи на сайте *angiejones.tech* и руководит онлайн-платформой обучения Test Automation University. Энджи, владеющая почетным званием Master Inventor, известна своим инновационным и нестандартным стилем мышления. Результат ее неординарного подхода — более 25 запатентованных изобретений в США и Китае. В свободное время Энджи работает волонтером в Black Girls Code, проводя семинары по программированию для девушек: она стремится ввести в мир технологий как можно больше женщин и представителей меньшинств.

Применение объектно-ориентированных принципов в тестовом коде, стр. 235

Бен Эванс

Бен Эванс — главный инженер и архитектор JVM technologies в New Relic. До прихода в New Relic Бен был соучредителем jClarity (приобретен Microsoft) и главным архитектором в области биржевых деривативов в Deutsche Bank. Бен написал книги «*Хорошо* зарекомендовавший себя разработчик Java» (издательство Manning Publications), «Java: Легенда» (O'Reilly), «Оптимизация Java» (O'Reilly) и работал над последними изданиями «Java в двух словах» (O'Reilly). Ведущий специалист по Java/JVM в InfoQ, он регулярно пишет статьи для отраслевых изданий и часто выступает с докладами на технических конференциях по всему миру. Бен, соучредитель инициативы AdoptOpenJDK (совместно с Мартином Вербургом), более 20 лет занимается разработкой бесплатного программного обеспечения с открытым исходным кодом и шесть лет проработал в Исполнительном комитете JCP.

Java — дитя 90-х, стр. 110

Неуказываемые типы Java, стр. 117

Бенджамин Мушко

Бенджамин Мушко — инженер-программист, консультант и тренер, уже больше пятнадцати лет работающий в отрасли. Он по-настоящему увлечен автоматизацией проектов, тестированием и непрерывными поставками. Бен часто выступает на конференциях и яро поддерживает концепцию открытого исходного кода. Программные проекты иногда кажутся восхождением на гору — неудивительно, что в свободное время Бен любит совершать походы на Colorado's 14ers* и с удовольствием преодолевает треки на большие расстояния.

«Но на моем компьютере это работает!», стр. 44

Бенджамин Мушкала

Бенджамин Мушкала (Benny, @bmuskalla) в течение последних 12 лет воплощал в жизнь свое стремление создавать инструменты, повышающие производительность разработчиков. Он активно поддерживал IDE мирового класса Eclipse. В минувшие годы Бенджамин посвятил много времени созданию инструментов, фреймворков и подходов к тестированию, призванных помочь его коллегам повысить эффективность работы. TDD и дизайн API дороги его сердцу, как и создание ПО с открытым исходным кодом. В настоящее время Бенни трудится в Gradle Inc. над инструментом сборки Gradle.

Рефакторинг для ускорения чтения, стр. 188

Билли Корандо

Билли Корандо — евангелист в IBM с более чем десятилетним опытом работы. Билли всеми силами старается помочь разработчикам найти способы сократить напрасные затраты умственной энергии в результате утомительных процедур (например, инициирования проекта, развертывания, тестирования и валидации и т. д.), опираясь на автоматизацию и передовые методы управления. Вне работы Билли любит путешествовать, играть в кикбол и болеть за Kansas City Chiefs. Кроме того, он — один из организаторов Группы пользователей Java в Канзас-Сити.

Улучшение повторяемости и контролируемости посредством непрерывной поставки, стр. 97

---

* Colorado 14ers — пятьдесят три горные вершины высотой более 14 тысяч футов (4267 м), расположенные в штате Колорадо. — *Прим. ред.*

Брайан Вермеер

Брайан Вермеер, евангелист Snyk и инженер-программист, обладает более чем десятилетним практическим опытом создания и сопровождения ПО. Он увлечен Java, (чистым) функциональным программированием и кибербезопасностью. Брайан — амбассадор Oracle Groundbreaker, соруководитель Utrecht JUG и MyDevSecOps, а также организатор Virtual JUG. Он постоянно выступает с докладами на международных конференциях, в основном связанных с Java, таких как JavaOne, Devoxx, Devnexus, Jfokus, JavaZone и многих других. Помимо всего этого, Брайан еще и военный резервист Королевских военно-воздушных сил Нидерландов и мастер/преподаватель тхэквондо.

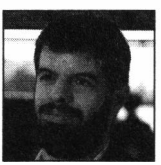
Заботьтесь о создаваемых зависимостях, стр. 197

Бурк Хуфнагель

Бурк Хуфнагель — программист и архитектор решений в Daugherty Business Solutions, где он сосредоточенно ищет способы более быстрого и качественного выполнения кода и обучает других тому, как делать то же самое. Он входит в совет директоров Atlanta Java User Group и помогает проводить конференцию Devnexus. Бурк присутствует на собраниях групп пользователей и технических конференциях, включая Connect.Tech, Devnexus, JavaOne и Oracle Code One. В 2010 году он получил титул JavaOne RockStar. В качестве писателя Бурк предоставил несколько статей для «97 вещей, которые должен знать каждый архитектор программного обеспечения» и «97 вещей, которые должен знать каждый программист» (O'Reilly). Он также выступал в качестве технического рецензента нескольких книг, в том числе *Head First Software Development* (O'Reilly) и учебного пособия Кэти Сьерра и Берта Бейтса *Sun Certified Programmer for Java Study Guide* (McGraw-Hill), за что получил неожиданный комплимент: «Бурк исправил больше нашего кода, чем мы готовы признать».

Поставляйте качественное ПО быстрее, стр. 58

Карлос Обрегон



Карлос Обрегон работает в сфере разработки программного обеспечения с 2008 года. Поскольку он всегда жаждал обмениваться знаниями, то основал Группу пользователей Java в Боготе (которая теперь называется Bogotá JVM), где выступал с докладами, в основном о лучших практиках языка Java. Помимо создания ПО, он координирует буткемпы, связанные с веб-разработкой. Хотя

в начале учебы Карлос завел интрижку с C++, еще до окончания университета встретил настоящую любовь — Java. Через несколько лет он попробовал познакомиться с другими языками JVM, но обнаружил, что ни один из них не доставляет ему столько же радости. Помимо программирования, Карлос также любит проводить время с семьей и друзьями, играя в настольные и консольные игры. Кроме того, он старается читать по крайней мере одну книгу в месяц, в основном технические издания, но также и художественную литературу. Для него нет ничего важнее Лины, Мариахозе и Эви — его жены, дочери и собаки.

Как избежать Null, стр. 92

Крис О'Делл

Крис О'Делл почти 15 лет проработала в качестве бэкенд-инженера, в основном в Microsoft Technologies, но с недавнего времени занимается Go на крупной платформе микросервисов. Она руководила командами, предоставляющими высокодоступные веб-API, распределенные системы и облачные сервисы. Также под ее контролем создавались внутренние инструменты сборки и развертывания, призванные повысить удобство работы программистов. В настоящее время Крис трудится в Monzo, помогая строить будущее банковского дела. Она регулярно выступает с докладами на конференциях по темам непрерывной доставки и практики разработки. О'Делл написала книги *Build Quality In* (издательство Leanpub) и, в соавторстве, *Continuous Delivery with Windows and .NET* (O'Reilly).

Частые релизы снижают риск, стр. 80

Кристин Горман

Кристин Горман профессионально пишет ПО в течение 20 лет и набиралась опыта во всех возможных местах, от стартапов до крупных предприятий, всегда трудясь над практическим написанием кода. Она наиболее известна своим восторженным стилем публичных выступлений и тем, что ведет блоги о ПО. Фундаментальная тема любых ее заявлений — важность полноценного вовлечения разработчиков в создание ПО. Трагедия в том, что навыки программистов недоиспользуются — они вынуждены выбирать изолированные задачи с доски, подготовленной кем-то другим, писать код в стилях, языках и фреймворках, при выборе которых не имели права голоса, и обходиться без встреч с пользователями своего программного обеспечения. Кристин

страстно стремится привлечь разработчиков к более активному участию, раскрыть их потенциал и побудить их заботиться о каждом аспекте того, над чем они трудятся, — не только для того, чтобы они получали больше удовольствия от своего занятия, но и чтобы создаваемые ими продукты приносили больше пользы (что более важно). В настоящее время Кристин сотрудничает с норвежской консалтинговой компанией Kodemaker.

Не знаете, который час? — стр. 60

Колин Випурс

Колин Випурс только что отпраздновал двадцать первую годовщину своей карьеры разработчика. Он часто путешествовал по Великобритании, работал в сферах финансов, прессы, музыки и аэронавтики, а в настоящее время трудится в Masabi, в сфере общественного транспорта. Когда-то он много работал на C/Perl, затем перешел на Java, чуточку поиграл в Scala, но теперь почти всегда программирует на Kotlin. Однажды он написал книгу, а сейчас выступает на конференциях, когда ему не лень готовить материалы. Его увлечения — TDD/BDD, создание масштабируемых высокопроизводительных систем и еда.

Почему я не придаю никакого значения сертификации, стр. 242

Дэниел Брайант

Дэниел Брайант работает архитектором продукта в Datawire, а также является менеджером по новостям в InfoQ и председателем QCon London. В настоящее время он прикладывает свой технический талант к инструментам DevOps, облачным/контейнерным платформам и внедрению микросервисов. Дэниел — Java Champion и лидер Лондонского сообщества Java (LJC). Также он участвует в нескольких проектах с открытым исходным кодом, пишет для известных технических веб-сайтов вроде InfoQ, O'Reilly и DZone, и регулярно выступает на международных конференциях, таких как QCon, JavaOne и Devoxx.

Преимущества систематизации и проверки архитектурного качества, стр. 34

Дело против fat JAR, стр. 47

Дэниел Инохоса

Дэниел Инохоса — программист, консультант, преподаватель, оратор и писатель с более чем двадцатилетним опытом работы — сотрудничает с частными, образовательными и государственными учреждениями. Дэниел любит языки JVM, такие как Java, Groovy и Scala, но также работает с ЯП не для JVM — например, Haskell, Ruby, Python, LISP, C и C++. Он заядлый практик Pomodoro Technique и каждый год прилагает все усилия, чтобы изучить еще один язык программирования. Дэниел написал книгу «Тестирование в Scala» (O'Reilly) и разработал видеоурок «Начало программирования на Scala» из серии видеоуроков O'Reilly Media. В свободное время он любит читать, плавать, собирать Lego, играть в футбол и готовить.

Знай flatMap свой, стр. 127

Дэйв Фарли

Дэйв Фарли — признанный авторитет в области непрерывной поставки. Он является соавтором книги «Непрерывная поставка» (издательство Addison-Wesley), отмеченной наградой Jolt, постоянным докладчиком на конференциях и блогером, одним из авторов Reactive Manifesto и концепции BDD. Дэйв уже более 35 лет получает удовольствие от работы с компьютерами: он трудился над большинством типов программного обеспечения, встроенного ПО, коммерческих приложений и торговых систем с низкой задержкой. Фарли начал работать в крупномасштабных распределенных системах более 30 лет назад, исследуя разработку слабосвязанных систем на основе сообщений — предшественников микросервисов. Дэйв — бывший директор по инновациям в ThoughtWorks и руководитель отдела разработки программного обеспечения в LMAX Ltd., родине OSS Disruptor, компании, хорошо известной совершенством своего кода и образцовым характером процесса разработки. В настоящее время Дэйв выступает независимым консультантом, а также руководит Continuous Delivery Ltd., которую сам основал.

Принимайте разделение ответственности всерьез, стр. 199

Разработка на основе тестирования, стр. 204

Пишите «читаемый код», стр. 247

Давид Делабассе

Давид Делабассе трудится в экосистеме Java уже более двух десятилетий. Он живет и дышит Java! В настоящее время он занимает должность евангелиста в группе Java Platform Group в Oracle. На протяжении многих лет Делабассе активно продвигал Java по всему миру, выступая на конференциях и в группах пользователей. Давид — автор многочисленных технических статей и тренингов, а иногда он пишет в блоге на *delabassee.com*. В свободное время он активно участвует в деятельности многих НКО, занимающихся улучшением ситуации с правами людей с ограниченными возможностями. Также он помогает движению за доступность городской среды. Давид живет в Бельгии, где с удовольствием играет на консоли с Лайлу, своей очаровательной (но непростой) дочерью.

Будьте внимательны к контейнерному окружению, стр. 27

Доун и Дэвид Гриффитс

Доун и Дэвид Гриффитс — авторы *Head First Kotlin* and *Head First Android Development* (O'Reilly). Они также написали другие книги из серии *Head First* и разработали анимированный видеокурс *The Agile Sketchpad* для обучения ключевым концепциям и техникам по методике, обеспечивающей активность и занятость вашего мозга.

Мысли о сопрограммах, стр. 211

Дин Уэмплер

Дин Уэмплер (*@deanwampler*) — эксперт в области потоковых систем, специализирующийся на ML/AI. Он возглавляет отдел по связям с разработчиками в Anyscale.io, который разрабатывает Ray для распределенного Python. Ранее он занимал должность вице-президента по инжинирингу в Lightbend, где руководил разработкой Lightbend Cloudflow, интегрированной системы для приложений потоковой передачи данных с популярными инструментами с открытым исходным кодом. Дин написал книги для O'Reilly и внес свой вклад в несколько проектов с открытым исходным кодом. Он часто выступает

на симпозиумах и выпускает самоучители, а также играет роль соорганизатора нескольких конференций и групп пользователей в Чикаго. Дин получил докторскую степень по физике в Вашингтонском университете.

Научитесь использовать SQL-мышление по максимуму, стр. 69

Дональд Рааб

Дональд Рааб более 18 лет проработал в качестве инженера-программиста в сфере финансовых услуг. Начал использовать Java в 1997 году, а за последующие годы профессионально применял более двадцати ЯП. Он является членом экспертной группы JSR 335, а также создателем Java-библиотеки Eclipse Collections, которую первоначально выпустили с открытым исходным кодом как GS Collections в 2012 году и перенесли в Eclipse Foundation в 2015 году. В 2018 году Дональда удостоили звания Java Champion. Он часто выступает в качестве докладчика и приглашенного тренера на ключевых конференциях Java и встречах групп пользователей, включая Oracle CodeOne, JavaOne, QCon New York, Devnexus, Devoxx US, EclipseCon, JVM Language Summit и Great Indian Developer Summit (GIDS).

Учитесь создавать kata и создавайте kata, чтобы учиться, стр. 138

Эдсон Янага

Эдсон Янага — директор Red Hat по созданию рабочей среды для разработчиков, обладатель титулов Java Champion и MVP Microsoft. Кроме того, он состоявшийся автор и частый докладчик на международных конференциях. Его выступления посвящены Java, микросервисам, облачным вычислениям, DevOps и мастерству разработки программного обеспечения. Янага считает себя «народным умельцем» по части ПО и убежден, что мы все поможем сотворить лучший мир для жизни людей, разрабатывая лучшее программное обеспечение. Цель своей жизни он видит в том, чтобы помогать коллегам по всей планете, предоставляя им возможности для того, чтобы они создавали более качественное ПО быстрее и безопаснее. И он даже может утверждать, что это его работа!

Поведение — это «легко»; состояние — это сложно, стр. 29

Эмили Бач

Эмили Бач — технический тренер по гибкому управлению в ProAgile. Она помогает командам улучшить методики совместного написания кода и обучает разработке на основе тестирования. Живет Эмили в Гетеборге (Швеция), но родом из Великобритании. Она написала и самостоятельно выпустила «*Руководство по программированию в Dojo*» (самоизданное), а также часто выступает на международных конференциях.

Тестирование на одобрение, стр. 23

Используйте покрытие, чтобы улучшить ваши модульные тесты, стр. 228

Эмили Цзян

Java Champion (https://oreil.ly/HheKg) Эмили Цзян — архитектор и евангелист Liberty Microservices, старший технический сотрудник (STSM) IBM. Место ее обитания — комплекс IBM в Херсли, Великобритания. Цзян, гуру MicroProfile, работает над ним с 2016 года. Она руководит спецификациями MicroProfile Config, Fault Tolerance и Service Mesh, а также входит в экспертную группу CDI. Эмили увлечена Java, MicroProfile и Jakarta EE. Она регулярно выступает на таких конференциях, как QCon, Code One, Devoxx, Devnexus, JAX, Voxxed, EclipseCon, GeeCON, JFokus и других. Вы можете найти ее в Twitter *@emilyfhjiang* и LinkedIn (*http://www.linkedin.com/in/emily-jiang-60803812*).

Делайте код простым и читабельным, стр. 150

Гейл С. Андерсон

Гейл С. Андерсон — Java Champion, амбассадор Oracle Groundbreaker и бывшая участница «Команды мечты NetBeans». Она один из основателей и директор по исследованиям Anderson Software Group, ведущего поставщика учебных курсов по Java, JavaFX, Python, Go, современному C++ и другим языкам программирования. Гейл любит исследовать передовые технологии Java и писать о них. В настоящее время она особенно увлечена JavaFX с GraalVM для кроссплатформенных мобильных приложений. Гейл участвовала в создании восьми учебников по программированию, а совсем недавно стала соавтором «Всеобъемлющего руководства по современным Java-клиентам с JavaFX.

Кроссплатформенная мобильная и облачная разработка» (Apress). Гейл выступала по всему миру на различных конференциях по Java, включая Devoxx, Devnexus, JCrete и Oracle Code / JavaOne. Адрес ее веб-сайта: *asgteach.com*.

Научитесь использовать новые функции Java, стр. 143

Доктор Гейл Оллис

Доктор Гейл Оллис занимается программированием с тех пор, как выучила BASIC на единственном школьном компьютере в математической лаборантской. С тех пор она познала множество ЯП, а на разных этапах карьеры участвовала в профессиональной разработке ПО, исследованиях в области психологии разработки ПО, а также читала лекции студентам и магистрантам по программированию и киберпсихологии. При этом на протяжении лет она не утратила страстного желания помогать людям лучше программировать, проявляющегося в самых разнообразных областях ее деятельности — от обучения информатике и подготовки начинающих разработчиков до проведения актуальных для отрасли академических исследований, призванных обеспечить практическую поддержку кибербезопасности при профессиональной разработке ПО.

Не скрывайте от себя все инструменты, используя IDE, стр. 63

Хизер Ванчура

Хизер Ванчура, директор и председатель программы Java Community Process (JCP), в рамках своей роли отвечает за руководство сообществом. Она также выступает в качестве международного докладчика, наставника и лидера Hack days. Ванчура несет ответственность за работу Исполнительного комитета JCP, сайта *JCP.org*, управление JSR, развитие сообщества, мероприятия, коммуникации и расширение членства. Она также является участником и руководителем общественных программ, призванных находить лидеров для групп пользователей Java (JUG). Наконец, Ванчура — ведущий специалист по JSR в рамках постоянно действующего проекта JCP.Next по развитию самой программы JCP. Хизер живет в Области залива Сан-Франциско, посвящает себя Java и сообществам разработчиков, а в свободное время любит пробовать новые виды спорта и фитнеса. Вы можете найти ее в Twitter: *@heathervc*.

Что такое программа JCP и как в ней участвовать, стр. 240

Доктор Хайнц М. Кабуц

Доктор Хайнц М. Кабуц — автор увлекательной и в некотором роде полезной рассылки #Java Specialists' Newsletter, которую можно найти на *javaspecialists.eu*. Если желаете, обратитесь к нему по электронной почте: *heinz@javaspecialists.eu*.

Ежедневно читайте OpenJDK, стр. 178

Холли Камминс

Холли Камминс работает с IBM и возглавляет сообщество разработчиков в IBM Garage. Будучи участником Garage, Холли полагается на технологии для обеспечения инноваций для клиентов в самых разных отраслях, от банковского дела до общественного питания, розничной торговли и НПО. Она руководила проектами по подсчету рыбы с использованием ИИ, помогла слепому спортсмену в одиночку пробежать ультрамарафон в пустыне, улучшила медицинское обслуживание пожилых людей и изменила принцип работы городских парковок. Холли также владеет титулами Oracle Java Champion, амбассадор IBM Q и JavaOne RockStar. До прихода в IBM Garage она руководила отделом доставки WebSphere Liberty Profile (теперь Open Liberty). Холли участвовала в написании книги «Enterprise OSGi в действии» (издательство Manning) и до сих пор рада объяснить, почему OSGi великолепен. До того как присоединиться к IBM, Камминс получила степень доктора наук в области квантовых вычислений. Холли аккуратна со своими шерстяными шарфами и еще не потеряла ни одного, но регулярно где-то оставляет зимнее пальто (бр-р-р).

Сборщик мусора — ваш друг, стр. 86

Java должна приносить радость, стр. 115

Иэн Ф. Дарвин

Иэн Ф. Дарвин работал в связанных с компьютерами областях целую вечность, трудился над системами почти любого размера, формы и для каких угодно ОС. Он программирует на нескольких языках, включая Java, Python, Dart/Flutter и скрипты для командной строки. Кроме того, Иэн внес свой вклад в OpenBSD, Linux и другие проекты с открытым исходным кодом. Он работал

в Университетской сети здравоохранения Торонто, где создал первую Android-версию Medly, спасительного приложения mHealth. Широко известен как автор «Поваренной книги Java» и «Поваренной книги Android» (O'Reilly). Кроме того, Дарвин разрабатывал и преподавал курсы Unix и Java для организации Learning Tree, а также программу для старшекурсников по Unix и C для Университета Торонто. Наконец, Иэн пишет о путешествиях, электромобилях, средневековой литературе и чуть ли не о любой «более гладкой гальке или необычно красивой ракушке», о которую споткнется на морском берегу. Найдите его на *darwinsys.com* или в Twitter *@Ian_Darwin*.

Вылезайте из песочницы Java, стр. 209

Икшель Руис

Икшель Руис разрабатывает программные приложения и инструменты с 2000 года. В круг ее научных интересов входят Java, динамические языки, клиентские технологии и тестирование. Она — Java Champion, амбассадор Groundbreaker, энтузиаст Hackergarten, сторонник открытого исходного кода, лидер JUG, оратор и наставник.

Создавайте разнообразные команды, стр. 39

Джеймс Эллиотт

Джеймс Эллиотт, старший инженер-программист Singlewire в Мэдисоне, штат Висконсин, тридцать лет накапливал профессиональный опыт в качестве системного разработчика. Он любит все, от ассемблера для 6502 до Java, а в настоящее время радостно применяет Clojure — как на работе, так и в самостоятельных проектах с открытым исходным кодом вроде Deep Symmetry. Иногда Джеймс выступает в роли диджея или продюсирует электронные музыкальные шоу со своим соратником Крисом. Джеймс написал (в одиночку и в соавторстве) несколько книг и обновленных изданий для O'Reilly и с удовольствием наставляет новые поколения разработчиков в постоянно меняющемся (но принципиально неподвластном времени) мире программного обеспечения.

Усильте Javadoc AsciiDoc'ом, стр. 25

Заново откройте для себя JVM с помощью Clojure, стр. 184

Джанна Патчей

Джанна Патчей — общепризнанный отраслевой эксперт и консультант в области финансовых рынков. В список ее специализаций входят инновации на финансовых рынках и оказание помощи фирмам в определении, разработке и реализации их коммерческих стратегий в строго регулируемой среде. Особую страсть она испытывает к структуре рынка (участникам финансовых рынков, тому, как они взаимодействуют, и последствиям этого), а также к поиску творческих решений проблем, связанных с доступом к рынкам и ликвидностью. Ее интересуют не только традиционные финансовые рынки и классы активов, но и развивающаяся область рынков цифровых активов. Кроме того, Джанна — директор и посланник по защите нормативных требований London Blockchain Foundation. Также она пишет статьи на темы финансовых и технологических инноваций для журнала Best Execution. Джанна получила степень бакалавра математики и компьютерных наук в Кейптаунском университете и степень магистра права в области международного банковского и финансового права в Ливерпульском университете.

Три черты по-настоящему отличных разработчиков, стр. 216

Жанна Боярски

Жанна Боярски живет в Нью-Йорке и носит титул Java Champion. Она написала пять книг о сертификации Java и семнадцать лет получала деньги за то, что программировала на этом языке. Сейчас Боярски работает волонтером в *coderanch.com* и с командой робототехники FIRST. Жанна регулярно участвует в конференциях и владеет званием Distinguished Toastmaster, которого удостаивается тот, кто провел больше пятидесяти выступлений.

Разбивайте проблемы и задачи на небольшие фрагменты, стр. 37

Дело сделано, но..., стр. 106

Изучайте идиомы Java и храните их в памяти, стр. 136

Дженн Стретер

Дженн Стретер — давний член сообщества Groovy и менеджер сообщества Groovy slack. Она внесла свой вклад в различные проекты с открытым исходным кодом, включая CodeNarc, Gradle, Groovy и Spring REST Docs. В качестве докладчика на конференциях Дженн выступала на таких мероприятиях, как Devoxx Belgium, Grace Hopper Celebration of Women in Computing, SpringOne

Platform и O'Reilly Velocity Conference. В 2013 году она основала организацию GR8Ladies (ныне GR8DI), служащую ей платформой для наставления студентов и младших разработчиков. Дженн окончила колледж Гамильтона в Клинтоне, штат Нью-Йорк, и получала грант Фулбрайта в 2016–2017 годах. В настоящее время она проживает в агломерации Миннеаполиса и Сент-Пола.

Сборки не обязательно должны быть медленными и ненадежными, стр. 42

Создавайте только те части, которые изменяются, и повторно используйте остальные, стр. 163

Проекты с открытым кодом — это не волшебство, стр. 165

Дженнифер Райф

Дженнифер Райф — заядлый разработчик и специалист по решению проблем. Она участвовала в проектах (как для сообществ разработчиков, так и для крупных предприятий), призванных организовать и осмыслить глобальные информационные ресурсы, чтобы использовать их с максимальной отдачей. Райф применяла всевозможные инструменты, как коммерческие, так и с открытым исходным кодом, и с удовольствием познаёт новые технологии, иногда на ежедневной основе! Повседневная деятельность Дженнифер прежде всего связана с изучением и написанием кода, и ей нравится создавать материалы, которыми она может поделиться с другими. Часто сюда входят выступления на конференциях и мероприятиях, посвященных разработчикам, а также написание статей. Ее страсть — находить способы упорядочить хаос и более эффективно предоставлять программное обеспечение. Если говорить о других увлечениях, то Райф любит кошек, путешествия с семьей, походы, чтение, выпечку и верховую езду.

В языковых войнах Java может за себя постоять, стр. 99

Джессика Керр

Джессика Керр — симматезист* в среде программирования. Это значит, что она выступает за обучающие системы, которые состоят из обучающихся частей: увлеченных людей и развивающегося программного обеспечения. За 20 лет, отданных профессиональной разработке ПО, она писала на любых языках: от

* Неологизм «symmathecist» образован от греческого слова mathe — «изучать, познавать» и приставки sym-, которая на русский язык переводится как «со-». Автор неологизма, Джессика Керр, считает разработку процессом совместного познания: «This is a symmathesy. "Sym" = together, "mathesy" = learning». — *Прим. науч. ред.*

Java до Scala и Clojure, от Ruby до Elixir и Elm, от Bash до TypeScript и PowerShell. В те годы, когда Джессика участвовала в конференциях как ведущий и докладчик, она рассказывала обо всем этом, а также о менее очевидных принципах разработки программного обеспечения. Она черпает вдохновение в инженерии устойчивости, системном мышлении и искусстве. Ей нравится помогать разработчикам автоматизировать скучные задачи и проявлять более творческий подход во всем прочем. Послушайте в Twitter, как она учится, комментируя вслух (*@jessitron*), посмотрите на Twitch, как она программирует в прямом эфире, и почитайте ее блог *blog.jessitron.com*. Керр живет в своем доме в Сент-Луисе, штат Миссури, где воспитывает двух новых непредсказуемых людей.

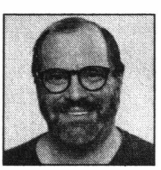
От головоломок к продуктам, стр. 82

Джош Лонг

Джош Лонг (*@starbuxman*) — инженер с многолетним опытом написания кода, а также первый евангелист Spring, Java Champion, автор книг (в том числе «Java в облаке. Проектирование устойчивых систем с помощью Spring Boot, Spring Cloud и Cloud Foundry», выпущенной издательством O'Reilly, и самостоятельно опубликованной *Reactive Spring*) и многочисленных видеоуроков-бестселлеров (например, «Создание микросервисов с помощью Spring Boot Livelessons», где участвовал Фил Уэбб, один из разработчиков Spring Boot). Джош нередко посещает конференции: он уже выступил в сотнях городов по всему миру, на всех континентах (кроме Антарктиды). Он любит писать код и вносит свой вклад в различное ПО с открытым исходным кодом (Spring Framework, Spring Boot, Spring Integration, Spring Cloud, Activiti, Vaadin, MyBatis и т. д.). Наконец, Лонг ведет подкаст (*A Bootiful Podcast*) и канал на YouTube (*Spring Tips, http://bit.ly/spring-tips-playlist*).

Продакшн — самое радостное место на земле, стр. 172

Кен Коузен



Кен Коузен — Java Champion, амбассадор Oracle Groundbreaker, Java RockStar и Grails RockStar. Он написал книги *Kotlin Cookbook, Modern Java Recipes* и *Gradle Recipes for Android* (издательство O'Reilly) а также *Making Java Groovy* (издательство Manning), и подготовил несколько видеокурсов для платформы O'Reilly Learning. Он постоянно выступает с докладами в конференц-

туре No Fluff Just Stuff и участвовал в симпозиумах по всему миру. В созданной им компании Kousen IT, Inc. разработке ПО обучались тысячи студентов и работающих специалистов.

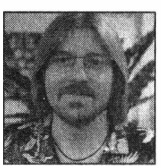
Добавьте в вашу Java немного Groovy, стр. 153

Кенни Бустани

Кенни Бустани — истовый проповедник технологий и адепт программного обеспечения с открытым исходным кодом в Кремниевой долине. В качестве консультанта по корпоративному ПО он развил обширный набор навыков, необходимых для проектов, требующих веб-разработчика полного цикла в Agile-методологии. Будучи завзятым блогером и автором ПО с открытым исходным кодом, Кенни взаимодействует с сообществом увлеченных программистов, которые хотят воспользоваться преимуществами новых методов обработки графиков для анализа данных.

Компромиссы в архитектуре микросервисов, стр. 218

Кевин Виттек



Кевин Виттек, сопровождающий Testcontainers и автор Testcontainers-Spock, увлекается FLOSS и Linux. Он получил награду Oracle Groundbreaker Ambassador Award за свой вклад в сообщество разработчиков ПО с открытым кодом. Кевин — искусник программного обеспечения и фанат тестирования. Он влюбился в TDD из-за Спока. Кевин верит в экстремальное программирование как в одну из лучших гибких методологий. Ему нравится писать программы MATLAB, чтобы помогать жене с научными экспериментами по исследованию поведения голубей. В своей второй жизни Виттек — музыкант-исполнитель, играющий на электрогитаре. После многих лет работы в отрасли в качестве инженера Кевин сейчас защищает в RWTH Aachen докторскую диссертацию на тему проверки смарт-контрактов и возглавляет исследовательскую лабораторию блокчейна в Институте интернет-безопасности в Гельзенкирхене при Вестфальском университете прикладных наук.

Как высвободить потенциал интеграционного тестирования с использованием контейнеров, стр. 223

Кевлин Хенни

Кевлин Хенни (*@KevlinHenney*) — независимый консультант, тренер, программист и писатель. Его интересы в отношении разработки ПО связаны с программированием, языками и практикой: он помогает отдельным лицам, командам и организациям совершенствоваться в этих областях. Кевлин по-настоящему влюблен в программирование и языки, что, как он с радостью обнаружил, обеспечивает его занятостью уже дольше трех десятилетий. Хенни выступал с докладами, проводил консультации и семинары на сотнях конференций и встреч по всему миру. Он писал обозрения для различных изданий, журналов и веб-сайтов, создавал ПО с открытым и закрытым исходным кодом, а также участвовал в работе столь большого количества групп, организаций и комитетов, что, пожалуй, навредил себе (как говорится, «комитет — это тупик, куда заманивают идеи, чтобы тихо придушить»). Кевлин — один из соавторов *A Pattern Language for Distributed Computing* и *On Patterns and Pattern Languages,* двух томов из серии «*Pattern-Oriented Software Architecture*» (издательство Wiley), и редактор «97 вещей, которые должен знать каждый программист» (O'Reilly).

Назовите дату, стр. 159

Программируйте с GUT, стр. 175

Не проверяйте свои исключения, стр. 220

Кирк Пеппердайн

Кирк Пеппердайн уже более 20 лет занимается настройкой производительности Java-приложений. Именно он изначально организовал семинар по настройке производительности Java (JPTW). В 2006 году Кирк удостоился титула Java Champion за передовые идеи в области производительности Java. Он часто выступает в группах пользователей и на конференциях, и его много раз называли JavaOne Rockstar. Пеппердайн по-прежнему активно поддерживает сообщества Java в роли соучредителя JCrete, «неконференции» Java, которая использовалась в качестве шаблона для ряда других неконференций в Европе, Азии и Северной Америке. В 2019 году Кирк продал свой стартап jClarity корпорации Microsoft, где он сейчас работает главным инженером.

Эй, Фред, не мог бы ты передать мне HashMap? — стр. 90

Лиз Кио

Лиз Кио — консультант из Лондона по тематике Lean и Agile, известный блогер и международный оратор, основной член сообщества BDD, а также страстный сторонник фреймворка Cynefin и его способности изменять мысленные установки программистов. У Лиз отличный багаж технических знаний и навыков — она двадцать лет приносила прибыль компаниям и учила этому других повсюду, от небольших стартапов до глобальных предприятий. Сейчас она в основном сосредотачивает усилия на Lean, Agile и организационных преобразованиях, а также стремится к тому, чтобы сделать изменения инновационными, легкими и увлекательными, опираясь на прозрачность, позитивный язык, хорошо сформированные результаты и защищенные от неудачи эксперименты.

Циклы обратной связи, стр. 74

Мацей Валковяк

Мацей Валковяк — независимый консультант по программному обеспечению. Он помогает компаниям принимать архитектурные решения, а также проектировать и разрабатывать системы, часто основанные на Spring stack. Будучи активным членом сообщества Spring, он участвовал в нескольких проектах Spring. В последние годы он все заметнее увлекается преподаванием и обменом знаниями. Мацей ведет канал на YouTube (Spring Academy), выступает на конференциях и проводит слишком много времени в Twitter.

«Разработчик полного цикла» — это образ мышления, стр. 84

Мала Гупта

Мала Гупта — евангелист в JetBrains, а также основатель и ведущий наставник в *eJavaGuru.com*, где она обучает претендентов на получение сертификатов Java, помогая им успешно пройти тесты. Будучи Java Champion, Мала продвигает изучение и использование технологий Java на различных платформах посредством своих книг по Java, курсов, лекций и выступлений. Она твердо верит в равенство обязанностей и возможностей для всех. Гупта уже более девятнадцати лет связана с индустрией ПО в качестве автора, докладчика, наставника, консультанта, технологического лидера и разработчика. Мала сотрудничает с издательством Manning Publications, и ее книги по Java

помогают получить наивысший рейтинг сертификации Oracle по всему миру. Частый докладчик на отраслевых конференциях, она также участвует в руководстве отделения Java User Group в Дели. Будучи убежденной сторонницей движения Women in Technology, она руководит инициативами Делийского отделения Women Who Code, направленными на расширение участия женщин в сфере технологий.

Сертификаты Java: пробирный камень технологий, стр. 108

Марко Билен

Марко Билен — разработчик программного обеспечения со страстью к поддерживаемому и читабельному коду. Марко подвизается на этом поприще с 2005 года. До этого Билен работал системным администратором, что привило ему понимание того, как важна наблюдаемость программных систем. Марко был ведущим различных тренингов и встреч по программированию, в том числе мини-сериала о разработке на основе тестирования. У него есть жена и двое детей. Что касается использования компанией собственных приложений, то предложенный Джо Хопп вариант «пить собственное шампанское» нравится ему больше, чем «поедать собственный собачий корм» (тем более что он любит пить шампанское). В сети Билен отыщется под именем *@mcbeelen*.

Упаковка по функциям с модификатором доступа по умолчанию, стр. 170

Мария Ариас де Рейна

Мария Ариас де Рейна — старший инженер-программист Java, фанатка геопространственных проектов и сторонник открытого исходного кода. С 2004 года она является лидером сообщества и основным сопровождающим нескольких бесплатных проектов с открытым исходным кодом. В настоящее время Мария работает в Red Hat, где специализируется на промежуточном программном обеспечении, а также поддерживает Apache Camel и Syndesis. Она опытный ведущий и оратор. В период с 2017 по 2019 год де Рейну выбирали президентом OSGeo, Геопространственного фонда с открытым исходным кодом, который служит основой для многих наиболее актуальных геопространственных программ. Кроме того, она феминистка и активистка движения Women In Technology.

Молодые, старые и мусор, стр. 250

Марио Фуско

Марио Фуско — главный инженер-программист Red Hat, руководитель проекта Drools. У него огромный опыт работы в качестве Java-разработчика: он участвовал во многих проектах корпоративного уровня (зачастую возглавляя их) в нескольких отраслях, начиная от медиакомпаний и заканчивая финансовым сектором. В круг его интересов входят функциональное программирование и языки, специфичные для конкретной предметной области. Черпая энергию в своих увлечениях, он создал библиотеку с открытым исходным кодом lambdaj, стремясь предоставить разработчикам внутренний Java DSL для управления коллекциями, чтобы хоть немного реализовать функциональное программирование на Java. Кроме того, он Java Champion, координатор JUG в Милане, регулярный докладчик и соавтор книги *Modern Java in Action*, опубликованной издательством Manning.

Параллелизм в JVM, стр. 51

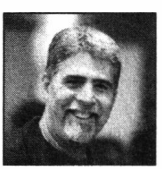
Давайте заключим контракт: искусство разработки Java API, стр. 148

Марит ван Дейк

Марит ван Дейк уже почти двадцать лет занимается разработкой программного обеспечения на различных ролях и в разных компаниях. Она любит создавать потрясающее ПО в контакте с удивительными людьми и является основным участником Cucumber с открытым исходным кодом, а также эпизодически примыкает к другим проектам. Ей нравится узнавать что-то новое, а также делиться знаниями по программированию, автоматизации тестирования, Cucumber/BDD и разработке ПО. Она выступает на международных конференциях, на вебинарах, в подкастах и блогах на *medium.com/@mlvandijk*. В настоящее время Марит работает инженером-программистом в *bol.com*.

Тестируйте, чтобы разрабатывать более качественное ПО быстрее, стр. 233

Марк Ричардс



Марк Ричардс — опытный практический архитектор ПО, занимающийся архитектурой, проектированием и внедрением архитектур микросервисов, архитектур, управляемых событиями, и распределенных систем. Он работает в индустрии программного обеспечения с 1983 года и имеет степень магистра компьютерных наук. Марк основал *DeveloperToArchitect.com*, бесплатный

веб-сайт, призванный помогать разработчикам на пути к созданию ПО. Кроме того, он выступал на сотнях конференций по всему миру, а также написал множество книг и видеороликов по микросервисам и архитектуре ПО. Его последняя на данный момент работа, «Основы архитектуры программного обеспечения», вышла в издательстве O'Reilly.

Свободно применяйте нестандартные идентификационные аннотации, стр. 230

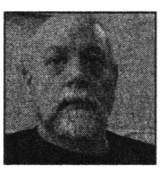
Майкл Хангер

Майкл Хангер увлечен разработкой программного обеспечения уже более тридцати пяти лет, 25 из которых он обретается в экосистеме Java. Последние 10 лет он работал над базой данных Neo4j graph с открытым исходным кодом, выполняя множество функций, а не так давно возглавил начинания Neo4j Labs. Как опекун сообщества и экосистемы Neo4j, он особенно любит работать с проектами, пользователями и участниками, связанными с графиками. В качестве разработчика Майкл увлекается многими аспектами языков программирования: каждый день изучает что-то новое, участвует в захватывающих и амбициозных проектах с открытым исходным кодом, вносит свой вклад и пишет книги и статьи, связанные с ПО. Хангер помогал организовывать конференции и выступал на многих других мероприятиях. Его усилия привели к тому, что его приняли в программу Java Champions. Майкл помогает детям научиться программировать, проводя в местных школах еженедельные занятия по написанию кода только для девочек.

Бенчмаркинг — это сложно, JMH поможет, стр. 31

На всю катушку, стр. 76

Майк Данн



Майк Данн — главный мобильный инженер и технический руководитель Android в O'Reilly Media, признанный член сообщества AOSP и преданный участник экосистемы Android с открытым исходным кодом. Именно он когда-то давно создал популярную библиотеку плиточных изображений TileView. Кроме того, Майк написал книги «Нативная мобильная разработка. Перекрестный справочник по нативной разработке для Android и iOS» (в соавторстве с Шоном Льюисом для O'Reilly) и выходящую в скором времени

«Программирование Android с Kotlin: из Java в Kotlin на примерах» (совместно с Пьер-Оливье Лоренсом, также для O'Reilly). Он внес свой вклад в закрытую библиотеку JavaScript Google и предоставлял поддержку с открытым исходным кодом для каких угодно проектов, от библиотек управления цветом до быстрого поиска, шифрования на уровне блоков с помощью медиаплеера Android следующего поколения от Google ExoPlayer и компактного движка маршрутизации PHP. Майк профессионально программирует уже почти 20 лет, однако продолжает изучать информатику в магистратуре Технологического института Джорджии. На домашней странице Майка (*http://moagrius.com*) вы найдете несколько разнообразных фрагментов кода, отмеченных различными печатями древности и медленного сползания в небытие, проектов с открытым исходным кодом и клиентских проектов, а также его блог.

Обратите внимание на Kotlin, стр. 132

Моника Беквит

Моника Беквит — Java Champion, Первый тренер Lego League и соавтор *Java Performance Companion* (издательство Addison-Wesley), а также автор готовящейся к печати монографии *Java 11 LTS+–A Performance Perspective.* Моника работает в Microsoft, где увлекается вопросами производительности JVM.

Программирование на Java в аспекте производительности JVM, стр. 112

Нэт Прайс

Нэт Прайс занимается программированием в течение кхе!.. дцати… кхе!..х лет, причем на протяжении многих из них он использовал Java и/или JVM. Он работал консультантом-разработчиком и архитектором в различных отраслях промышленности и создавал критически важные для бизнеса системы, масштаб которых варьируется от встроенных потребительских устройств до крупных вычислительных ферм, поддерживающих глобальный бизнес. Также он регулярно выступает с докладами на конференциях. Нэт — один из авторов книги «Создание объектно-ориентированного программного обеспечения на основе тестирования» (издательство Addison-Wesley), очень популярного издания, посвященного объектно-ориентированному проектированию и разработке на основе тестирования.

Необъяснимая эффективность фаззинга, стр. 225

Николай Парлог

Николай Парлог (он же nipafx) — Java Champion со страстью к обучению и обмену опытом. Этим он занимается в статьях, постах в блогах, информационных бюллетенях и книгах; в твитах, репостах, видео и стримах; на конференциях и внутренних тренингах... Подробнее вы обо всем этом узнаете на *nipafx.dev*. Впрочем, больше всего Николай известен своей стрижкой.

Виды комментариев, стр. 124

Optional — монада, нарушающая закон, но это хороший тип, стр. 167

Позаботьтесь о своих объявлениях модулей, стр. 194

Никхил Нанивадекар

Никхил Нанивадекар — коммиттер и руководитель проекта Eclipse Collections framework с открытым исходным кодом. Он работает в финансовом секторе в качестве Java-разработчика с 2012 года. Прежде чем начать карьеру в качестве разработчика ПО, Никхил получил степень бакалавра в области машиностроения в индийском Университете Пуны и степень магистра в области машиностроения со специализацией в области робототехники в Университете Юты. В 2018 году он удостоился титула Java Champion. Нанивадекар постоянно участвует в местных и международных выступлениях, а также является активным сторонником детского образования и наставничества. Он ведет несколько семинаров по обучению детей робототехнике на таких мероприятиях, как JCrete4Kids, JavaOne4Kids, OracleCodeOne4Kids и Devoxx4Kids. Никхил любит готовить со своей семьей, ходить в походы, кататься на лыжах или на мотоцикле, а еще работать с организациями по спасению и оказанию помощи животным.

Знайте свои коллекции, стр. 130

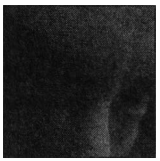
Патриция Аас

Патриция Аас — опытный программист на C++, в начале карьеры писавшая на Java. Она работала над двумя браузерами, Opera и Vivaldi, и создавала встроенные системы телеприсутствия в Cisco. Чрезвычайно любознательный человек, Патриция всегда рада узнать что-то новое. Она является соучредителем

компании TurtleSec, где и работает в настоящее время консультантом и тренером, специализируясь на безопасности приложений.

Встроенное мышление, стр. 102

Пол У. Гомер



Вот уже тридцать лет Пол У. Гомер профессионально разрабатывает программное обеспечение. Он создавал коммерческие продукты для финансов, маркетинга, печати и здравоохранения, а последние пятнадцать лет еще и писал об этом в блогах. За годы Пол перепробовал практически все аспекты разработки ПО, а также часто занимал должность ведущего программиста. В своем блоге *The Programmer's Paradox* он пробует синтезировать нечто осмысленное из этих разнообразных эпизодов своего прошлого. Там обсуждаются масштабные закономерности, которые он подмечал, переходя из одной организации в другую. Пол предпочитает серверное алгоритмическое программирование, но часто получает удовольствие от попыток сделать доменные интерфейсы полностью динамичными. Когда он не погружен в сложный код, то старается уделять время разговорам с программистами и предпринимателями об основах разработки ПО.

Необходимость технологий промышленной прочности, стр. 161

Питер Хилтон

Питер Хилтон — менеджер по продуктам, разработчик, писатель, оратор, тренер и музыкант. Круг его профессиональных интересов охватывает управление продуктами, автоматизацию рабочего процесса, функциональный дизайн ПО, гибкие методы разработки программного обеспечения, а также удобство обслуживания и документирование ПО. Питер консультирует компании, создающие ПО, и команды разработчиков, а также время от времени проводит презентации и семинары. Ранее Хилтон выступал на многочисленных европейских конференциях разработчиков, а также стал соавтором книги *Play for Scala* (издательство Manning Publications). Он научил Fast Track играть со Scala, а совсем недавно создал собственный учебный курс «Как писать поддерживаемый код».

Называйте вещи своими именами, стр. 88

Преобразование логических значений в перечисления, стр. 186

Пишите к документации комментарии в одно предложение, стр. 244

Рафаэль Беневидес

Рафаэль Беневидес — истовый сторонник облачных технологий, работающий в Oracle. Опираясь на многолетний опыт работы в нескольких областях IT-индустрии, он помогает программистам и компаниям по всему миру более эффективно разрабатывать ПО. Рафаэль считает себя человеком, который решает проблемы и очень любит делиться знаниями. Он является членом Apache DeltaSpike PMC и победителем проекта Duke's Choice Award. Беневидес выступает с докладами на таких конференциях, как JavaOne, Devoxx, TDC, Devnexus и многих других. В Twitter его можно найти под именем *@rafabene*.

По-настоящему заглянуть «под капот», стр. 180

Род Хилтон

Род Хилтон — инженер-программист, работающий со Scala и Java в Twitter. Он пишет блог о программном обеспечении, технологиях и иногда о «Звездных войнах» на *nomachetejuggling.com*. Вы можете найти его в Twitter: *@heathervc*.

В вашем каталоге bin/ отличные инструменты, стр. 207

Доктор Рассел Уиндер

Доктор Рассел Уиндер сначала был теоретиком физики частиц высоких энергий, а затем переквалифицировался в программиста Unix-систем. Таким путем он стал академиком компьютерных наук (Университетский колледж Лондона, затем Королевский колледж Лондона), который интересуется программированием; ЯП, инструментами и средами; многопоточностью; параллелизмом; сборкой; взаимодействием человека и компьютера, а также и социотехническими системами. Став профессором компьютерных наук и заведующим кафедрой компьютерных наук в Королевском колледже Лондона, он покинул академические круги, чтобы заняться стартапами в качестве технического директора или генерального директора. В дальнейшем он на протяжении десяти лет работал независимым консультантом, аналитиком, автором, тренером и свидетелем-экспертом, после чего ушел на пенсию в 2016 году. Доктор Уиндер по-прежнему очень интересуется

программированием; ЯП, инструментами и средами; многопоточностью; параллелизмом и сборкой, что помогает ему не закоснеть на заслуженном отдыхе.

Декларативное выражение — вот путь к параллелизму, стр. 56

JVM — мультипарадигмальная платформа. Используйте это, чтобы повысить свой уровень программирования, стр. 120

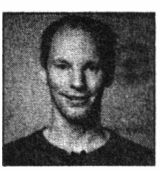
Потоки — это инфраструктура, относитесь к ним соответственно, стр. 214

Сэм Хепберн

Сэм Хепберн провела последние девять лет в Лондоне, где стала хорошо известным лицом технологической стартап-сцены. Она работала с различными организациями в Лондоне, а теперь покоряет новые земли в США, Великобритании и Польше, создавая одни из крупнейших технологических сообществ в мире. Ее главная цель — сформировать среду, где люди чувствовали бы себя желанными гостями, а сообщества процветали бы. В настоящее время она возглавляет общественную команду в Snyk.io, помогая программистам внедрять безопасность в их процессы разработки. Что касается нерабочего времени Сэм, то она — соучредитель Circle, сети для продвижения карьеры женщин в нашем новом мире труда, и ведущая *Busy Being Human*, подкаста, освещающего неотредактированную, честную, человеческую историю о том, как наши любимые люди стали теми, кто они есть.

Как развивать карьеру, опираясь на силы сообщества, стр. 238

Сандер Мак



Сандер Мак — Java Champion и директор по технологиям в Picnic, голландской онлайн-продуктовой сети, масштабно создающей системы на базе Java. Для издательства O'Reilly он написал книгу «Модульность Java 9». Будучи заядлым докладчиком на конференциях, блогером и автором Pluralsight, Сандер любит делиться знаниями.

Возрождение Java, стр. 182

Себастьяно Поджи

Себастьяно Поджи, уроженец туманных равнин Северной Италии, «наточил клыки», работая в стартапе по производству умных часов. Затем он со своими кудрями переехал в Лондон, где помогал крупным клиентам с разработкой приложений для Android, трудясь в известных агентствах AKQA и Novoda. Эксперт Google по разработкам с 2014 года, Поджи часто выступает на конференциях и время от времени пишет статьи в блогах. Вернувшись в Италию, Себастьяно в настоящее время работает в JetBrains как над программным продуктом, так и над приложением для Android. У него отличные навыки в областях дизайна, типографии и фотографии, а в прошлом он монтировал видео. Себастьяно часто можно встретить на *twitter.com/seebrock3r*, где он без спросу вылезает со своими мнениями.

Взаимодействие с Kotlin, стр. 104

Стив Фримен

Стив Фримен, соавтор книги «Создание объектно-ориентированного программного обеспечения на основе тестирования» (Addison-Wesley), прокладывал дорогу для разработки ПО по Agile-методологии в Великобритании. Ему доводилось работать в консалтинговых компаниях и с поставщиками ПО в качестве независимого консультанта и тренера, а также создавать прототипы для ведущих исследовательских лабораторий.

Стив получил докторскую степень в Кембриджском университете. В настоящее время он играет роль авторитетного консультанта в Zuhlke Engineering Ltd., базирующейся в Великобритании. В свободное время Стив прежде всего старается больше не покупать тромбоны.

Не меняйте свои переменные, стр. 65

Минимизируйте конструкторы, стр. 156

Простые объекты значений, стр. 191

Томас Ронзон

Томас Ронзон уже более 20 лет занимается модернизацией критически важных для бизнеса приложений. Кроме того, он публикует статьи и выступает на конференциях. Выказывая свой профессионализм, Томас с удовольствием, страстно и глубоко погружается в технические аспекты. Полагаясь на чуткость, опыт и конкретные предложения решений, он помогает наводить мосты между бизнесом и ИТ.

Как вывести из строя JVM, стр. 95

Триша Джи

Триша Джи разрабатывала Java-приложения для целого ряда отраслей (включая финансы, производство, программное обеспечение и некоммерческие организации) и компаний любого размера. Она обладает опытом работы в высокопроизводительных системах Java и увлечена повышением продуктивности разработчиков. Триша — евангелист JetBrains, лидер Java User Group в Севилье и Java Champion. Она считает, что здоровые сообщества и обмен идеями помогают нам учиться на ошибках и развивать успехи.

Держите руку на пульсе, стр. 122

Изучите свою IDE, чтобы уменьшить когнитивную нагрузку, стр. 146

Техническое интервьюирование — это навык, который стоит развивать, стр. 202

Уберто Барбини

Програмист-полиглот Уберто Барбини уже дольше двадцати лет разрабатывает и создает успешные программные продукты во многих отраслях промышленности. Он обнаружил, что любит программирование, когда создал свою первую видеоигру на ZX Spectrum, и его до сих пор весьма увлекают идеи того, как писать лучший код, приносящий пользу бизнесу, причем не один раз, а регулярно. Когда Уберто не занят программированием, он с удовольствием проводит публичные выступления, пишет и преподает. В настоящее время он пишет книгу о прагматичном функциональном Kotlin.

Научитесь любить ваш устаревший код, стр. 141

Предметный указатель

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Я

Производственно-практическое издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Хенни Кевлин, Джи Триша

97 ВЕЩЕЙ, О КОТОРЫХ ДОЛЖЕН ЗНАТЬ КАЖДЫЙ JAVA-ПРОГРАММИСТ
СОВЕТЫ ЛУЧШИХ ЭКСПЕРТОВ

Главный редактор *Р. Фасхутдинов*
Руководитель направления *В. Обручев*
Продюсер *В. Палитко*
Научный редактор *А. Лютикова*
Литературный редактор *Ю. Войтко*
Ответственный редактор *Д. Калачева*
Младший редактор *Д. Данилова*
Художественный редактор *А. Шуклин*
Компьютерная верстка *Е. Матусовская*
Корректоры *Т. А. Шельен, Н. Болотина*

В оформлении обложки использована иллюстрация:
Irina Trusova / HYPERLINK «http://shutterstock.com/» Shutterstock.com
Используется по лицензии от HYPERLINK «http://shutterstock.com/» Shutterstock.com

Страна происхождения: Российская Федерация
Шығарылған елі: Ресей Федерациясы

ООО «Издательство «Эксмо»
123308, Россия, город Москва, улица Зорге, дом 1, строение 1, этаж 20, каб. 2013.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru
Өндіруші: «ЭКСМО» АҚБ Баспасы,
123308, Ресей, қала Мәскеу, Зорге көшесі, 1 үй, 1 ғимарат, 20 қабат, офис 2013 ж.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru.
Тауар белгісі: «Эксмо»
Интернет-магазин : www.book24.ru

Интернет-магазин : www.book24.kz
Интернет-дүкен : www.book24.kz
Импортёр в Республику Казахстан ТОО «РДЦ-Алматы».
Қазақстан Республикасындағы импорттаушы «РДЦ-Алматы» ЖШС.
Дистрибьютор и представитель по приему претензий на продукцию,
в Республике Казахстан: ТОО «РДЦ-Алматы»
Қазақстан Республикасында дистрибьютор және өнім бойынша арыз-талаптарды қабылдаушының өкілі «РДЦ-Алматы» ЖШС,
Алматы қ., Домбровский көш., 3-«а», литер Б, офис 1.
Тел.: 8 (727) 251-59-90/91/92; E-mail: RDC-Almaty@eksmo.kz
Өнімнің қауіпсіздігінің мерзімі шектелмеген.
Сертификация туралы ақпарат сайты: www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «Эксмо»:
www.eksmo.ru/certification
Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Дата изготовления / Подписано в печать 03.05.2023.
Формат 70x100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 23,33.
Тираж 2000 экз. Заказ 5111.

Отпечатано с готовых файлов заказчика
в АО «Первая Образцовая типография»,
филиал «УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ»
432980, Россия, г. Ульяновск, ул. Гончарова, 14

ISBN 978-5-04-169254-4

ВСЕ, ЧТО ВАМ НУЖНО, – ЭТО JAVA!

Что должен знать каждый Java-программист? Ответов на этот вопрос может быть очень много. Авторы этой книги собрали мнения нескольких десятков опытных разработчиков на Java, чтобы создать единое руководство для тех, кто только начинает свой путь в программировании.

Внутри Вы найдете подробные инструкции по основным темам, касающимся работы с Java:

- **ФУНКЦИОНАЛЬНОСТЬ JAVA VIRTUAL MACHINE**
- **МЕТОДЫ ТЕСТИРОВАНИЯ КОДА**
- **НАБОР ИНСТРУМЕНТОВ JAVA DEVELOPMENT KIT**
- **ОСОБЕННОСТИ ЯЗЫКА**
- **АРХИТЕКТУРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

И еще – советы от экспертов и их истории профессионального развития в программировании на Java!

Об авторах:

КЕВЛИН ХЕННИ – консультант по разработке программного обеспечения на Java и C++ и постоянный спикер на популярных технологических конференциях.
ТРИША ДЖИ – инженер программного обеспечения и ведущий разработчик (Lead Developer Advocate) в Gradle.

ISBN 978-5-04-169254-4

БОМБОРА
издательство

O'REILLY®

БОМБОРА – лидер на рынке полезных и вдохновляющих книг.
Мы любим книги и создаем их, чтобы вы могли творить, открывать мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

bombora.ru bomborabooks bombora