Elegant Objects

by Yegor Bugayenko

ЭЛЕГАНТНЫЕ ОБЪЕКТЫ
Java Edition

Егор Бугаенко

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

2018

Оглавление

Предисловие

Об объектно-ориентированном программировании (ООП) написано много книг. Зачем нужна еще одна? Затем, что мы в опасности. Мы все дальше уходим от того, что было задумано создателями ООП, и у нас все меньше шансов вернуться. Все существующие ООП-языки предлагают рассматривать объекты как структуры данных с прикрепленными процедурами, что в корне неверно. Появляются новые языки, но они делают так же или даже хуже. Объектно-ориентированных программистов заставляют думать так, как процедурные программисты думали 40 лет назад. То есть думать не как объекты, а как компьютеры.

Эта книга представляет собой сборник практических рекомендаций, которые, как мне кажется, могут изменить ситуацию и остановить деградацию ООП. Большинство из них я прочел в различных источниках, а некоторые просто придумал.

Двадцать три совета сгруппированы в четыре главы: рождение, школа, трудоустройство и выход на пенсию. Речь пойдет о мистере Объекте, антропоморфной сущности в объектно-ориентированном мире. Он родится, пойдет в школу, устроится

на какую-нибудь работу, а затем выйдет на пенсию. Посмотрим, как будут развиваться события, и попробуем узнать что-то новое. Вместе. Поехали!

Погодите. Знаете, прежде чем опубликовать эту книгу, я отправил ее десятку рецензентов, и почти все они возмутились из-за отсутствия введения. Они сказали, что я отправляю читателей на свидание вслепую с первой темой, не дав им необходимого контекста. Еще сказали, что мои идеи сложно воспринимать людям с богатым опытом программирования на C++/Java. Они находят, что их понимание ООП расходится с моим. Короче говоря, все потребовали, чтобы я написал введение. Собственно, вот оно.

Мне кажется, что ООП было разработано для решения проблем *процедурного* программирования, особенно на языках вроде C или COBOL. Процедурный стиль написания кода очень прост для понимания теми, кто знает, что процессор последовательно обрабатывает инструкции, манипулирующие данными в памяти. Фрагмент кода на C, также известный как функция, — это множество операторов, которые должны выполняться в хронологическом порядке, перемещая данные из одного места в памяти в другое и попутно проделывая над ними некоторые преобразования. Это работало много лет и работает до сих пор. Таким образом написана большая часть программного обеспечения, включая, к примеру, все основные Unix-подобные операционные системы.

Такой подход технически работает — код компилируется и запускается. Но при этом существует проблема с *сопровождением*. Автор кода более или менее понимает, как тот работает, пока пишет его. Но если заглянуть в него позже, то будет довольно трудно выяснить, что имел в виду его создатель. Иными словами, код написан для компьютеров, а не для людей. Лучший пример такого императивно-процедурного языка — ассемблер. Он ближе всего к процессору и очень далек от языка, на котором люди общаются в жизни. В ассемблере нет клиентов, файлов, прямоугольников и цен. Только регистры, байты, биты и указатели — то, что процессор понимает лучше всего.

Так было много лет назад, когда компьютеры были большими, медленными и повелевали всем. Мы вынуждены были говорить на их языке, а не наоборот. Так происходило преимущественно потому, что программное обеспечение должно было быть быстрым, чтобы стать полезным. Шла борьба за каждую инструкцию, за каждый байт памяти. Мы больше беспокоились о скорости и использовании памяти, чем о сопровождении кода. Важно отметить, что программисты тогда были намного дешевле компьютеров. Уж простите мое сравнение, но это правда. Нанять нового программиста было дешевле, чем купить новый жесткий диск. Иногда даже не представлялось возможным решить проблему добавлением вычислительных ресурсов. Более быстрого или объемного аппаратного обеспечения попросту не было. Программисты были довольно дешевы — поищите статистику 20-летней давности по их зарплатам. Именно поэтому приходилось делать то, что диктовали нам процессоры.

К счастью, некоторое время назад ситуация переменилась и проблема сопровождения стала более важна, чем скорость исполнения или расходование памяти. Жизненный цикл программных продуктов начал расти, и стало очевидно, что ассемблерный код не сможет пережить смену команды — новые люди предпочтут переписать код вместо того, чтобы разбираться, как работает подпрограмма из 5000 строк. Я считаю, что так и появились более высокоуровневые парадигмы программирования, такие как функциональная, логическая и объектно-ориентированная (есть и другие, но эти три, как мне кажется, наиболее популярны). Они перенесли фокус внимания с машин на людей. Они позволили нам говорить на своем языке, а не на том, к которому привык

процессор. Они помогли сделать код более читаемым и, как следствие, более простым для поддержки. Так было задумано.

Исторически ООП унаследовало многое от процедурного программирования. Под ООП здесь понимается не парадигма, а семейство популярных языков программирования, которые были названы объектно-ориентированными. Речь идет в основном о C++ и Java. Остальные, например Ruby, просто последовали их примеру. Возможно, поэтому C++ и стал так популярен — он выглядит как C, соответственно, его проще изучить. Язык Java также разрабатывался с целью упростить переход с C++ — его синтаксис очень похож на синтаксис C++ и прост для изучения программистами на C++. Из-за компромиссов в переходе от C к C++ и от C++ к Java ООП на сегодняшний день сильно напоминает процедурный C.

И пускай у нас есть классы и объекты — у нас все еще остались операторы, инструкции и их последовательное исполнение. Мы больше не работаем напрямую с указателями, памятью и регистрами процессора, но основной принцип остается неизменным — мы даем инструкции процессору и манипулируем данными в памяти. «Что с этим не так?» — можете спросить вы. Все в порядке, если вы хотите придерживаться процедурного подхода. Так же, как все было в порядке с ассемблером. Кроме того, что написанный на нем код было практически невозможно поддерживать. Точно такая же проблема сейчас и с программным обеспечением, написанным на Java/Ruby/Python, — его невозможно поддерживать, поскольку оно никогда не было объектно-ориентированным.

В нашем коде есть классы, методы, объекты, наследование и полиморфизм, но он не совсем объектно-ориентированный. Что именно с ним не так? Это я и попытаюсь объяснить в данной книге. Очень сложно уместить то, что я хочу сказать, в пару разделов. Чтобы понять идею и образ мышления, свойственные чистому ООП, вы должны прочитать всю книгу.

Я старался сделать материал максимально приближенным к практике и проиллюстрировать освещаемые идеи реалистичными примерами программного кода. Кроме того, в начале почти каждого раздела есть ссылка на статью в блоге, посвященную той же или очень близкой теме. Не стесняйтесь оставлять там свои комментарии, я постараюсь на них ответить.

Честно говоря, я не думаю, что прав во всем, о чем говорю. Я сам многие годы был процедурным программистом. Сложно оставить прошлый опыт позади и начать думать в терминах объектов, а не инструкций и операторов. Буду рад вашим отзывам. На этом введение закончено. В нем немного информации, но теперь вы по крайней мере знаете, чего ожидать от последующих страниц. Будьте готовы ко множеству противоречий. Наберитесь смелости бросить себе вызов. Приятного прочтения!

Благодарности

Большое спасибо тем, кто рецензировал эту книгу и помог сделать ее лучше и чище. Имена и фамилии этих людей упорядочены не по алфавиту, а по важности их вклада:

- Танасис Папапанагиоту (Thanasis Papapanagiotou);
- Франческо Бьянчи (Francesco Bianchi);
- Филипп Буук (Philip Buuck);
- Константин Комков (Konstantin Komkov);
- Андрей Истомин (Andrei Istomin).

Полный список помощников (в алфавитном порядке): Алексей Абашев (Alexey Abashev), Антон Архипов (Anton Arhipov), Фабрицио Баррос Кабрал (Fabrício Barros Cabral), Айон Бордиан (Ion Bordian), Тамила Бугаенко (Tamila Bugayenko), Филипп Буук (Philip Buuck), Франческо Бьянчи (Francesco Bianchi), Андрей Валяев (Andrey Valyaev), Илья Василевский (Ilya Vassilevsky), Виктор Гамов (Viktor Gamov), Артем Гапченко (Artem Gapchenko), Куин Гиль (Quinn Gil), Константин Гуков (Konstantin Gukov), Игорь Дмитриев (Igor Dmitriev), Анеш Догра (Aneesh Dogra), Андрей Истомин (Andrei Istomin), Кирилл Коротецкий (Kiryl Korotsetski), Никос Кекчидис (Nicos Kekchidis), Кристиан Кестлин (Christian Köstlin), Константин Комков (Konstantin Komkov), Николь Кордес (Nicole Cordes), Жанез Кухар (Janez Kuhar), Матеуш Ошлишлок (Mateusz Ośliślok), Сясонг Пан (Xiasong Pan), Танасис Папапанагиоту (Thanasis Papapanagiotou), Джон Пейдж (John Page), Ефим Пышнограев (Efim Pyshnograev), Силас Рейнагель (Silas Reinagel), Барух Садогурский (Baruch Sadogursky), Маркос Дуглас Б. Сантос (Marcos Douglas B. Santos), Оксана Семенкова (Oksana Semenkova), Маурицио Тоньери (Mauricio Togneri), Саймон Цай (Simon Tsai), Антон Черноусов (Anton Chernousov), Кшиштоф Шафраньски (Krzysztof Szafrański), Михал Швец (Michal Svec), Петр Шмелевский (Piotr Chmielowski).

Хотите увидеть себя в этом списке в следующем издании книги? Высылайте свои соображения на book@yegor256.com. Я отвечаю на все письма.

И конечно же, спасибо Андрии Миронюк (Andreea Mironiuc) за кактус на обложке.

1 Рождение

Начнем с того, что объект — это живой организм. С самой первой страницы мы приложим максимум усилий для его *антропоморфирования*. Иными словами, будем считать объект человеком. Поэтому я стану использовать в отношении объекта местоимение *«он»*. Мои дорогие читатели-женщины, пожалуйста, не обижайтесь. Я могу быть груб по отношению к бедному объекту, но не хочу быть грубым по отношению к женщинам. В этой книге объект будет мужского рода.

Он живет в своей *области видимости*, например (я в основном работаю с языком Java и буду так поступать далее в этой книге; надеюсь, что он вам понятен):

```
if (price < 100) {
  Cash extra = new Cash(5);
  price.add(extra);
}
```

Объект `extra` виден только внутри блока `if` — это его область видимости. Почему это важно именно сейчас? Потому что объект — живой организм. Прежде чем вдохнуть в него жизнь, мы должны определить его среду обитания. Что находится внутри него, а что находится снаружи? В данном примере `price` находится снаружи, а число `5` — внутри, верно?

К слову, прежде чем мы продолжим, хочу уверить вас, что все, что вы прочтете в этой книге, весьма практично и прагматично.

Бóльшая ее часть посвящена практическому приложению объектно-ориентированного программирования к реальным проблемам, а не философствованию. Главная цель, которую я преследую данной книгой, — улучшить *сопровождаемость* вашего кода. Нашего кода.

Сопровождаемость — важное качество любого программного обеспечения, оно может быть измерено как время, необходимое для того, чтобы понять ваш код. Чем больше времени требуется, тем ниже сопровождаемость и тем хуже код. Я бы даже сказал: *если я вас не понимаю, то виноваты в этом вы*. Понимая объекты и их роль в ООП, вы повысите сопровождаемость своего кода. Он станет короче, проще для восприятия, модульнее, целостнее и т. д. Он станет лучше, а в большинстве случаев и дешевле.

Пожалуйста, не удивляйтесь моим, казалось бы, излишне философским и абстрактным рассуждениям. Они на самом деле весьма практичны.

Теперь вернемся к области видимости. Если я — `extra`, то `price` — это моя окружающая среда. Число `5` внутри меня — это мой внутренний мир. Но это не совсем верно. Пока достаточно считать, что `price` находится снаружи, а `5` — внутри. Мы вернемся к этому чуть позже, в разделе 3.4.

1.1. Не используйте имена, заканчивающиеся на -er

Обсуждение на http://goo.gl/Uy3wZ6.

После того как вы определили область видимости будущего объекта, первостепенной задачей будет придумать ему хорошее имя.

Но отступим от основной линии повествования и обсудим разницу между объектом и классом. Я уверен, вы ее понимаете. Класс — это *фабрика* объектов. Уверяю вас, это важно.

Класс создает объекты, обычно говорят — *инстанцирует* их:

```
class Cash {
  public Cash(int dollars) {
    //...
  }
}
Cash five = new Cash(5);
```

Инстанцирование отличается от того, что мы называем паттерном Factory, но только потому, что оператор new в Java не настолько функционален, насколько мог бы быть. Его можно использовать лишь для создания экземпляра класса — объекта. Если мы попросим класс Cash создать новый объект, то и получим новый объект. При этом не проверяется, существуют ли похожие объекты, которые можно применять повторно, нельзя задать параметры, модифицирующие поведение оператора new, и т. д.

Оператор new — простейший механизм управления фабрикой объектов. В C++ также есть оператор delete, который позволяет удалить объект из фабрики. В Java и других «более продвинутых» языках мы, к сожалению, не имеем такой возможности. В C++ можно попросить фабрику создать объект, использовать его, затем указать той же фабрике его уничтожить:

```
class Cash {
public:
  public Cash(int dollars);
}
Cash five = new Cash(5); // создаем объект
cout << five;
delete five; // уничтожаем его
```

В Ruby идея класса как фабрики наиболее правильно выражается следующим образом:

```
class Cash
  def initialize(dollars)
    # ...
  end
end
Cash five = Cash.new(5)
```


new — статический метод класса Cash, когда он вызывается, класс получает управление и создает объект five. Этот объект инкапсулирует число 5 и ведет себя как целое число.

Следовательно, хорошо известный паттерн «Фабрика» является более функциональной альтернативой оператору new, но идея у них одна. Класс — это фабрика объектов. Он создает объекты, следит за ними, при необходимости уничтожает и т. д. Большая часть этих возможностей в большинстве языков реализована средствами среды исполнения, а не кодом класса, но это не имеет особого значения. На поверхности мы видим класс, который дает нам объекты по запросу. У вас может возникнуть вопрос относительно классов-утилит, не имеющих объектов. Мы поговорим о них позже, в разделе 3.2.

Паттерн проектирования «Фабрика» в Java работает как расширение оператора new. Он делает оператор более гибким и функциональным, присоединяя к нему дополнительную логику, например:

```
class Shapes {
  public Shape make(String name) {
    if (name.equals("круг")) {
      return new Circle();
    }
    if (name.equals("прямоугольник")) {
      return new Rectangle();
    }
    throw new IllegalArgumentException("фигура не найдена");
  }
}
```

Это типовой пример фабрики в Java. Она позволяет инстанцировать объекты, используя текстовые наименования их типов. Но в результате все равно применяется оператор new. Этим я хочу сказать, что разница между шаблоном «Фабрика» и оператором new невелика. В идеальном ООП-языке его функциональность была бы доступна в операторе new. Я хочу, чтобы вы представляли себе класс как склад объектов, которые можно

брать оттуда при необходимости и возвращать, когда потребность в них исчезает.

Иногда, чтобы объяснить, что такое класс, используют понятие «шаблон объекта». Это совершенно неверно, поскольку такое определение делает класс пассивным безмозглым набором кода, который куда-то копируется при необходимости. Даже если, с вашей точки зрения, технически это выглядит именно так, старайтесь так не *думать*. Класс — это *фабрика* объектов, и точка. Кстати, я не пытаюсь рекламировать паттерн «Фабрика». На самом деле я не очень большой его приверженец, хотя его идея технически верна. Хочу сказать, что мы должны представлять себе класс активным менеджером объектов. Также можем назвать его хранилищем или складом — местом, откуда мы берем объекты и куда их возвращаем.

К слову, учитывая, что объект — это живое существо, его класс — это его мать. Такая метафора будет наиболее точна.

А теперь вернемся к основной теме данного раздела — проблеме выбора хорошего имени класса. По сути, существует два подхода — правильный и неправильный. Неправильный — это когда мы смотрим, что класс *делает*, и даем ему имя согласно функциональности. Приведу пример класса, названного в соответствии с таким подходом:

```
class CashFormatter {
  private int dollars;
  CashFormatter(int dlr) {
    this.dollars = dlr;
  }
  public String format() {
    return String.format("$ %d", this.dollars);
  }
}
```

Если у меня есть нечто под названием `CashFormatter`, то что оно делает? Оно форматирует сумму в долларах в виде текстовой

строки. И должно называться `Formatter`, так ведь? Разве это не очевидно?

Вы, вероятно, заметили, что я не назвал объект `CashFormatter` «он». Я так поступил, потому что не могу заставить себя уважать такой объект. Я не могу его антропоморфировать и обращаться с ним как с уважаемым гражданином моего кода.

Такой принцип именования совершенно неверен, но весьма широко распространен. Призываю вас не придерживаться такого образа мышления. Имя класса не должно происходить от названия функциональности, предоставляемой его объектами! Напротив, класс должен быть назван на основе того, чем он *является*, а не того, что он *делает*. `CashFormatter` необходимо переименовать в `Cash`, или `USDCash`, или `CashInUSD` и т. п. Метод `format()` нужно назвать `usd()`, например:

```
class Cash {
  private int dollars;
  Cash(int dlr) {
    this.dollars = dlr;
  }
  public String usd() {
    return String.format("$ %d", this.dollars);
  }
}
```

Иными словами, объекты должны характеризоваться своими способностями. То, что я есть, выражается в том, что я могу, а не в моих параметрах вроде роста, веса или цвета кожи.

«Вредный» ингредиент здесь — суффикс -er.

Существует масса примеров классов, названных подобным образом, и у всех них есть суффикс -er, например: `Manager`, `Controller`, `Helper`, `Handler`, `Writer`, `Reader`, `Converter`, `Validator` (-or также вреден), `Router`, `Dispatcher`, `Observer`, `Listener`, `Sorter`, `Encoder` и `Decoder`. Все эти имена плохи. Уверен, немало примеров этого вы и сами видели. Вот несколько контрпримеров:

`Target`, `EncodedText`, `DecodedData`, `Content`, `SortedLines`, `ValidPage`, `Source` и т. п.

Но у этого правила есть исключения. Некоторые англоязычные существительные имеют суффикс -er, который в свое время (правда, оно давно прошло) указывал, что эти слова обозначают исполнителей каких-то действий, например computer или user. Мы больше не называем user что-то, что буквально пользуется (use) чем-то. Это скорее персона, взаимодействующая с системой. Мы воспринимаем computer не как что-то, что вычисляет (computes), а как устройство, которое является, как бы сказать, компьютером. Но таких исключений не так уж много.

Объект не переходник между внешним миром и своим внутренним состоянием. Объект не набор процедур, вызываемых для манипуляции инкапсулированными в нем данными. Ни в коем случае! Напротив, объект — это представитель инкапсулированных в нем данных. Чувствуете разницу?

Переходник не заслуживает уважения, поскольку он просто передает через себя информацию, не будучи достаточно сильным или умным, чтобы модифицировать ее или делать что-то самостоятельно. Напротив, *представитель* — самодостаточная сущность, способная принимать собственные решения и действовать самостоятельно. Объекты должны быть представителями, а не переходниками.

Имя класса, которое заканчивается на -er, говорит нам о том, что это создание является не объектом, а лишь набором процедур, которые могут манипулировать некоторыми данными. Это процедурный стиль мышления, унаследованный многими объектно-ориентированными разработчиками из C, COBOL, BASIC и других языков. Сейчас мы используем Java и Ruby, но все еще думаем в терминах данных и процедур.

И все-таки как правильно называть классы?

Все просто: посмотрите, что инкапсулируют объекты этого класса, и придумайте для этого название. Пусть у нас есть список чисел и алгоритм, который определяет, какие из них являются простыми. Если вам нужно вывести только простые числа из упомянутого списка, не называйте класс `Primer`, или `PrimeFinder`, или `PrimeChooser`, или `PrimeHelper`. Лучше назовите его `PrimeNumbers` (для разнообразия приведем код на Ruby):

```
class PrimeNumbers
  def initialize(origin)
    @origin = origin
  end
  def each
    @origin
      .select { |i| prime? i }
      .each { |i| yield i }
  end
  def prime?(x)
    # ...
  end
end
```

Понимаете, о чем я? Класс `PrimeNumbers` ведет себя как список чисел, но возвращает только те из них, которые являются простыми. Подобную функциональность можно реализовать на C в чисто процедурном стиле следующим образом:

```
void find_prime_numbers(int* origin,
  int* primes, int size) {
  for (int i = 0; i < size; ++i) {
    primes[i] = (int) is_prime(origin[i]);
  }
}
```

Здесь мы приводим процедуру `find_prime_numbers`, которая принимает два массива целых чисел, последовательно обходит первый массив в поисках простых чисел и помечает соответствующие позиции во втором массиве. Никаких объектов тут

нет. Это чисто *процедурный* подход, и он неверен. Он работает в процедурных языках, но мы находимся в мире ООП.

Эта процедура — переходник между двумя наборами данных: исходным списком чисел и списком простых чисел. Объект — это нечто иное. Объект не переходник, а представитель других объектов и их сочетаний. В приведенном ранее примере мы создаем объект класса `PrimeNumbers`, который ведет себя как набор чисел, но видны в нем только простые числа.

Если ваш объект на самом деле является процедурой `find_prime_numbers`, то у вас проблема. Объект не должен работать как набор процедур, хотя и может выглядеть очень похоже. Несмотря на то что класс `PrimeNumbers` инкапсулирует список чисел, он не позволяет управлять этим списком или искать в нем что-либо. Вместо этого он заявляет: «Я теперь список!» Если я хочу что-то сделать со списком, то прошу объект сделать это, а объект уже решает, как реагировать на мою просьбу. Если он захочет, то возьмет данные из исходного списка. Если нет — его право.

`PrimeNumbers` *является* списком чисел, а не набором методов его обработки. Он — *список*!

Обобщим этот раздел. Когда приходит время давать имя новому классу, думайте о том, что он *есть*, а не о том, что он *делает*. Он — список, и он может выбирать элементы из списка по индексу. Он — SQL-запись, и он может извлечь отдельную ячейку как целое число. Он — пиксел, и он может изменить свой цвет. Он — файл, и он может читать содержимое с диска. Он — алгоритм кодирования, и он может кодировать. Он — HTML-документ, и он может быть отображен.

То, что я делаю, и то, кто я есть, — две разные вещи.

Кроме того, имена, заканчивающиеся на `Util` или `Utils`, — еще один пример плохого именования класса. Это так называемые классы-утилиты, мы поговорим о них в разделе 3.2.

Andriy спросил 15 февраля 2017 года:

Как быть с `ILogger`?

Егор Бугаенко:

Переименуйте его в `ILog`.

Andriy:

Но log — это сообщение, а `logger` имеет дело с сообщениями. Возможно, подойдет имя `LoggingTool`?

Егор Бугаенко:

Согласно словарю Merriam-Webster log определяется как журнал достижений, событий, повседневной деятельности; данными, добавляемыми в log, могут быть заметки или события. Журнал (журнальная книга) — лист бумаги, на котором записывают заметки.

Mikhail Gromov спросил 18 декабря 2016 года:

Допустим, у меня есть какой-то метод класса, который в одной из строк делает вызов `appleSorter.sort(apples)`. Если `appleSorter` — коллаборатор, он передается как параметр конструктора и я могу передать фиктивный экземпляр этого сортировщика и протестировать метод. Но что мне делать, если нужно вызвать `new Sorter(apples)`?

Егор Бугаенко:

Не создавайте фиктивный `sorter`. Просто не оборачивайте `apples` в `Sorter`.

---

losaciertos спросил 3 ноября 2016 года:

Как насчет паттерна `Observer`? Как его реализовать в «настоящем» ООП-мире?

Егор Бугаенко:

Мне нравится имя `Target`.

losaciertos:

То есть вы хотите сказать, что `Listener` и `Observer` — нормальные классы, если их назвать `EventTarget` и `EventSource`?

Егор Бугаенко:

Именно.

Fabricio Cabral спросил 31 мая 2016 года:

Что вы думаете насчет классов или интерфейсов с суффиксом ABLE, например `Serializable`, `Cloneable`, `Cacheable`?

Егор Бугаенко:

Мне кажется, что они не лучше, чем те, что с суффиксом -er. Имя `Printable` означает, что меня можно напечатать, но ничего не говорит о том, кто я такой. Это неправильно. Я понимаю, что они удобны чисто технически, но не рекомендовал бы их использовать.

Juliano Boesel Mohr спросил 17 мая 2016 года:

Как насчет паттерна `Builder`? Вы бы рекомендовали его применять?

Егор Бугаенко:

Я считаю, что паттерн `Builder` плох, поскольку он поощряет создание крупных объектов. Идеальный объект не должен инкапсулировать более 1–4 свойств. Определенно не больше пяти. `Builder` создан, чтобы помочь нам строить более крупные объекты. Следовательно, его использование — очень плохая идея.

---

pixdigit написал 10 июня 2015 года:

А если я хочу создать класс, который и правда *является* набором функций? (Кажется, я знаю ответ, просто хочу проверить.)

Егор Бугаенко:

Тогда вам нужно менять образ мышления. Вы хотите создать не класс, а библиотеку процедур. Вернитесь к процедурному программированию на C или COBOL, там такие вещи поощряются.

Riccardo Cardin написал 11 марта 2015 года:

Мне кажется, нам нужно сформулировать некоторые соображения относительно объектов. Как подсказывает мой опыт разработчика, мы можем разделить объекты на два типа:

1) объекты, моделирующие действительность, а также домены и операции, определенные на них;
2) объекты, взаимодействующие с первыми для построения архитектуры приложения, удовлетворяющей потребностям пользователей.

Объекты, принадлежащие первому множеству, могут не следовать правилу, объясняемому в статье. Объекты наподобие контроллеров, служб, декораторов и фабрик принадлежат второму множеству. Они не имеют отношения к *реальному миру*, а помогают первым взаимодействовать друг с другом. Мне кажется, что для второго множества объектов ваше правило «без -er» слишком строгое и не будет соблюдаться.

Егор Бугаенко:

Между «внешними» и «внутренними» объектами не должно быть разницы. Каждый объект имеет собственную область

видимости, по отношению к которой все остальные объекты являются внешними. Связаны ли эти объекты с реальностью в нашем понимании, не имеет значения. Нам это знать совершенно не обязательно. Все, что мы знаем об объекте, — это поведение, которое он демонстрирует посредством своих методов.

1.2. Сделайте один конструктор главным

Обсуждение на http://goo.gl/brqhYS.

Конструктор — точка входа нового объекта. Он принимает несколько аргументов и что-то делает с ними, чтобы подготовить объект к выполнению своих обязанностей:

```
class Cash {
  private int dollars;
  Cash(int dlr) {
    this.dollars = dlr;
  }
}
```

В данном примере есть только один конструктор, и единственное, что он делает, — инкапсулирует сумму в долларах в приватное целочисленное свойство `dollars`. Если вы правильно проектируете свои классы (в соответствии с рекомендациями из последующих разделов), то у них будет много конструкторов и немного методов. Вы все правильно поняли: конструкторов в классах должно быть больше, чем методов. Я знаю, что не все языки поддерживают множественные конструкторы из-за отсутствия возможности перегрузки методов. Мы обсудим это ограничение через минуту.

Итак, 2–3 метода и 5–10 конструкторов. Так, по моему мнению, должен выглядеть идеальный класс. Эти цифры, конечно же, взяты из головы и не имеют строгого обоснования. Мы обсудим

количество публичных методов в разделе 3.1. Этим я хочу сказать, что связный и гибкий класс имеет небольшое количество методов и сравнительно большое количество конструкторов.

Чем больше в вашем классе конструкторов, тем лучше, тем удобнее классы для меня — их пользователя. Я хочу иметь возможность создать экземпляр класса `Cash` многими способами, например:

```
new Cash(30);
new Cash("$29.95");
new Cash(29.95d);
new Cash(29.95f);
new Cash(29.95, "USD");
```

Все эти операторы должны создавать одинаковые в смысле поведения объекты. Чем больше конструкторов, тем большую гибкость применения ваших классов вы обеспечиваете мне, своему клиенту. И наоборот, чем больше методов предоставляет ваш класс, тем сложнее мне его использовать. Большое количество методов приводит к размыванию фокуса и нарушению принципа единственности ответственности, который мы обсудим в разделе 3.1. Большее количество конструкторов означает большую гибкость.

Пользуясь классом `Cash`, я получаю дополнительную гибкость, поскольку мне не нужно выполнять преобразование классов или разбор строк, если у меня есть число в текстовом формате. Класс `Cash` делает эту работу за меня. У меня есть строка, для нее предусмотрен конструктор. У меня есть число с плавающей точкой, конструктор предусмотрен и для него. Благодаря такой гибкости я пишу меньше кода и реже создаю повторяющиеся фрагменты кода. Напротив, иметь большое количество открытых методов — плохо, поскольку это снижает гибкость.

Основная задача конструктора — инициализировать инкапсулированные свойства, используя переданные ему аргументы.

Я рекомендую поместить инициализацию свойств лишь в один из конструкторов и сделать его основным. Остальные, так называемые вторичные конструкторы пусть вызывают основной, например:

```
class Cash {
  private int dollars;
  Cash(float dlr) {
    this((int) dlr);
  }
  Cash(String dlr) {
    this(Cash.parse(dlr));
  }
  Cash(int dlr) {
    this.dollars = dlr;
  }
}
```

Я всегда стараюсь поместить основной конструктор последним в коде, после всех вторичных, как показано в примере. Главным образом из соображений лучшей сопровождаемости. Когда я открываю код класса с десятью конструкторами, созданный полгода назад, я не собираюсь читать его весь в поисках основного из них. Я просто прокручиваю код до последнего конструктора, который всегда будет основным.

В приведенном фрагменте один основной конструктор и два вторичных. Основной конструктор инициализирует свойство `this.dollars` переданным ему целочисленным аргументом. Вторичные конструкторы готовят целочисленный аргумент для основного, либо разбирая строку, либо преобразуя его из других форматов. В одном из конструкторов я ссылаюсь на приватный статический метод `Cash.parse()`, который разбирает строку и преобразует ее в число. Так было сделано потому, что Java не позволяет ничего делать перед вызовом `this()`. В C++ такие ухищрения не нужны.

Каков смысл принципа «один основной, много вторичных»? Он в основном позволяет избежать дублирования кода, сделать его чище, а значит, улучшить сопровождаемость. Вот как выглядел бы класс, написанный без учета данного принципа:

```
class Cash {
  private int dollars;
  Cash(float dlr) {                    // плохо!
    this.dollars = (int) dlr;
  }
  Cash(String dlr) {                   // плохо!
    this.dollars = Cash.parse(dlr);
  }
  Cash(int dlr) {
    this.dollars = dlr;
  }
}
```

Допустим, мы хотим убедиться, что сумма долларов всегда положительная. Нам придется поместить код проверки в трех разных местах, в трех конструкторах. В первом примере за счет использования одного основного и двух вторичных конструкторов код проверки нужно будет добавить только в одном месте.

К сожалению, не все объектно-ориентированные языки поддерживают *перегрузку методов* — механизм объявления методов или конструкторы с одинаковыми именами, но разными наборами аргументов. Например, Ruby и PHP не поддерживают перегрузку методов. И они почему-то называются объектно-ориентированными. И я не шучу. Перегрузка методов — фундаментальная и очень важная часть ООП. Она существенно улучшает читаемость кода, семантически приближая его к языку задачи. К примеру, код был бы намного чище, если бы в нем были методы `content(File)` и `content(File,Charset)`, а не `content(File)` и `contentInCharset(File,Charset)`.

Тем не менее даже в этих языках необходимо делать конструкторы гибкими и многоцелевыми. Первым делом вам следует задуматься о том, чтобы прекратить пользоваться ими и перейти на Java, C++ или другой подобный язык, который имел бы достаточно возможностей, чтобы называться ООП-языком. Если это невозможно (например, вы работаете с JavaScript и у вас нет лучшей альтернативы), используйте ассоциативные массивы (map, dictionary) аргументов. Пример для PHP 5.4:

```
class Cash {
  private $_dollars;
  public function __construct($args) {
    if (is_int($args)) {
      $this->_dollars = $args;
    } else if (array_key_exists('float', $args)) {
      $this->__construct(intval($args['float']));
    } else if (array_key_exists('iso', $args)) {
      $this->__construct(
        parse_dollars($args['iso'])
      );
    } else {
      throw new Exception('can\'t initialize');
    }
  }
}
new Cash(30);
new Cash(['float' => 29.95]);
new Cash(['iso' => 'USD 29.95']);
```

Такой код намного более многословен и намного менее читабелен, нежели код на Java. Но, как видите, в нем используется тот же принцип — инициализировать поля класса необходимо только в одном месте. Во всех других местах следует просто подготавливать аргументы и отправлять их в это место. Вызов метода `__construct` — плохой тон в PHP, но в данном случае такое приемлемо, поскольку у нас нет иного выбора.

В языках, не поддерживающих перегрузку методов, вы, вероятно, можете прибегать и к другим приемам, но основной

принцип остается тем же — инициализация внутренних свойств происходит только в одном месте. Во всех остальных местах аргументы подготавливаются, форматируются, разбираются, преобразуются и т. п.

Как и в случае с другими рекомендациями, приводимыми в данной книге, основная цель — сопровождаемость. Этот принцип позволит вам снизить сложность кода и избежать дублирования — двух злейших врагов сопровождаемости.

Jean-Paul Wenger спросил 25 сентября 2017 года:

Если совместить правило «единственный основной конструктор», описанное ранее, с правилами «в конструкторе не должно быть кода» и «не используйте оператор new за пределами вторичных конструкторов», тогда что же делает единственный основной конструктор? Он просто присваивает членам класса значения аргументов. Тогда из этих трех правил следует новое правило: «Основные конструкторы не делают ничего, кроме присвоения членам класса значений аргументов».

Егор Бугаенко:

Да, именно так.

Kata написал 25 сентября 2017 года:

Как разработчики мы, безусловно, часто сталкиваемся с такой ситуацией. Ваш подход называется «телескопические конструкторы». Он достаточно хорош для небольшого количества параметров. Если список параметров большой (пять и более) и не может быть «телескопирован», неплохо подойдет паттерн «Строитель» в Java, но он увеличивает размер кода и количество классов. Хотелось бы узнать ваши соображения по поводу того, как передавать длинные списки параметров в конструкторы.

И есть ли какой-то общий способ избежать большого количества параметров в конструкторах?

Егор Бугаенко:

Если у вас слишком много параметров в конструкторе, с вашим классом что-то не так. Он попросту слишком большой. Разбейте его на части. Паттерн «Строитель» не решает проблему, а просто маскирует ее. Я бы не рекомендовал его использовать. Любой класс с более чем пятью параметрами в конструкторе спроектирован плохо. Без исключения.

1.3. В конструкторах не должно быть кода

Обсуждение на http://goo.gl/DCMFDY.

У нас есть класс с основным конструктором, который принимает все необходимые аргументы. Этих аргументов достаточно для того, чтобы инициализировать состояние нового объекта. Очевидно, так ведь? Поскольку этот конструктор — единственная точка входа в процесс инициализации объекта, предоставляемый набор аргументов полон — ничто не упущено, ничего лишнего. Вопрос в том, что мы можем или не можем делать с этим набором аргументов. Как ими манипулировать?

Эмпирическое правило выглядит следующим образом: «Не трогайте аргументы». Сначала рассмотрим противоположный пример. Данный код «трогает» свой единственный аргумент во время инициализации:

```
class Cash {
  private int dollars;
  Cash(String dlr) {
    this.dollars = Integer.parseInt(dlr);
  }
}
```


Я хочу инкапсулировать целое число, притом что аргументом конструктора является текстовая строка. Мне нужно перевести строку в число, и я выполняю это преобразование прямо внутри конструктора. Все кажется простым и очевидным, не так ли? Возможно, но это *очень плохой* подход.

Инициализация объекта не должна содержать код и затрагивать аргументы. Вместо этого она должна при необходимости оборачивать их или инкапсулировать в необработанном виде. Вот пример того же кода, который не трогает текст:

```
class Cash {
  private Number dollars;
  Cash(String dlr) {
    this.dollars = new StringAsInteger(dlr);
  }
}
class StringAsInteger implements Number {
  private String source;
  StringAsInteger(String src) {
    this.source = src;
  }
  int intValue() {
    return Integer.parseInt(this.source);
  }
}
```

Чувствуете разницу? В первом примере преобразование из строки в число происходит непосредственно в момент инициализации объекта. Во втором оно откладывается до момента использования объекта класса `Cash`.

Разумеется, в соответствии с принципом, рассмотренным в предыдущем разделе, класс `Cash` должен иметь два конструктора — один основной и один вторичный:

```
class Cash {
  private Number dollars;
  Cash(String dlr) {                    // вторичный
    this(new StringAsInteger(dlr));
```

```
  }
  Cash(Number dlr) {                    // основной
    this.dollars = dlr;
  }
}
```

Внешне создание экземпляра класса `Cash` выглядит одинаково в обоих случаях:

```
Cash five = new Cash("5");
```

Однако в первом примере объект `five` инкапсулирует число 5, а во втором — экземпляр класса `StringAsInteger`, который *похож* на `Number`. Класс `StringAsInteger` я придумал сам. Его не существует в Java. Как я уже сказал, язык Java не является чисто объектно-ориентированным, поэтому иногда мне приходится прибегать к некоторым ухищрениям. Считайте эти примеры псевдокодом. Но это отнюдь не значит, что мои рекомендации абстрактны и не годятся для практического применения. Просто не все они подходят для того программного обеспечения, которое вы сейчас пишете. Наша первоочередная цель в рамках данной книги — изменить свое мировоззрение и понимание ООП, вторая — привести практические примеры и применить новое мировоззрение к написанию программ. К сожалению, вторая цель не всегда легко достижима.

При истинном объектно-ориентированном подходе инстанцирование объекта подразумевает *компоновку* меньших объектов в один более крупный. Единственной причиной необходимости данного процесса является потребность в новой сущности, которая подчиняется новому контракту.

Взгляните на пример с объектом `five` типа `Cash`. Что не так со строковым объектом `"5"`? Зачем нужно было создавать экземпляр класса `Cash`? Почему нельзя работать с объектом `"5"`? Потому что он не предоставляет необходимых нам методов. Не работает согласно нужному контракту. Поэтому пришлось

создать новый объект другого типа — `five` класса `Cash`. Он больше не работает по контракту класса `String`, он работает по другому контракту. Например, предоставляет метод `cents()`.

Мы создали его, но еще не попросили работать на себя!

Первый шаг — инстанцировать объект, второй шаг — позволить ему работать на нас. Они не должны перекрываться. Конструктор не должен просить свои аргументы что-либо делать, так как его самого еще не просили ничего делать. Иными словами, в конструкторе не должно быть кода — только операторы присваивания.

В случае с C++ тело конструктора должно быть пустым, например:

```
class Cash {
public:
  Cash(const string& txt):
    dollars(new StringAsInteger(txt)) {
    // тело конструктора всегда пустое
  }
private:
  int dollars;
}
```

Существует несколько чисто технических причин для такой рекомендации. Во-первых, производительность конструктора без кода легче оптимизировать, а значит, такие конструкторы ускоряют ваш код. Вот пример, который на первый взгляд кажется медленным, но на самом деле оказывается быстрым:

```
class StringAsInteger implements Number {
  private String text;
  public StringAsInteger(String txt) {
    this.text = txt;
  }
  public int intValue() {
    return Integer.parseInt(this.text);
  }
}
```

Похоже, преобразование строки в число будет выполняться при каждом вызове `intValue()`, верно? И это действительно так. Такой код будет выполнять разбор дважды:

```
Number num = new StringAsInteger("123");
num.intValue();                          // первый разбор
num.intValue();                          // второй разбор
```

Как тогда, спросите вы, он может быть более быстрым, чем такой код:

```
class StringAsInteger implements Number {
  private int num;
  public StringAsInteger(String txt) {
    this.num = Integer.parseInt(txt);
  }
  public int intValue() {
    return this.num;
  }
}
```

Такой код действительно более эффективен, поскольку он выполняет разбор лишь однажды — во время инициализации объекта. При каждом последующем вызове `intValue()` объект просто возвращает инкапсулированное число. И в чем тогда смысл?

А вот в чем. Второй пример, где разбор строки происходит в конструкторе, оптимизировать не получится. Разбор будет выполняться всякий раз при создании объекта. Мы не можем этим управлять. Даже если в отдельных случаях не надо вызывать `intValue()`, процессор будет тратить время на разбор строки. Рассмотрим следующий пример:

```
Number five = new StringAsInteger("5");
if (/* что-то не так */) {
  throw new Exception("какая-то проблема");
}
five.intValue();
```

Мы сначала разобрали объект `"5"`, а потом увидели, что он нам не нужен! И нет способа предотвратить такой случай. Каждый

раз, когда мы создаем объект, он немедленно обрабатывает аргументы, которые мы ему передаем. Это происходит без нашего ведома, причем всегда. Напротив, если мы инкапсулируем аргументы в том виде, в котором они были переданы, и обрабатываем их позже, *по требованию*, то даем пользователям свободу выбора момента, когда это должно произойти.

Когда пользователь хочет предотвратить повторный разбор, он всегда может создать *декоратор*, который закэширует результат разбора после первого вызова:

```
class CachedNumber implements Number {
  private Number origin;
  private Collection<Integer> cached =
    ArrayList<>(1);
  public CachedNumber(Number num) {
    this.origin = num;
  }
  public int intValue() {
    if (this.cached.isEmpty()) {
      this.cached.add(this.origin.intValue());
    }
    return this.cached.get(0);
  }
}
```

Я использую `ArrayList`, чтобы избежать `null` — злейшего врага ООП. Мы обсудим это позже, в разделах 3.3 и 4.1.

Эта реализация кэширования весьма примитивна, но, надеюсь, идею вы поняли. Затем для повышения эффективности объекта применяется обертывание в кэширующий декоратор:

```
Number num = new CachedNumber(
  new StringAsInteger("123")
);
num.intValue();                  // первый разбор
num.intValue();                  // здесь разбора не происходит
```

Красота решения в том, что оно хорошо управляемое и прозрачное. Инстанцирование объекта не делает ничего, кроме его *сборки*, —

реальная работа выполняется методами объекта. В то же время можно контролировать все! Мы оптимизируем работу объекта.

Убирая исполняемый код из конструкторов, мы делаем объекты более управляемыми и прозрачными для конечных пользователей. Мы помогаем лучше понять их и упрощаем повторное использование. Они работают только тогда, когда их просят об этом, а до этого момента не делают ничего.

Они очень ленивы — в хорошем смысле.

Могут возникать ситуации, когда совершенно очевидно, что все манипуляции должны происходить ровно один раз. Почему бы в таких ситуациях не поместить их в конструктор? Подобным образом поступать можно, но я бы не рекомендовал так делать в первую очередь из соображений однородности. Вы не знаете, что случится с классом в будущем и насколько он изменится после очередного рефакторинга. Помещая все манипуляции в конструктор, мы существенно усложняем рефакторинг. Тот, кто им займется, вынужден будет вынести все манипуляции в методы класса. Ведь только тогда он сможет внести реальные изменения.

Я попытался найти вторую техническую причину рекомендации оставлять конструктор пустым, но у меня не вышло. Похоже, что та причина, которую я раскрыл ранее, — единственная. Облегченные конструкторы упрощают создание объектов, делая их более настраиваемыми и прозрачными. Вот и все.

Кроме того, если вы заглянете в код качественно спроектированного объектно-ориентированного ПО, то наверняка увидите там что-то подобное следующему:

```
App app = new App(new Data(), new Screen());
app.run();
```

Это очень абстрактный пример, но, надеюсь, вы меня понимаете. Сначала мы собираем приложение, затем передаем ему управление. Пока строим приложение, оно ничего не делает: не подключается к базам данных, не открывает порты, не обрабатывает информацию. Оно просто создает все внутренние объекты и подготавливает их к работе. Затем мы вызываем `run()`, что позволяет объектам делать свое дело в нужное время и в нужном месте.

При разработке всех ваших объектов, от `App` на верхнем уровне до самого низкоуровневого `StringAsInteger`, необходимо держать в голове мысль о том, что их конструкторы не должны содержать кода.

Fernando спросил 7 сентября 2017 года:

> А если объект конструирует свое внутреннее состояние на основе JSON-ответа на API-запрос к веб-службе, разве он не должен бросать исключение?

Егор Бугаенко:

> Должен, но только тогда, когда с ним начинают работать, а не тогда, когда его собирают.

Riccardo Cardin написал 9 мая 2015 года:

> Мне кажется, в некоторых обстоятельствах ваш подход «ленивой инициализации» может дать некоторые преимущества. Возьмем, к примеру, бесконечные потоки или передачу по ссылке в Scala. Кроме того, он хорош также в случаях, когда объект весьма тяжел. Возьмем, к примеру, паттерн «Виртуальный прокси» из книги «банды четырех»[1]. Но в большинстве случаев не стоит откладывать исполнение кода, создающего объект. Делая так, вы нарушаете принцип скорейшего отказа — один из основных принципов в программировании.

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2015.

Егор Бугаенко:

Я понимаю вашу позицию, но позволю себе не согласиться с ней. Мне кажется, что отказ должен происходить не во время создания объекта, а только во время его использования. Иными словами, объекты имеют право на отказ только тогда, когда их просят реализовать некоторое поведение.

Fabricio Cabral спросил 9 мая 2015 года:

То есть, по-вашему, конструктор никогда не должен бросать исключения? А как же базовый принцип ООП, который гласит: «Нельзя позволять создание объектов с некорректным состоянием»? К примеру, если программист написал `new EnglishName("")` (имя не может быть пустой строкой), то конструктор `EnglishName` не должен бросать исключение? Как бы вы поступили в этом случае?

Егор Бугаенко:

Да, мне кажется, что в конструкторе не должна происходить проверка введенных данных. Даже если вы передали `NULL` в качестве единственного аргумента конструктора `new EnglishName()`, он не должен «возмущаться». Если вы впоследствии вызовете у него метод `first()`, то не узнаете, что созданный объект оказался неполон. Он был хорошим работником для ничегонеделания, но оказался некомпетентен в задаче извлечения имени. Видите, к чему я клоню? Если вы не дадите мне работу, то никогда не узнаете, хороший ли я работник. Здесь то же самое.

2 Образование

Разделы данной книги сгруппированы в главы весьма искусственно, но в этом есть некоторая логика. В данной главе мы обсудим несколько принципов подготовки объекта к взаимодействию с другими объектами. Мы отправим его в школу и преподадим ему несколько уроков этикета.

Вкратце сформулируем советы, которые мы рассмотрим в последующих разделах. Объект должен быть *небольшим*. Маленький объект — это элегантный и хорошо сопровождаемый объект. В ООП не может быть никакого оправдания классу в 1000 строк кода. К сожалению, проще о маленьких объектах говорить, чем их создавать. Как можно уменьшить объект, если в проекте столько функциональных требований? Потерпите немного. Далее приведу несколько практических рекомендаций.

2.1. Инкапсулируйте как можно меньше

Помните: все делается ради улучшения сопровождаемости. Все, о чем я пишу в этой книге, напрямую влияет на сложность кода, которая напрямую влияет на его *сопровождаемость*. Чем выше сложность, тем хуже сопровождаемость, тем больше потери

денег и времени и тем меньше удовлетворенных потребителей[1]. Уверен, тут вы со мной солидарны.

Поэтому я рекомендую инкапсулировать не более *четырех* объектов. Если вам нужно инкапсулировать больше объектов, значит, с вашим классом что-то не так и он нуждается в рефакторинге. Без исключения. Не больше четырех. Я взял это число из головы, у меня нет никаких научных доказательств. Позже объясню, почему выбрал именно его.

Набор инкапсулированных объектов называется *состоянием* или *идентичностью* объекта. Например:

```
class Cash {
  private Integer digits;
  private Integer cents;
  private String currency;
}
```

Здесь мы инкапсулируем три объекта. Все вместе они идентифицируют объекты класса `Cash`, то есть любые два объекта, инкапсулирующие одни и те же значения долларов, центов и название валюты, равны друг другу. Да, в Java это чисто технически неверно, но я считаю это недостатком языка. Вот как, по-моему, должна быть реализована объектная парадигма в чистом объектно-ориентированном языке:

```
Cash x = new Cash(29, 95, "USD");
Cash y = new Cash(29, 95, "USD");
assert x.equals(y);
assert x == y;
```

В Java, как и в C++, идентичность объекта отделена от его состояния. Два объекта, `x` и `y`, имеют одинаковое состояние, но разные

[1] У меня нет статистических данных, которые подкрепили бы это утверждение, но мне оно кажется очень логичным. Если вам известны исследования на эту тему, пожалуйста, сообщите мне о них, и я включу их в следующее издание этой книги. — *Примеч. авт.*

идентичности. С точки зрения оператора `==` они не равны друг другу, реализация метода `equals()` по умолчанию также считает, что они не совпадают.

Это недостаток языка Java, унаследованный им от C++. Насколько я понимаю, объект в ООП — это агрегат из других объектов, работающих совместно для получения более высокоуровневого поведения. Книга — агрегат из страниц, обложки и ISBN, а книжная полка — агрегат из книг и названия. Машина — агрегат из колес, двигателя и лобового стекла, а гараж — агрегат из машин и адреса. Работник — агрегат из имени, возраста и зарплаты, а отдел — агрегат из сотрудников, названия и начальника. Эти примеры весьма примитивны, но они показывают, что объект не существует, да и не может существовать без инкапсулированных объектов — он ничто без своих частей.

В Java, однако же, объект может существовать без составных частей и при этом не быть равным своей точной копии, у которой тоже нет составных частей. Это противоречит здравому смыслу. Но в Java это имеет смысл. В Java и почти во всех остальных ООП-языках объект — это всего лишь набор данных с прикрепленными к нему методами. Что-то вроде оболочки, где можно хранить данные. Неважно, есть ли в ней данные или нет, одна оболочка отличается от другой, даже если содержит дубликаты объектов:

```
Object x = new Object();
Object y = new Object();
assert x.equals(y); // не выполняется
```

Это вполне допустимый фрагмент кода, который показывает, что эти два объекта — пустые оболочки, не содержащие никаких данных. Они, конечно же, не равны друг другу, поскольку являются разными оболочками. Вот так Java рассматривает объекты. И это совершенно неправильно. Объект не может существовать без состояния, и оно должно быть его идентичностью.

Раз уж мы пришли к мнению, что инкапсулированные объекты являются частью идентичности, пора решить, сколько объектов *разумно* инкапсулировать. Как я уже говорил, не более четырех — разумное количество. Почему четыре? Вот мое обоснование.

Идентичность объекта — это своего рода его координаты во Вселенной. Моя идентичность — это имя и дата рождения. Используя эти два свойства, вы можете найти меня во всей Вселенной (если, конечно, планета Земля и наше представление о времени — единственно существующие координатные пространства в ней). У моей машины есть производитель, модель и год выпуска. Эти три свойства уникально идентифицируют ее во Вселенной. Я могу привести еще несколько примеров, но их смысл в том, что наличие более четырех координат противоречит здравому смыслу. При существующем уровне понимания объектов во Вселенной тяжело понять что-то более сложное. Один из моих рецензентов привел следующий контрпример. Он сказал, что если у него и у его соседа машины одной и той же модели, одних и тех же производителя и года выпуска, то они все же разные. Это правда, но только потому, что в контрпримере машина намного более сложна, чем в моем объектно-ориентированном примере. Если наш объект обозначает реальную машину из реального мира, то, конечно же, у него должно быть намного больше координат и атрибутов, чтобы его можно было идентифицировать уникальным образом. Но эти атрибуты будут сгруппированы в другие объекты, организованные в виде дерева. Скажем, машина будет инкапсулировать тип и идентификационный номер VIN. Тип будет инкапсулировать производителя, модель и год выпуска. Таким образом, мы получаем небольшое дерево объектов.

Я, безусловно, видел классы, инкапсулирующие десятки объектов. Это совершенно неправильно. Не делайте так. Четыре, не более. Если вам нужно инкапсулировать больше объектов, разбейте класс на несколько меньших.

И, к слову, чтобы устранить упомянутый недостаток Java, советую вам избегать оператора `==` и всегда переопределять метод `equals()`[1].

2.2. Инкапсулируйте хотя бы что-нибудь

Обсуждение на http://goo.gl/QE9aXg.

Другая крайность — объект, не инкапсулирующий вообще ничего. Например (алгоритм некорректен, но здесь важно не это):

```
class Year {
  int read() {
    return System.currentTimeMillis()
      / (1000 * 60 * 60 * 24 * 30 * 12) - 1970;
  }
}
```

Экземпляр этого класса ничего не инкапсулирует, то есть с учетом сказанного в разделе 2.1 все объекты класса `Year` будут равны друг другу, верно? Такой подход тоже плох. Инкапсулировать слишком много — плохо, но ничего не инкапсулировать тоже не годится.

Класс без свойств похож на статический метод, а это ужасная вещь в объектно-ориентированном программировании (см. раздел 3.2). У такого класса нет состояния и идентичности, только поведение. «Что с этим не так?» — спросите вы. Ответ прост. В чистом ООП без статических методов и со строгим разделением инстанцирования и исполнения (см. раздел 3.6) такое технически невозможно.

Инстанцирование должно быть отделено от исполнения, что означает следующее: оператор `new` разрешен лишь в конструкторах

[1] Для упрощения кода я использую @EqualsAndHashCode из проекта Lombok. — *Примеч. авт.*

(подробнее читайте в разделе 3.6). Пока предположим, что в истинно объектно-ориентированном проектировании использовать оператор new разрешено только в конструкторе.

Теперь рассмотрим класс, приведенный ранее. Его метод read() применяет статический метод из класса-утилиты System. В чистом ООП не может быть статических методов, и такой вызов сделать невозможно. Вместо этого придется создать экземпляр некоторого класса, который получит значение системных часов. Вот как это будет выглядеть:

```
class Year {
  private Millis millis;
  Year(Millis msec) {
    this.millis = msec;
  }
  int read() {
    return this.millis.read()
      / (1000 * 60 * 60 * 24 * 30 * 12) - 1970;
  }
}
```

Мы всегда должны что-то инкапсулировать, за исключением тех случаев, когда объект ничтожен или близок к таковому. Под ничтожным объектом я понимаю сущность, не имеющую координат во Вселенной. Только ей будет нечего инкапсулировать, поскольку она единственна и не нуждается в других сущностях для выживания и позиционирования. Напротив, любой объект, который что-либо делает, сосуществует с другими объектами и использует их. Он должен их инкапсулировать, чтобы идентифицировать себя. Это может звучать абстрактно и философски. Так и должно быть. Этому нет практического обоснования. Мы определенно можем создать объект, который ничего не инкапсулирует, и этому есть масса примеров. Но это неверно и с философской, и с практической точки зрения.

А еще посмотрим на эту проблему под другим углом. Как говорилось в разделе 2.1, инкапсулированное состояние — это уни-

кальный идентификатор объекта, который позиционирует его во Вселенной. Если инкапсулированных объектов нет, каковы его координаты? Они равны всей Вселенной:

```
class Universe {
}
```

Такой класс может существовать, но только единожды, потому что Вселенная единственная. Я, однако же, не вижу практических причин для его существования.

К слову, ранее я упомянул, что такой подход лучше, но он неидеален.

Вот как выглядел бы идеально спроектированный в канонах объектно-ориентированного программирования класс:

```
class Year {
  private Number num;
  Year(final Millis msec) {
    this.num = new Min(
      new Div(
        msec,
        new Mul(1000, 60, 60, 24, 30, 12)
      ),
      1970
    );
  }
  int read() {
    return this.num.intValue();
  }
}
```

Или как-то так:

```
class Year {
  private Number num;
  Year(final Millis msec) {
    this.num = msec.div(
      1000.mul(60).mul(60).mul(24).mul(30).mul(12)
    ).min(1970);
  }
  int read() {
```

```
    return this.num.intValue();
  }
}
```

Но подробнее об этом позже.

Bharat Savani написал 4 марта 2016 года:

Вопрос, наверное, глупый, но хотелось бы спросить вот что. Какая разница между инкапсуляцией и сокрытием данных? Я обратил внимание, что окружающие меня люди с одинаковой частотой применяют оба этих термина. Означают ли они одно и то же? Поясните, пожалуйста, на примере.

Егор Бугаенко:

Я считаю, что сокрытие данных — это суть POJO-объектов (Plain Old Java Objects). Они не имеют ничего общего с ООП, а просто прячут данные за геттерами и сеттерами. Суть инкапсуляции — в делегировании ответственности объекту. Таким образом объект получает право управлять своими (и не только) данными удобным для себя способом.

2.3. Всегда используйте интерфейсы

Обсуждение на http://goo.gl/vo9F2g.

Теперь поговорим о миссии объекта в том мире, в котором ему предстоит жить. Как я уже говорил, объект — это живой организм, который общается с другими организмами и помогает им делать их работу. Они, в свою очередь, помогают ему делать его работу.

Объект живет в тесном социальном окружении.

Под этим я понимаю то, что объекты *взаимосвязаны*, поскольку они нуждаются друг в друге. В самом начале, когда мы точно знаем, что каждый объект должен делать и какие услуги предоставлять другим объектам, все просто. Но когда приложение начинает разрастаться и количество объектов превышает несколько десятков, *тесная связь* между ними становится серьезной проблемой. И эта проблема влияет на сопровождаемость. Все сводится к сопровождаемости. Каждый раздел данной книги должен убедить вас задумываться в первую очередь о сопровождаемости. Она важнее всего остального, включая производительность.

Чтобы повысить сопровождаемость приложения в целом, мы должны приложить максимум усилий к *расцеплению* (decoupling) объектов. Технически это означает возможность модифицировать объект, не модифицируя связанные с ним объекты. Лучший инструмент для этого — *интерфейсы*.

Например:

```
interface Cash {
  Cash multiply(float factor);
}
```

Это интерфейс. Иными словами, это *контракт*, которому должен подчиняться объект, чтобы общаться с другими объектами. Вот как это выглядит:

```
class DefaultCash implements Cash {
  private int dollars;
  DefaultCash(int dlr) {
    this.dollars = dlr;
  }
  @Override
  Cash multiply(float factor) {
    return new DefaultCash(this.dollars * factor);
  }
}
```

Теперь, когда мне понадобится сумма в долларах, я могу рассчитывать на контракт, а не на конкретную его реализацию:

```
class Employee {
  private Cash salary;
}
```

Класс `Employee` не особо интересует, как реализован интерфейс `Cash`. Его не интересует, как работает метод `multiply()`. Он попросту не знает, как тот работает. Это означает, что интерфейс `Cash` помогает нам расцепить классы `Employee` и `DefaultCash`. Теперь я могу поменять класс `DefaultCash` или даже заменить его чем-то еще. Классу `Employee` все равно.

Я уверен, что все это очевидно, но вот вам мой совет: удостоверьтесь, что *все* публичные методы класса реализуют какой-то интерфейс. Грамотно спроектированный класс не должен содержать публичных методов, которые не реализуют хотя бы один интерфейс. Иными словами, неприемлем следующий класс:

```
class Cash {
  public int cents() {
    // какой-то код
  }
}
```

Метод `cents()` ничего не переопределяет, а так нельзя. Такой подход способствует сильному сцеплению класса с его пользователями (другими классами). Объекты других классов будут использовать `Cash.cents()` напрямую, что в дальнейшем станет препятствовать замене реализации этого метода на новую.

Небольшое философское замечание: класс существует только потому, что кому-то нужны его услуги. Эти услуги должны быть где-то документированы, например в контракте (интерфейсе). Кроме того, между поставщиками услуг должна существовать конкуренция. Именно в этом суть нескольких классов, реализующих один и тот же интерфейс. Каждый из конкурентов должен быть легко заменяем другим. В этом и заключается суть *слабого сцепления*.

Можно сказать, что, хотя классы больше не сцеплены напрямую, они сцепляются через интерфейсы. Класс должен реализовывать интерфейс, чтобы его могли понимать и использовать другие классы. Мы не можем поменять интерфейс, не внося

непосредственные изменения во все классы, которые реализуют и применяют его. Это действительно так. Сцепление все равно существует, и избавиться от него невозможно. Вообще говоря, такого рода сцепление — не такая уж и плохая вещь. Оно позволяет поддерживать всю систему в стабильном состоянии. Ее не получится сломать случайными изменениями в одной из частей за счет того, что другая ее часть не знает об этих изменениях. Интерфейсы, играя роль контрактов между частями системы, помогают поддерживать организованность окружения в целом.

Ryan Goodrich спросил 3 февраля 2016 года:

Вы говорите, что каждый публичный метод должен реализовывать некоторый интерфейс. Но разве это не может стать несколько избыточным и ненужным? Мне кажется, что если бы я делал так в большей части своего проекта, то у меня оказалось бы много интерфейсов, которые используются лишь однажды и зашумляют код.

Егор Бугаенко:

Если у вас нет юнит-тестов, то вы действительно будете использовать интерфейсы лишь однажды. Однако если вы пишете грамотные юнит-тесты, то для создания фиктивных объектов интерфейсы понадобятся. Следовательно, в таком случае вы будете использовать интерфейсы минимум дважды.

Ryan Goodrich:

Что вы думаете о применении абстрактных классов вместо интерфейсов для этих целей? Это вообще уместно?

Егор Бугаенко:

Я думаю, что на сегодняшний день больше не стоит задействовать абстрактные классы. Почитайте мои недавние статьи на эту тему.

asicfr спросил 22 января 2016 года:

Можете ли вы пояснить, почему использовать PowerMock — плохо?

Егор Бугаенко:

Практика мокинга объектов плоха сама по себе. Но PowerMock доводит мокинг до крайности. Тесты, создаваемые в PowerMock, невозможно поддерживать, и они препятствуют рефакторингу объектов. Вы, по сути, создаете юнит-тесты на основе PowerMock, а потом, если в тестируемом объекте что-то меняется, попросту их выкидываете.

PowerMock — неплохой инструмент, но его нужно использовать очень аккуратно и очень редко. Почти никогда. Если вам нужно его применять, ваш код плохо написан. Перепишите его.

2.4. Тщательно выбирайте имена методов

Мы уже обсуждали именование классов в разделе 1.1. Теперь пора научиться правильно именовать объекты. Я предлагаю следующее эмпирическое правило: «строителей» называть *именами существительными*, «манипуляторов» — *глаголами*[1].

Строителями я называю такие методы, которые что-то конструируют и возвращают новый объект.

[1] Этот совет очень похож на предложенную Бертраном Мейером (Bertrand Meyer) в книге «Объектно-ориентированное конструирование программных систем» («Русская редакция», 2005) идею, состоящую в разделении методов объекта на две непересекающиеся категории — запросы и команды. — *Примеч. авт.*

Строители всегда что-то возвращают. Они никогда не возвращают `void`, и их имена всегда являются существительными, например:

```
int pow(int base, int power);
float speed();
Employee employee(int id);
String parsedCell(int x, int y);
```

Обратите внимание на последний метод — `parsedCell()`. Это не просто существительное, а существительное с прилагательным. Принципу это не противоречит, а имя становится более описательным. Это все еще существительное, но уже с дополнительной информацией. Не просто ячейка, а разобранная ячейка. Мы, вероятно, ожидаем, что данный метод вернет ячейку, содержимое которой было определенным способом преобразовано.

Манипуляторами я называю такие методы, которые изменяют сущность реального мира, абстрагируемую объектом. Они всегда возвращают `void`, и их имена всегда являются глаголами, например:

```
void save(String content);
void put(String key, Float value);
void remove(Employee emp);
void quicklyPrint(int id);
```

Обратите внимание на последний метод — `quicklyPrint()`. Это глагол с наречием. Ключевой элемент здесь — глагол print, наречие quickly просто уточняет его, дает больше информации о контексте и назначении метода.

Можете давать методам-строителям и методам-манипуляторам любые имена, но старайтесь придерживаться принципа «строители строят, а манипуляторы манипулируют». Третьего не дано. Не должно быть как методов, которые манипулируют и возвращают что-то, так и методов, которые одновременно

строят и манипулируют. Позвольте привести несколько плохих примеров:

```
// возвращает количество сохраненных байтов
int save(String content);
// возвращает TRUE, если ассоциативный массив был изменен
boolean put(String key, Float value);
// изменяет скорость и возвращает ее предыдущую величину
float speed(float val);
```

Метод `save()` спроектирован плохо, потому что является манипулятором. Он «сохраняет», но в то же время возвращает `int`, как будто он является строителем. Мы должны либо возвращать `void`, либо переименовать его в нечто наподобие `bytesSaved()`.

Та же проблема и с методом `put()`, который работает как манипулятор, но возвращает `boolean` как строитель. Единственное решение — возвращать `void`. Но мы хотим знать, изменилось ли значение данного ключа. В этом случае необходимо полностью перепроектировать класс и сделать так, чтобы этот метод возвращал, к примеру, экземпляр класса `PutOperation`. В него входит манипулятор `save()`, а статус «успех/отказ» будет возвращаться методом `success()`. Метод `speed()` сохраняет значение и возвращает предыдущее. Это еще один пример плохого проектирования, поскольку он одновременно и строитель, и манипулятор. Исправить его можно аналогично предыдущему примеру, введя класс `SaveSpeed` с двумя методами: один сохраняет значение скорости, а другой возвращает ее предыдущее значение.

Мы обсудим геттеры и сеттеры позже, в разделе 3.5. Здесь, мне кажется, очевидно, что использовать имена, начинающиеся на `get`, просто неправильно. Хотя бы потому, что get — это глагол, но геттеры, по сути, являются строителями, поскольку должны что-то возвращать. Это был мой первый аргумент против методов-геттеров.

Думаю, теперь я должен объяснить свою мысль. В ее пользу можно привести несколько аргументов.

Строители — это существительные

Во-первых, некорректно называть метод глаголом, если он что-то возвращает. Такое название противоречит идее объектного мышления. Когда я захожу в кафе, я не прошу испечь мне кекс или сварить чашку кофе. Я говорю: «Я хотел бы кекс» или «Я хотел бы чашку кофе». Говорить: «Испеките мне» или «Сварите мне» — весьма грубо. Меня не должно интересовать, как именно испечен этот кекс или как сварена чашка кофе. Как их готовить — частное дело конкретного кафе. У меня есть спрос на кекс или чашку кофе. Они могут удовлетворить его. Как именно это происходит внутри кафе, меня не касается. Вот класс, описывающий кафе:

```
class Bakery {
  Food cookBrownie();
  Drink brewCupOfCoffee(String flavor);
}
```

На самом деле эти два метода не являются методами объекта. Это *процедуры*. Такой принцип именования говорит о том, что мы не доверяем кафе как самостоятельной самоуправляемой сущности и указываем ей, что делать. Это процедурный, а не объектно-ориентированный подход. На языке C эти процедуры могли бы быть реализованы, например, так:

```
Food* cook_brownie() {
  // приготовить кекс
  // и вернуть его
}
Drink* brew_cup_of_coffee(char* flavor) {
  // сварить чашку кофе
  // и вернуть ее
}
```

Кафе в этих процедурах не участвует. У нас просто есть два набора машинных инструкций, записанных на языке C, и мы их вызываем. В C они называются функциями, но, по сути, являются процедурами, поскольку к функциональному программированию

почти не имеют отношения. Мы просим компьютер выполнить эти инструкции и вернуть нам результат. Мы думаем как компьютер, а не как объект. Мы не доверяем кафе и говорим: «Иди уже свари этот чертов кофе» — вместо того, чтобы попросить чашку кофе определенного вкуса и доверить получение результата, неважно какого, заведению.

Я не хочу много философствовать, но проблема именования носит весьма абстрактный и принципиальный характер. Грамотно названный метод помогает его пользователям понять, для чего был создан объект, каковы его миссия, цель существования и смысл жизни. Неграмотно названный метод может разрушить представление об объекте и способствовать тому, что его станут использовать как мешок с данными и набором процедур. Это типичная ошибка, которую часто делают разработчики ООП-библиотек, SDK, API и т. п. Объект — это *живой организм*, который знает, как выполнять свои обязанности, и хочет, чтобы его уважали. Он хочет работать по контракту, а не просто следовать инструкциям. В этом и есть его основное отличие. Он прямо как программист, правда?

Вот почему, когда название метода — глагол, оно, по сути, указывает объекту, что ему делать. А просить объект построить что-то невежливо и неуважительно по отношению к нему. Просто сообщите объекту, что должно быть построено, и пусть он сам решает, как это сделать. Все приведенные далее имена некорректны:

```
InputStream load(URL url);
String read(File file);
int add(int x, int y);
```

Их нужно заменить следующими:

```
InputStream stream(URL url);
String content(File file);
int sum(int x, int y);
```


Обратите внимание на то, что я предлагаю вместо `add(x,y)` использовать `sum(x,y)`. Это изменение может показаться мелким и несущественным, но оно создает большую разницу в восприятии. Мы не должны просить объект сложить `x` и `y`. Вместо этого должны просить его создать сумму `x` и `y` и вернуть получившийся объект. Действительно ли он найдет сумму? Я не знаю. Может быть. Все, что я знаю, — то, что результат будет выглядеть как сумма `x` и `y`. Опять же я не указываю объекту, что ему делать, а просто прошу его породить результат, который подчинялся бы определенному контракту — был целым числом. В Java и многих других языках число не объект, а скаляр. Это их недостаток. В истинно объектно-ориентированном окружении все является объектом, особенно строки, числа, логические переменные, биты и байты.

Это первый аргумент и первый сценарий. Мы получаем нечто от объекта, или, иными словами, просим его собрать нам что-нибудь. А теперь обсудим второй аргумент и второй сценарий, когда мы просим объект выполнить какое-то преобразование.

Манипуляторы — это глаголы

Объект — это представитель некоторой сущности внешнего мира. Объект класса `File` представляет файл на диске, объект класса `Pixel` — точку на экране, объект класса `Integer` — четыре байта ОЗУ. (Удивлены? Подробнее об этом — в разделе 3.4.)

Если нам нужно манипулировать сущностью внешнего мира, мы просим объект выполнить эту манипуляцию, например:

```
class Pixel {
  void paint(Color color);
}
Pixel center = new Pixel(50, 50);
center.paint(new Color("red"));
```

Мы просим объект `center` нарисовать на экране точку с координатами (50; 50). И не рассчитываем на то, что что-то должно быть построено. Мы хотим, чтобы во внешнем мире произошло изменение, а объект выступает его представителем. «И в чем же здесь отличие от процедуры? — спросите вы. — Название метода — глагол, и оно указывает объекту, что необходимо сделать». Вопрос справедливый, но ключевое отличие здесь — в возвращаемом результате.

Метод `paint()` не возвращает результата. В рамках метафоры кафе можно, к примеру, попросить бармена сделать музыку погромче. Сделает ли он громче? Может быть, да. Может быть, нет. Нашу просьбу могут проигнорировать. Это не будет грубым или неуважительным, поскольку мы ничего не ожидаем в ответ. Представьте, как бы это звучало в противном случае: «Сделайте музыку громче, а как сделаете — сообщите уровень громкости». Именно так выглядит манипулятор, который возвращает значение.

Чертовски неуважительно.

Отличие, стало быть, в возвращаемом значении. Только метод-строитель может возвращать значения, и его имя должно быть существительным. Если объект позволяет нам выполнять преобразования, его имя должно быть глаголом и он не должен ничего возвращать.

Я думаю, можно задействовать другое соглашение об именовании, не упуская из виду основной принцип. К примеру, при использовании паттерна «Строитель» к именам добавлять приставку `with`:

```
class Book {
  Book withAuthor(String author);
  Book withTitle(String title);
  Book withPage(Page page);
}
```


Имя `withTitle` — сокращение от `bookWithTitle`. Чтобы избежать использования префикса `book` во всех методах, мы можем ограничиться префиксом `with`. Принцип остается в силе — эти методы являются строителями, а их имена можно расценивать как существительные. Вообще говоря, я противник этого паттерна, поскольку он способствует созданию крупных объектов, которые неизбежно более сложны в поддержке и намного слабее связаны, чем компактные. Паттерн «Строитель» применяется, когда мы не хотим передавать много параметров в конструктор. В таких случаях он оказывается полезным. Но большое количество аргументов — это само по себе проблема.

Вместо того чтобы использовать паттерн «Строитель», стоит разбить сложные объекты на несколько более простых.

Короче говоря, не применяйте этот паттерн.

Примеры

Обсудим несколько практических примеров рефакторинга. Допустим, у нас есть метод, который сохраняет содержимое файла и возвращает количество сохраненных байтов:

```
class Document {
  int write(InputStream content);
}
```

Метод выглядит корректно, но нарушает только что описанный принцип. Он должен возвращать `void`, но нам-то нужно знать, сколько байтов было фактически сохранено на диск. Что делать? Переименовать его в `bytesWritten()`? Это неправильно, поскольку этот метод предназначен для записи файла документа на диск, а не для подсчета байтов.

Принцип именования «строитель/манипулятор» в данном примере говорит о том, что метод `write()` берет на себя слишком много обязанностей. Он записывает данные и считает количество

байтов. Это слишком сложно для одного метода. Мы не можем четко назвать его глаголом или существительным, поскольку его назначение неоднозначно. Он расплывчат, несфокусирован.

Вот как я рекомендовал бы его переработать:

```
class Document {
  OutputPipe output();
}
class OutputPipe {
  void write(InputStream content);
  int bytes();
  long time();
}
```

Как видите, метод `output()` — строитель. Он создает новый объект типа `OutputPipe`, готовый записывать данные (обратите внимание на то, что я не назвал его writer). Данные еще не записаны — мы просто получаем объект, готовый выполнить эту операцию. Затем вызываем метод `write()` объекта `pipe`, который собирает данные о транзакции. Теперь можно получить больше информации, чем просто количество байтов. Можно получить время, затраченное на транзакцию, и многое другое.

Разработчики языка Go, на мой взгляд, сделали большую ошибку. Они позволили возвращать из метода несколько значений. В Go мы можем объявить метод `write()` примерно так:

```
type Document struct {}
func (d Document) write(s Stream) (int, int) {}
```

Именно так код и становится грязным и неуправляемым, а ведь весь смысл ООП — в снижении сложности путем изоляции концептов. Чем меньший концепт изолируется, тем легче его понимать и сопровождать. В данном случае это концепт «записи байтов в файл документа». Выполнив предложенную ранее переработку, я изолирую это понятие в отдельный класс `OutputPipe`, а Go побуждает программиста оставаться в контексте класса `Document` и еще больше усложнять его метод `write()`.

Методы, возвращающие логические значения

Погодите, а как насчет методов, возвращающих логические значения? Возьмем, к примеру, метод `isEmpty()` из класса `String`. «Как бы вы его назвали? — спросите вы. — А метод `equals()` в классе `Object`? А метод `exists()` в классе `File`? Их полно повсюду». Если придерживаться описанных ранее принципов, то можно сделать вывод, что все эти имена некорректны. Но какова альтернатива?

Как мне кажется, методы, возвращающие логические значения, являются исключениями из этих правил. Они тоже строители, но для лучшей читаемости их имена необходимо сделать *прилагательными*, например:

```
boolean empty();
boolean readable();
boolean negative();
```

Префикс `is` избыточен и не должен использоваться явно, но имеет смысл мысленно ставить его перед именем метода, чтобы убедиться в том, что оно подобрано грамотно.

Подставьте префикс и прочитайте имя, но применяйте его без префикса. Такие мысленные упражнения необходимы, чтобы избежать использования глаголов вместо прилагательных. Вот как, например, будут звучать имена приведенных ранее методов:

```
boolean empty(); // is empty
boolean readable(); // is readable
boolean negative(); // is negative
```

Однако вызовут проблемы следующие имена методов:

```
boolean equals(Object obj);
boolean exists();
```

Названия `isEquals` и `isExists` просто не звучат. Намного лучше будет использовать `equalTo` и `present`, поскольку фразы Is equal to и Is present звучат нормально.

Почему для методов, возвращающих логические значения, делается исключение? Потому что Java и большинство других языков особым образом работают с ними в рамках логических конструкций. Скажем, у нас есть класс `String`, имеющий, в свою очередь, метод-строитель `length()`. Мы добавляем к нему метод `emptiness()`, который возвращает состояние строки — пустая она или нет. И затем используем его следующим образом:

```
if (name.emptiness() == true) {
  // что-то сделать
}
```

Это читается нормально: «Если пустота имени истинна». Однако так в Java не делают. Там применяется сокращенная форма такого сравнения. Часть `==true` просто опускается. Поэтому прилагательное звучит лучше:

```
if (name.empty()) { // "если имя пустое"
  // что-то сделать
}
```

Позвольте обобщить данный раздел. Во-первых, знайте миссию своего метода. Он либо строитель, либо манипулятор. И ни в коем случае не может выполнять обе роли. Во-вторых, называйте строители именами существительными, а манипуляторы — именами прилагательными. Единственным исключением будет строитель, который возвращает логическое значение. В таком случае применяйте прилагательные.

Вот и все.

2.5. Не используйте публичные константы

Обсуждение на http://goo.gl/QlUoru.

Свойства, обозначаемые спецификаторами *public*, *static*, *final*, также известные как константы — популярный механизм совместного использования данных объектами. Именно для этого и нужны константы — для совместного применения данных или других объектов. И против этого я категорически возражаю. Объекты не должны ничего использовать совместно — они должны быть самодостаточными и очень закрытыми. Механизм совместного использования противоречит идее инкапсуляции и объектно-ориентированному образу мышления в целом. Рассмотрим это на примере. Скажем, у меня есть метод, который записывает структурированные данные в `Writer` и заканчивает каждую строку символом перевода строки:


```
class Records {
  private static final String EOL = "\r\n";
  void write(Writer out) {
    for (Record rec : this.all) {
      out.write(rec.toString());
      out.write(Records.EOL);
    }
  }
}
```

В данном примере статическое константное свойство `EOL` является приватным и используется только внутри класса `Records`. Такая ситуация вполне корректна. Мы не хотим каждый раз прописывать `"\r\n"` в явном виде внутри класса. Допустим, теперь у нас есть другой класс, который делает что-то похожее, но с другими объектами:

```
class Rows {
  private static final String EOL = "\r\n";
  void print(PrintStream pnt) {
    for (Row row : this.fetch()) {
      pnt.printf(
        "{ %s }%s", row, Rows.EOL
      );
    }
  }
}
```

У этого класса иная логика, он работает с совершенно другим набором объектов. Классы `Records` и `Rows` никак не связаны. У них нет ничего общего. Однако они оба определяют приватную константу `EOL`. Это будет дублированием кода? Да, конечно же. И как нам его предотвратить? Как мы его предотвращали в C? У нас был макрос `#define`, который позволял объявить ее однажды и затем применять повсюду:

```
#define EOL "\r\n"
```

Однако мы не пишем на C. В ООП у нас есть объекты, и решение проблемы дублирования кода путем использования публичных констант — совершенно некорректный подход. Он очень процедурен и поэтому неправилен. Вот как можно решить эту проблему в Java:

```
public class Constants {
  public static final String EOL = "\r\n";
}
```

Чем это отличается от макроса `#define` в C? Мало чем. И в том и в другом случае константы находятся в глобальной области видимости — каждый класс может их использовать. Я бы даже сказал, что макрос лучше, поскольку он виден *не всем*. Он становится видимым, только если включить `.h`-файл, в котором объявлен макрос. В Java класс `Constants` публичный, поэтому с точки зрения загрузчика классов он должен быть виден другим классам.

Вводя класс `Constants`, мы решаем проблему дублирования кода, поскольку классы `Records` и `Rows` будут использовать `Constants.EOL` вместо `Records.EOL` и `Rows.EOL` соответственно. Им больше не придется объявлять эту константу локально. Они будут применять доступную всем константу. Проблема решена, не так ли? Отнюдь!

Решая одну проблему, мы создали две бóльшие проблемы: привнесли *сцепление* и потеряли *цельность*.

Привнесение сцепления

Сначала рассмотрим проблему сцепления. Вот как сейчас выглядит класс `Records`:

```
class Records {
  void write(Writer out) {
    for (Record rec : this.all) {
      out.write(rec.toString());
      out.write(Constants.EOL); // здесь!
    }
  }
}
```

Класс `Rows` выглядит так:

```
class Rows {
  void print(PrintStream pnt) {
    for (Row row : this.fetch()) {
      pnt.printf(
      "{ %s }", row, Constants.EOL // здесь!
      );
    }
  }
}
```

Теперь они оба зависят от одного объекта, и эти зависимости *жестко запрограммированы*. Разорвать их непросто. В трех местах фрагменты кода сцеплены и взаимосвязаны между собой.

`Records.write()`, `Rows.print()` и `Constants.EOL`. Если я поменяю значение `Constants.EOL`, то поведение двух классов изменится непредсказуемым образом. Почему непредсказуемым? Поскольку, когда я меняю значение `Constants.EOL`, я понятия не имею, *как* оно используется. Может, для перевода строки при печати. Может, для завершения строки в протоколе HTTP, где поменять его невозможно из-за требований протокола.

Объект `Constants.EOL` одинок в глобальной области видимости, где он применяется без всякой семантики. Мы попросту не знаем, как задействуется этот объект, в каком контексте и как

вносимые изменения повлияют на его пользователей. Мы способствуем сцеплению, что со временем приведет к серьезному ухудшению сопровождаемости. Помните: все ради сопровождаемости! Если множество объектов использует другой объект неизвестным образом, то они очень тесно сцеплены с ним.

В случае, когда константа примитивна, как в нашем примере с `EOL`, проблема не так уж велика, поскольку семантика конца строки весьма прозрачна. Но когда константа становится более сложной, то и серьезность проблемы возрастает.

Потеря цельности

Применяя публичные константы, объекты становятся менее цельными, иными словами, менее ориентированными на решение собственных задач. Они должны *знать*, как обращаться с константами. И добавлять собственную *семантику* к глупым константам. Последние действительно довольно глупы. Что `Constants.EOL` знает о себе? Ничего. Это просто кусок текста, не понимающий, зачем он нужен. Он не знает своей миссии, своего назначения. Если начать философствовать: смысл жизни этой константы неясен.

Чтобы добавить семантику, мы должны писать дополнительный код в классах `Records` и `Rows`. Нам приходится оборачивать примитивные статические константы в некий код, который уточняет их назначение. Но не это является целью классов `Records` и `Rows`.

Они предназначены для работы с записями и строками, а не с символами завершения строки. Эти классы были бы более цельными, если бы могли перепоручать работу: «Я обрабатываю записи, а ты — концы строк» Так было бы разумно, поскольку это помогло бы классам быть более цельными.

Итак, какова же альтернатива? Вот что я предлагаю для качественного решения проблемы дублирования кода. Объекты не должны совместно использовать данные. Вместо этого мы

должны создавать *новые классы*, которые помогли бы классам совместно использовать функциональность. Не данные, а функциональность! К примеру, мы видим, что в обоих классах необходимо печатать строки, которые заканчиваются переводом строки. Создадим для этого класс:

```
class EOLString {
  private final String origin;
  EOLString(String src) {
    this.origin = src;
  }
  @Override
  String toString() {
    return String.format("%s\r\n", origin);
  }
}
```

Теперь при необходимости можно использовать их. К примеру, в классе `Records`:

```
class Records {
  void write(Writer out) {
    for (Record rec : this.all) {
      out.write(new EOLString(rec.toString()));
    }
  }
}
```

А так — в классе `Rows`:

```
class Rows {
  void print(PrintStream pnt) {
    for (Row row : this.fetch()) {
      pnt.print(
        new EOLString(
          String.format("{ %s }", row)
        )
      );
    }
  }
}
```

Теперь функциональность, добавляющая символы в конец строки, надежно изолирована в классе `EOLString`. Как конкретно

суффикс добавляется к строке, теперь его дело. В классах `Records` и `Rows` больше нет этой логики. Мы не знаем, как конкретно строка приобретает необходимый суффикс. Знаем лишь, что за эту задачу отвечает класс `EOLString`.

Вы можете возразить, что сцепление теперь будет с классом `EOLString`, что равносильно сцеплению с классом `Constants.EOL`, но это не так. Действительно, имеется сцепление с классом `EOLString`, но оно не снижает сопровождаемость, поскольку является сцеплением *посредством контракта*, а это значит, что классы можно разъединить. В таком сцеплении участвуют два равнозначных элемента — объект класса `Records` и объект класса `EOLString`. Последний из них работает по контракту, семантика которого инкапсулирована внутри класса.

Предположим, завтра мы захотим, чтобы поведение класса менялось в зависимости от платформы, на которой он работает. Допустим, мы не хотим использовать последовательность `"\r\n"` при работе под Windows. В такой ситуации нужно бросить исключение. Контракт (интерфейс) остается неизменным, но поведение меняется:

```
class EOLString {
  private final String origin;
  EOLString(String src) {
    this.origin = src;
  }

  String toString() {
    if (/* работаем под Windows */) {
      throw new IllegalStateException(
        "Извините, EOL невозможно использовать под Windows"
      );
    }
    return String.format("%s\r\n", origin);
  }
}
```

Можно ли было сделать так с помощью публичного статического литерала? Нет.

Значит ли это, что для каждой публичной константы надо создавать новый класс, инкапсулирующий ее семантику? Да. Значит ли это, что у нас могут быть сотни микроклассов вместо сотен константных строковых литералов? Да. Сделает ли это код многословным? Загромоздит ли его избыточными микроклассами? Нет. Чем больше мелких классов, тем чище код, при условии, что они не дублируют друг друга. Это утверждение может показаться вам нелогичным, но задумаемся на секунду — это важно. Я хочу сказать, что чем больше классов в вашем приложении, тем лучше оно спроектировано и тем легче его сопровождать. Наилучшей аналогией этому будет язык, на котором мы говорим. Чем больше слов вы применяете, если, конечно, это не синонимы, производящие впечатление на читателей, тем проще становится читать ваш текст. Напротив, когда вы вкладываете слишком широкий смысл в одно слово и часто его используете, текст читать труднее.

Вот фраза: «Мой кот любит есть рыбу и пить молоко».

А вот другая: «Моя вещь любит есть одну вещь и пить другую вещь». Здесь мы слишком часто задействуем слово «вещь» и вкладываем в него слишком большой смысл. Читатель должен разбираться, что означает «вещь» в первом, втором и третьем случаях. Слова «кот», «рыба», «молоко» позволяют быстрее уловить семантику написанного. То же происходит с классами, которые очень велики и имеют слишком много возможностей. Когда повсюду используешь класс `java.io.File`, иногда просто непонятно, что именно он означает. Намного более удобны `TextFile`, `JPGFile` или `TempFile`.

Позвольте привести еще один пример. Во всех известных мне HTTP-клиентах, не только в Java, есть возможность изменить HTTP-запрос следующим образом:

```
String body = new HttpRequest()
  .method("POST")
  .fetch();
```

А еще у вас есть набор публичных статических строковых литералов с именами HTTP-методов. В итоге код выглядит примерно так:

```
String body = new HttpRequest()
  .method(HttpMethods.POST)
  .fetch();
```

Это противоречит духу ООП. Намного лучше создать несколько небольших классов, представляющих эти методы:

```
String body = new PostRequest(new HttpRequest())
  .fetch();
```

`PostRequest` знает, как конфигурировать `HttpRequest` так, чтобы он делал POST-запрос вместо GET-запроса, выполняемого по умолчанию. Логика этой конфигурации, *семантика* литерала POST теперь инкапсулирована в новом классе `PostRequest`. Мы больше не должны помнить, что значит POST. Нам нужно выполнить POST-запрос, и нас не касается, как это происходит на уровне протокола HTTP.

Короче говоря, публичные константы в ООП — чистейшее зло, они не должны использоваться никогда. Им нет оправдания. Я понимаю, что современные библиотеки в Java, Ruby, PHP, Scala и им подобных, к сожалению, полны публичных констант. Не включайте их в *свой* код. Не создавайте себе дополнительных трудностей. Всегда заменяйте их микроклассами. Неважно, насколько малы они будут. Не решайте проблему дублирования кода публичными константами — применяйте классы.

Кстати, то же самое касается типов `enum` в Java. Перечисления ничем не отличаются от публичных констант, и их также необходимо избегать.

Martin написал 6 июля 2015 года:

Объясните, пожалуйста, чем зависимость от публичного статического константного атрибута отличается от зависимости от публичного класса, не считая ваших личных предпочтений относительно того, что вы считаете более объектно-ориентированным. Кстати, а какой объект реального мира представляет класс `UTF8String`? И еще, вы понимаете, что ваше решение порождает новый объект при каждом выполнении оператора присваивания?

Егор Бугаенко:

Вводя класс `UTF8String`, мы решаем проблему дублирования литерала `UTF-8`. Но решаем мы ее при помощи настоящего объектно-ориентированного подхода — инкапсулируем функциональность внутри класса и позволяем остальным инстанцировать и использовать его объекты. Тем самым решаем проблему дублирования функциональности, а не только дублирования данных. И да, я понимаю, что мое решение создает новый объект всякий раз, когда происходит присваивание. Мне кажется, что по сравнению с сильным сцеплением это небольшая проблема. В общем случае я отдаю предпочтение сопровождаемости (чистому коду), а не скорости.

2.6. Делайте классы неизменяемыми

Обсуждение на http://goo.gl/z1XGjO.

Делайте все классы неизменяемыми — это сильно улучшит сопровождаемость. Как и все остальное, о чем говорится в данной книге, неизменяемость помогает сделать классы небольшими, цельными, расцепленными и хорошо сопровождаемыми. Если фрагмент кода легко понять, то его несложно поддерживать. Неизменяемый класс намного проще понять, чем изменяемый. Если вы заставите себя думать в терминах неизменяемых объектов, ваш код станет чище, короче и проще для понимания.

Обсудим, что значит *неизменяемость*, а затем я покажу вам парочку практических преимуществ, которые она дает.

Объект называется неизменяемым, если его состояние нельзя изменить после создания. К примеру, является изменяемым объект приведенного далее класса:

```
class Cash {
  private int dollars;
  public void setDollars(int val) {
    this.dollars = val;
  }
}
```

Вот похожий класс, объекты которого неизменяемы:

```
class Cash {
  private final int dollars;
  Cash(int val) {
    this.dollars = val;
  }
}
```

Как видите, разница — в наличии у приватного свойства `dollars` ключего слова `final`. Оно говорит компилятору, что любые попытки изменить свойство вне конструктора должны приводить к ошибке компиляции. Неизменяемый объект инкапсулирует все необходимое и ничего не может поменять в процессе существования. Если нам необходимо изменить неизменяемый объект, то придется создавать новый объект. К примеру, мы хотим реализовать несложную арифметическую операцию умножения для денежного класса `Cash`. Вот пример изменяемого класса:

```
class Cash {
  private int dollars;
  public void mul(int factor) {
    this.dollars *= factor;
  }
}
```

Здесь мы делаем то же, но при помощи неизменяемого класса:

```
class Cash {
  private final int dollars;
  public Cash mul(int factor) {
    return new Cash(this.dollars * factor);
  }
}
```


Разница очевидна. Неизменяемый объект не может никак модифицировать себя. Он будет пытаться создать другой объект с желаемыми характеристиками.

Изменяемый объект мы будем использовать следующим образом:

```
Cash five = new Cash(5);
five.mul(10);
System.out.println(five); // будет выведено "$50"
```

А так станем работать с неизменяемым классом:

```
Cash five = new Cash(5);
Cash fifty = five.mul(10);
System.out.println(fifty); // будет выведено "$50"
```

Я подвожу к тому, что никогда не стоит делать объекты изменяемыми — всегда работайте с неизменяемыми объектами. Изменяемые объекты — злоупотребление объектно-ориентированной парадигмой. Последние два фрагмента кода идеально иллюстрируют эту мысль. Как только был инстанцирован объект `five`, он не сможет стать объектом `fifty`. Пять — всегда пять и будет пятью до конца своего жизненного цикла. Если нам нужно *пятьдесят*, то мы должны инстанцировать новый объект. Еще раз взглянем на код:

```
Cash five = new Cash(5);
five.mul(10);
System.out.println(five); // ой, это же $50
```

Последняя строчка сбивает с толку, не так ли? Мы ожидаем, что объект `five` будет вести себя как пять долларов, но он демонстрирует поведение как у 50 долларов. Надеюсь, это хорошо показывает, насколько изменяемость делает код сложным для понимания и поддержки. Изменяемость привносит беспорядок.

Вы можете возразить, что если мы назовем переменную `money`, то решим эту проблему и код снова станет читаемым.

```
Cash money = new Cash(5); // вот $5
money.mul(10);
System.out.println(money); // а вот $50
```

Может быть, но только в таком простом примере, как этот. Причем в очень ограниченной области. Мы фактически заменили конкретное имя на более абстрактное. В общем случае такая тактика плоха. В граничном случае мы должны будем назвать все переменные `var`. Не стоит объяснять, почему так не надо делать.

Уясните следующее. Я не говорю, что неизменяемые классы лучше изменяемых, что они более эффективны в некоторых ситуациях, могут решать некоторые проблемы более элегантно или использоваться чаще изменяемых. Совсем нет. Я говорю, что изменяемые объекты не имеют права на существование. Их использование должно быть *строго запрещено*. Их просто не должно быть в ООП, как это сделано, например, в Haskell. Все классы должны инстанцировать неизменяемые объекты, которые никогда не меняют своего состояния вне зависимости от области применения, будь то игры, пользовательский интерфейс, мобильные или веб-приложения или даже алгоритмы.

Есть несколько хорошо известных аргументов в пользу неизменяемости[1]. Кратко просмотрим их и обсудим контраргументы, которые часто сводятся к тому, что «неизменяемые объекты хороши, но не в нашем проекте».

Погодите. Прежде чем перейти к рассмотрению этих аргументов, обсудим «ленивую» инициализацию, которая технически невозможна при неизменяемых объектах. Как минимум в Java, Ruby, C++ и еще нескольких известных мне языках. Объект инициализируется «лениво», если он обновляет инкапсулированные свойства по требованию, например:

[1] *Brian Goetz et. al.* Java Concurrency in Practice. — Addison-Wesley Professional, 2006.


```
class Page {
  private final String uri;
  private String html;
  Page(String address) {
    this.uri = address;
    this.html = null;
  }
  public String content() {
    if (this.html == null) {
      this.html = /* загрузить из сети */
    }
    return this.html;
  }
}
```

Так работает «ленивая» инициализация. При создании объекта в поле `this.html` ничего нет. Вместо реальных данных оно содержит `null`. Затем, когда впервые вызывается метод `content()`, мы загружаем данные из сети и сохраняем их в упомянутом поле. При следующем вызове `content()` обращения к сети не происходит. Вместо этого мы возвращаем содержимое поля `this.html`. Очевидно, что этот класс изменяемый. Можем ли мы сделать его неизменяемым? Не в Java. А нужна ли нам вообще «ленивая» инициализация? Конечно. В основном из соображений производительности. Мы не хотим загружать страницу много раз, одного достаточно.

Мне кажется, что сам язык должен предоставлять такую возможность. Должно быть доступно нечто подобное:

```
@OnlyOnce
public String content() {
  return /* загрузить из сети */
}
```

Аннотация `@OnlyOnce` (или что-то похожее по смыслу) должна говорить компилятору, что помеченный ею метод будет

вызываться в объекте лишь единожды. Все последующие вызовы должны возвращать ранее возвращенное значение. К сожалению, на момент написания этих строк ничего подобного в Java не было. Есть несколько обходных путей, которые позволяют сделать объект неизменяемым, но при этом реализовать «ленивую» загрузку. Все это костыли. И обычно все это, по сути, разные механизмы кэширования, основанные на фреймворках или статических ассоциативных массивах. Я затрагивал эту тему ранее, в разделе 1.3, где приводил пример механизма кэширования, который может быть полезен, если вы не стремитесь к чистой неизменяемости.

Изменяемость идентичности

Неизменяемые объекты не имеют проблем с так называемой изменяемостью идентичности. Если коротко: данная проблема проявляется, когда мы сравниваем два объекта, которые выглядят равными, но впоследствии один из них меняет состояние. Они больше не равны, но мы думаем, что они равны. Или наоборот. Например, в Java:

```
Map<Cash, String> map = new HashMap<>();
Cash five = new Cash("$5");
Cash ten = new Cash("$10");
map.put(five, "five");
map.put(ten, "ten");
five.mul(2);
System.out.println(map); // {$10=>"five", $10=>"ten"}
```

Ассоциативный массив стал некорректным после нашего вмешательства. Он содержит два одинаковых ключа. Как это произошло? Сперва мы создали два *не равных* друг другу объекта `five` и `ten`. Затем поместили их в ассоциативный массив, класс `HashMap` которого создал два элемента, поскольку ключи были не равны. Затем мы изменили состояние одного из них с пяти на десять, используя модифицирующий метод `mul()`. Ассоциативный массив не знал об этом изменении. Мы его никак об

этом не уведомили. И не дали ему возможности сравнить ключи и удалить дубликаты. В итоге состояние ассоциативного массива оказалось некорректным.

Кроме того, если мы попытаемся извлечь один из них, то получим непредсказуемый результат, поскольку ассоциативный массив теперь испорчен:

```
map.get(five); // может вернуть либо "ten", либо "five"
```

Эта проблема известна как изменяемость идентичности. Взглянем на предыдущий пример. Если я уберу все строки, кроме последней, сможете ли вы догадаться, почему метод `map.toString()` возвращает такое странное состояние? Легко ли вам будет понять, почему `HashMap` содержит дубликаты ключей и как это получилось? А ведь в примере всего пять строк кода.

Такого не случится с неизменяемым объектом, поскольку после того, как он попадет в ассоциативный массив, он не сможет менять состояние. `HashMap` вычислит хеш-функцию от его состояния, поместит его во внутреннюю хеш-таблицу и оставит там. Единственный способ сделать что-либо с элементом ассоциативного массива — добавить в него новый объект-ключ.

Неизменяемые объекты полностью устраняют проблемы, связанные с изменяемостью идентичности.

Атомарность отказов

Еще одно преимущество неизменяемых объектов — атомарность отказов. То есть у нас есть либо полный и целостный объект, либо отказ[1] — и никаких промежуточных состояний.

Рассмотрим пример изменяемого класса `Cash`:

```
class Cash {
  private int dollars;
  private int cents;
```

[1] *Bloch J.* Effective Java. 2nd Edition. — Addison-Wesley, 2008.

```
  public void mul(int factor) {
    this.dollars *= factor;
    if (/* что-то не так */) {
      throw new RuntimeException("ой...");
    }
    this.cents *= factor;
  }
}
```

Когда я вызываю метод `mul()` и он вызывает исключение, половина объекта будет изменена (`this.dollars`), а другая остается неизменной (`this.cents`). Это может привести к очень серьезным ошибкам, которые опять-таки очень сложно найти. Неизменяемые объекты избавлены от такого недостатка, поскольку ничего не модифицируют внутри себя.

Вместо этого они инстанцируют новые объекты с новым состоянием:

```
class Cash {
  private final int dollars;
  private final int cents;
  public Cash mul(int factor) {
    if (/* что-то не так */) {
      throw new RuntimeException("ой...");
    }
    return new Cash(
      this.dollars * factor,
      this.cents * factor
    );
  }
}
```

Очевидно, что добиться атомарности отказов можно и при использовании изменяемых объектов, но придется уделить этому особое внимание. Работая же с неизменяемыми объектами, мы получаем атомарность «из коробки» — нет необходимости заботиться об этом, так как все объекты атомарны по определению. Что не так с обеспечением атомарности отказов в изменяемых объектах? Сложность объекта сильно увеличивается, и, как следствие, вероятность ошибки также повышается. И конечно же,

сопровождаемость такого объекта серьезно ухудшается. Вот как, к примеру, может выглядеть изменяемый и способный к атомарным отказам класс `Cash`:

```
class Cash {
  private int dollars;
  private int cents;
  public void mul(int factor) {
    int before = this.dollars;
    this.dollars *= factor;
    if (/* что-то не так с центами */) {
      this.dollars = before;
      throw new RuntimeException("ой...");
    }
    this.cents *= factor;
  }
}
```

Мы сохраняем значение поля `this.dollars` во временную переменную, чтобы иметь возможность восстановить его непосредственно перед вызовом исключения. Для небольших объектов это не очень важно, но, когда объект начинает увеличиваться в размерах, легко пропустить свойство, которое надо восстановить. Даже в таком маленьком объекте код весьма запутан, не находите?

К слову, я перечисляю преимущества по их важности в моем понимании — от наименее важных к наиболее важным. Поэтому наиболее важные преимущества впереди.

Временное сцепление

Еще одно преимущество, которое вы получаете в результате использования неизменяемых объектов, — отсутствие так называемого временно́го сцепления. Лучше всего объяснить это на примере:

```
Cash price = new Cash();
price.setDollars(29);
price.setCents(95);
System.out.println(price); // "$29.95"
```

Этот пример очень прост, но он показывает, как обычно инстанцируются *и* инициализируются изменяемые объекты. Вначале мы создаем скелет, в котором все внутренние свойства равны NULL (инстанцирование). Затем устанавливаем их значения с использованием методов-сеттеров (инициализация). Именно так JavaBeans, JPA, JAXB и другие стандарты рекомендуют работать с объектами в Java. Вы наверняка понимаете, что я, мягко говоря, небольшой любитель этих стандартов. Все они — хорошие инструменты для процедурных программистов, пишущих программы на Java, но они очень плохи с точки зрения истинного объектного мышления. Класс `Cash` — идеальный представитель JavaBeans — «мешков» с данными и пристегнутыми к ним процедурами. Сперва мы создаем «мешок», потом внедряем в него данные, а затем даем команду их обработать. Старайтесь держаться подальше от этих «стандартов»...

В приведенном ранее примере четыре строки. Они идут одна за другой в строго определенном порядке и связаны друг с другом в хронологическом порядке. Если я по ошибке переупорядочу их следующим образом:

```
Cash price = new Cash();
price.setDollars(29);
System.out.println(price); // "$29.00"!
price.setCents(95);
```

логика поломается, но код все же скомпилируется.

Этот пример очень прост, и вы можете возразить, что для такого переупорядочения нет никакого повода. Так не надо делать, и все тут. Это может быть действительно так в конкретном примере, поскольку я в состоянии разобраться в логике кода в течение пары секунд. Но я все равно должен *понимать* временно́е сцепление между строками, прежде чем менять их. Компилятор мне не поможет. Тем не менее переупорядочение строк — корректная операция. Иными словами, моя задача — запомнить, в каком порядке стоят строки. А если объектов много и мне нужно помнить

их порядок или порядок манипуляций с ними, то с *сопровождаемостью* возникнет большая проблема. Как насчет следующего фрагмента кода:

```
Cash price = new Cash();
// 50 строк кода для вычисления X
price.setDollars(x);
// еще 30 строк кода вычисляют Y
price.setCents(y);
// 25 строк кода делают что-то еще
System.out.println(price);
```

Легко ли понять, что данный конкретный порядок сеттеров должен быть сохранен и все они должны вызываться перед `println()`? Отнюдь. А так проблема решается при использовании неизменяемых классов:

```
Cash price = new Cash(29, 95);
System.out.println(price); // "$29.95"
```

Объект всегда инстанцируется единственной строкой кода. Мы попросту не можем отделить *инстанцирование* от *инициализации*. Они всегда должны быть вместе. Я не могу изменить порядок этих двух строк, поскольку в противном случае код не скомпилируется. Следовательно, неизменяемость полностью избавляет нас от временно́й связи между строками кода. Прежде чем что-то делать с объектом, я должен его инициализировать. То, что произойдет потом, не имеет значения. Объект — самодостаточная и целостная сущность. Больше ничего не нужно инициализировать.

Отсутствие побочных эффектов

Если объект изменяем, практически кто угодно может изменить его на лету. Допустим, я передаю объект `price` методу, который должен вывести его на экран. Но в этот метод закралась ошибка. Кроме вывода на экран, он также удваивает цену:

```
void print(Cash price) {
  System.out.println(
```

```
    "Today price is: " + price
  );
  price.mul(2);
  System.out.println(
    "Купи сегодня выгодно! Завтрашняя цена: " + price
  );
}
```

При вызове данного метода проявляется так называемый побочный эффект:

```
Cash five = new Cash(5);
print(five);
System.out.println(five); // ой..., $10
```

Чтобы понять, что происходит, потребуется некоторое время. Мне придется отладить каждую манипуляцию с объектом `five`, чтобы найти место, где возникает ошибка. Когда код настолько прост, отладка займет не больше пары минут. Но если в проекте несколько тысяч строк кода и несколько сотен классов, придется потратить несколько дней.

А если мой класс `Cash` — неизменяемый, никто и нигде не сможет изменить его объект. И я в этом уверен. Мне не придется просматривать весь код в поисках побочных эффектов. Неизменяемость класса `Cash` придает мне уверенность в том, что `five` означает пять долларов в любое время в любом месте кода.

Никаких нулевых (NULL) ссылок

В разделах 3.3 и 4.1 мы еще обсудим, почему использование `NULL` в ООП — абсолютное зло, а пока поговорим о неинициализированных свойствах объекта. Например:

```
class User {
  private final int id;
  private String name = null;
  public User(int num) {
    this.id = num;
  }
```



```
  public void setName(String txt) {
    this.name = txt;
  }
}
```

При создании экземпляра этого класса свойству `name` присваивается значение `NULL`. Оно будет инициализировано позже, при вызове `setName()` (при условии, что он вообще произойдет). А до тех пор будет равно `NULL`. «Что с этим не так? — спросите вы. — Просто проверяйте его на `NULL` перед его использованием — и вы в безопасности». Да, это верно, но код будет замусорен проверками `if name != null`. А если мы где-то забудем выполнить проверку, то получим исключение `NullPointerException` или ошибку сегментации в C++. Но это не самые главные проблемы. В конце концов, `NULL` не особо отличается, скажем, от пустой строки. Мы можем время от времени делать проверку, и в этом нет ничего страшного. Так бывает.

Главная же проблема намного серьезнее. И она касается... вы наверняка догадались... *сопровождаемости*. Объект, у которого значения свойств могут быть равны `NULL`, а не полезной информации, намного сложнее сопровождать, поскольку трудно понять, когда он является объектом, а когда превратился во что-то, что *объектом не является*. Позвольте объяснить, что я имею в виду. Но сперва задам вопрос: «Зачем может понадобиться иметь объект класса `User` с неинициализированным именем?» Действительно, когда и почему у нас может возникнуть такая необходимость?

Мне кажется, я знаю ответ. В большинстве случаев так происходит потому, что нам на самом деле нужен *другой* класс, но мы слишком ленивы, чтобы его ввести. Или не знаем, как его создать. Или не понимаем, что такое класс в ООП. Причин может быть много, но результат всегда один — чрезмерно большой класс, который является одновременно и пользователем, и покупателем, и работником, и записью в базе данных. Если поле `name` инициализировано, то это покупатель. Если оно равно `NULL`, то это пользователь, и т. д.

Мы просто не знаем, как использовать наследование и инкапсуляцию, чтобы разбить задачу на подзадачи. Поэтому при появлении новых требований задействуем один и тот же класс. Но, чтобы как-то управлять его разнообразным поведением, нам приходится применять временно не инициализированные свойства. По состоянию их инициализации (`NULL` или нет) мы определяем, что такое наш объект — пользователь, покупатель или SQL-запись. Думаю, не стоит говорить, что это ужасный подход.

Само существование константы `NULL` подталкивает нас придерживаться этого ужасного подхода. Если же вы сделаете все объекты неизменяемыми, внутри них не будет никаких `NULL`. Иными словами, вы будете *вынуждены* создавать небольшие, целостные и связные, а следовательно, лучше сопровождаемые объекты.

Потокобезопасность

Потокобезопасность — свойство объекта, буквально означающее, что он может быть использован параллельно из нескольких потоков и при этом результаты его работы будут *предсказуемыми*. Вот пример класса, инстанцирующего объекты, которые не являются потокобезопасными:

```
class Cash {
  private int dollars;
  private int cents;
  public void mul(int factor) {
    this.dollars *= factor;
    this.cents *= factor;
  }
}
```

Этот код выглядит безобидно, но посмотрим, что получится, если я запущу его в двух параллельных потоках:

```
Cash price = new Cash("$15.10");
// следующие две строчки исполняются в двух потоках
price.mul(2);
// ожидается "$30.20" или "$60.40"
System.out.println(price);
```


Попробуйте сами и убедитесь, что при каждом запуске выводятся разные числа. Ожидаются же только два корректных результата. Первый поток выводит `$30.20`, а второй — `$60.40`, это означает, что первый поток умножил число на два, а второй умножил его еще раз. Однако иногда будет выводиться значение `$60.20`. Почему так происходит и что это число означает в действительности? Как можно умножить `$15.10` на два и получить `$60.20`?

Очень просто. Один поток умножает количество долларов на два и количество центов на два, в то время как другой поток тоже умножает количество долларов на два, но не успевает умножить количество центов. Он, конечно, умножит их позже, но на несколько микросекунд объект `price` окажется в «сломанном» состоянии — доллары были умножены, а центы — нет.

Найти, отладить и исправить такое поведение — одна из сложнейших задач в первую очередь потому, что его очень сложно, а иногда и невозможно воспроизвести. Необходимо запускать тесты несколько раз, но при этом нет гарантии, что проблема проявится.

Неизменяемые объекты полностью решают эту проблему, предотвращая всякие изменения своего состояния во время работы программы. Не имеет значения, сколько потоков одновременно работают с объектом, — ни один из них не может изменить его состояния.

Изменяемый класс также можно сделать потокобезопасным, используя явную синхронизацию:

```
class Cash {
  private int dollars;
  private int cents;
  public void mul(int factor) {
    synchronized (this) {
      this.dollars *= factor;
      this.cents *= factor;
    }
  }
}
```

Это сработает, но при таком подходе возникает несколько проблем. Во-первых, не так-то просто обеспечить потокобезопасность изменяемого класса. Во-вторых, синхронизация всегда снижает производительность. Каждый поток должен ждать освобождения объекта, чтобы смочь с ним работать. Каждый поток устанавливает *монопольную блокировку* на объект, а остальные в это время находятся в режиме ожидания. А еще не забывайте о возможных взаимоблокировках. Темный лес, короче говоря. Я настоятельно рекомендую держаться от него подальше и использовать неизменяемые объекты.

К слову, вот пример кода для эксперимента с классом `Cash`:

```
class Cash {
  private int dollars;
  private int cents;
  Cash(final int dlr, final int cts) {
    this.dollars = dlr;
    this.cents = cts;
  }
  @Override
  public String toString() {
    return String.format(
      "$%d.%d", this.dollars, this.cents
    );
  }
  public void mul(int factor) {
    this.dollars *= factor;
    this.cents *= factor;
  }
}
final Cash cash = new Cash(15, 10);
final CountDownLatch start = new CountDownLatch(1);
final Callable<Object> script = new Callable<>() {
  @Override
  public Object call() throws Exception {
    start.await(); // блокировка здесь
    cash.mul(2);
    System.out.println(cash);
    return null;
  }
};
final ExecutorService svc =
  Executors.newCachedThreadPool();
svc.submit(script); // первый поток
svc.submit(script); // второй поток
start.countDown();
```


Запустите его пару раз в своей IDE и посмотрите, что он выведет. Ради интереса добавьте еще пару строчек `svc.submit(script)`.

Объект `script`, конечно же, должен в конце вызывать метод `shutdown()` у объекта `svc`. Я опустил эту часть для краткости.

Меньшие и более простые объекты

А теперь мое любимое преимущество неизменяемости — *простота*. Как вы уже поняли, простота означает более высокую сопровождаемость. Чем проще объекты, тем они более цельные и тем лучше сопровождаемые. Чем сложнее программное обеспечение, тем ниже квалификация программиста, его создавшего. Лучшее ПО — простое ПО, простое для понимания, модификации, документирования, поддержки и рефакторинга.

Сопровождаемость — главная добродетель в современном программировании.

В большинстве случаев простота означает меньшее количество строк кода. Чем короче класс, тем проще понять, что он делает, где у него недостатки, как его переработать. Если класс состоит из тысяч строк, то очевидно, что даже автор понятия не имеет, что в нем происходит. Я бы сказал, что в Java максимальный размер класса не должен превышать *250 строк кода* (вместе с комментариями и пробельными строками). Все, что превышает эту цифру, сигнализирует о немедленной необходимости рефакторинга класса. Для Ruby я бы предложил верхнюю границу в 100 строк кода.

Конкретное количество строк кода не имеет значения, если оно небольшое. Я видел классы из 5000 строк. Такое абсолютно недопустимо, и этому нет оправдания. Кстати, я видел такое даже в исходниках OpenJDK. Не говоря уже об Android SDK.

Если у вас получается выдерживать размер классов в пределах 250 строк в рамках всего приложения, то я бы сказал, что вы хороший разработчик и архитектор ПО. Если получается делать классы еще меньше, то вы великолепны. Здесь я говорю как о тестовом, так и о «боевом» коде.

Неизменяемые объекты по своей природе меньше изменямых хотя бы потому, что тяжело сделать неизменяемый объект слишком большим — его состояние инициализируется только в конструкторе. Вы не станете делать конструктор с десятью аргументами — он будет выглядеть ужасно, и это станет бросаться в глаза. Вы начнете с небольшого объекта с парой аргументов в конструкторе. Затем станете добавлять в него новые возможности, и по мере роста их количества будет увеличиваться количество аргументов в конструкторе. Добавляя в него очередную возможность, вы будете вынуждены сделать конструктор больше по размеру. Вскоре вы осознаете, что что-то пошло не так, и разобьете класс на несколько более мелких. Вы никогда не напишете неизменяемый класс размером 2000 строк кода.

Мне кажется, этот аргумент самый сильный из приведенных. Неизменяемость делает код классов чище и короче. Это самое важное преимущество, которое вы получаете, делая классы неизменяемыми.

В начале этого раздела я обещал обсудить аргументы против неизменяемости, но не буду делать это здесь. Сделаю это позже, в разделе 3.4, поскольку ответ на все эти аргументы один и тот же. Продержитесь еще пару разделов, и мы непременно доберемся до критики неизменяемости классов и моей позиции по ее поводу.

В обобщение данного подраздела позвольте повторить сказанное в разделе 2.6: я категорически против изменяемых объектов. В истинно объектно-ориентированном ПО существуют только неизменяемые объекты. Изменяемость — ужасное наследие процедурного программирования. Никогда не делайте классы изменяемыми. Точка.

Ben Grunfeld спросил 14 декабря 2017 года:

Как неизменяемые объекты предотвращают нулевые ссылки? Разве нельзя инициализировать неизменяемый объект как `NULL`? Приведите, пожалуйста, пример. Простите мою непонятливость.

Егор Бугаенко:

Я вообще не использую нулевые ссылки. Я либо инстанцирую объект со всеми необходимыми аргументами, либо бросаю исключение. На мой взгляд, нулевое значение атрибута объектного типа — идеальный пример плохо спроектированного приложения.

Jean-Paul Wenger спросил 10 октября 2017 года:

Как реализовать неизменяемую динамическую структуру данных (например, дерево, в котором дочерние узлы могут добавляться во время выполнения)?

Егор Бугаенко:

Неизменяемый объект — не значит константный.

Roland Kuhn написал 2 августа 2015 года:

Описываемые вами преимущества относятся к ссылочной прозрачности и в общем справедливы и корректны, но вы, к сожалению, упускаете основную идею объектно-ориентированного

проектирования. Из истоков ООП понятно, что объекты должны быть процессами, принимающими и отправляющими сообщения. Работы Алана Кея, основанные на модели акторов, гласят именно об этом. Конечно же, существуют объекты, которые никогда не меняют поведения, но при этом невозможно построить полезную распределенную систему, в которой все объекты (в изначальном смысле слова) неизменяемы, — такая система не позволяет выражать изменение состояния. Обобщая эти два пункта, скажу, что то, о чем вы пишете, является не объектно-ориентированным программированием, а проектированием, основанным на классах. Объекты, о которых вы говорите, — на самом деле сообщения, которыми обмениваются реальные объекты, и вот они действительно должны быть неизменяемыми, иначе проблем не избежать.

Егор Бугаенко:

Я категорически не согласен с этим. Это чисто процедурная точка зрения. Объекты — это НЕ процессы.

Jack написал 8 июня 2015 года:

Я обратил внимание: Егор часто говорит, что нечто есть плохо с точки зрения ООП, но не есть плохо с точки зрения функционального подхода. Объекты с окончанием -er, внедрение зависимостей, классы-утилиты — все это чрезвычайно важно для решения задач с применением функционального подхода.

Егор Бугаенко:

Функциональное и объектно-ориентированное программирование близки, но ООП имеет бóльшие возможности в силу наличия наследования, полиморфизма и т. п.

Jack:

Наследование легко осуществляется путем повторного использования функций. Полиморфизм можно заменить передачей ссылки на другой метод и т. д.

Егор Бугаенко:

Вы правы. Тогда я бы сказал, что ООП просто более интуитивно понятно. Оно моделирует действительность лучше, чем функциональная парадигма, поскольку мы в состоянии понять, что такое объект, не имея опыта программирования. Прежде чем мы сможем легко оперировать функциями, мы должны изучить их, приспособиться к ним.

Matteo Vaccari написал 26 октября 2014 года:

Многие проблемы можно естественным образом смоделировать изменяемыми объектами. Например, симуляторы, игры... Да, все это можно смоделировать с использованием функционального подхода, но в таком случае вы привязываетесь к частному способу моделирования. Я предпочитаю стиль, который ближе к моему интуитивному ощущению предметной области. У меня нет аргументов против неизменяемости, но у меня есть аргумент за изменяемость. Я за сокращение расстояния между ментальной моделью и программным кодом.

Егор Бугаенко:

Весомый аргумент, я согласен. Но я-то как раз и стремлюсь изменить/улучшить вашу ментальную модель. Вы привыкли моделировать в терминах изменяемых объектов. Поэтому вам намного удобнее делать так, как вы делаете. Как симуляторы, так и игры можно строить с применением неизменяемых объектов.

2.7. Пишите тесты, а не документацию

Документация — очень важная составляющая сопровождаемости. Даже скорее не документация, а доступность вспомогательной информации о конкретном классе или методе. Как читателю вашего кода, мне могут понадобиться дополнительные подробности или пояснения. Возможно, я не настолько умен, как вы.

Я могу не знать, как действует ваш алгоритм сортировки, что такое MD5, как работает конкретное регулярное выражение или каково назначение `/dev/null`. Все это вполне возможно. Из собственного опыта скажу, что чтение кода, написанного «всезнающим» программистом, вызывает огромное раздражение.

Чтобы сделать свой код лучше читаемым, *представьте*, что я начинающий программист, слабо понимающий предметную область, язык программирования, шаблоны проектирования и алгоритмы. Представьте, что я намного глупее вас. Так вы демонстрируете свое уважение ко мне. Не хвастайтесь своими способностями — пишите простой легко читаемый код. Плохие программисты пишут сложный код. Хорошие программисты пишут простой код.

Идеальный код говорит сам за себя и не требует дополнительной документации, например:

```
Employee jeff = department.employee("Jeff");
jeff.giveRaise(new Cash("$5,000"));
if (jeff.performance() < 3.5) {
  jeff.fire();
}
```

Нужно ли документировать такой код? Мне кажется, он достаточно прозрачен сам по себе. А как насчет этого фрагмента:

```
class Helper {
  int saveAndCheck(float x) { .. }
  float extract(String text) { .. }
  boolean convert(int value, boolean extra) { .. }
}
```

Ужасное имя класса (см. раздел 1.1), ужасные имена методов (см. раздел 2.4), класс в целом спроектирован отвратительно. Естественно, для него необходима документация. Я не могу понять, что он делает, зачем нужны его методы и как их использовать. Плохо спроектированные классы вынуждают писать для них документацию. Соответственно хорошо спроектированные

классы документации не требуют. Их назначение понятно, а код элегантен, например:

```
class WebPage {
  String content() { .. }
  void update(String content) { .. }
}
```

Мой вам совет: не документируйте код — делайте его чище.

Под этим я, в частности, понимаю написание юнит-тестов. Хотя юнит-тестирование стало общепринятой практикой относительно недавно[1], юнит-тест должен рассматриваться как часть класса наравне с методами, свойствами, именем и перечнем реализуемых интерфейсов. К сожалению, в большинстве языков (возможно, во *всех*) делается совершенно не так. В Java, к примеру, юнит-тест — это файл `.java`, содержащий еще один класс. Если класс называется `Cash`, то соответствующий класс теста будет по договоренности называться `CashTest`. Этот подход неидеален, поскольку он позволяет создавать классы без юнит-тестов. Такого быть не должно.

Юнит-тест — *часть* класса, а не самостоятельная сущность. Естественно, концептуально, а не технически. Во всех известных мне языках юнит-тесты технически реализуются в виде отдельных файлов.

Создавая чистые и сопровождаемые тесты, вы делаете сами классы чистыми и улучшаете их сопровождаемость. Поэтому чем лучше тест, тем меньше документации требует класс. Юнит-тест и *есть* документация. Должным образом написанный юнит-тест очень поможет понять ваш класс. При этом он интернационален. Чтобы понять юнит-тест на Java, нет необходимости владеть английским в совершенстве, но чтобы понять текст

[1] *Beck K.* Test-Driven Development by Example. — Addison Wesley, 2003.

Javadoc-документации, нужны определенные навыки чтения по-английски.

По аналогии с тем, что «одна картинка стоит тысячи слов», я бы сказал, что один юнит-тест стоит страницы документации. Юнит-тест *показывает* мне, как использовать класс, в то время как документация рассказывает историю, которую намного труднее понять и интерпретировать. Не говорите, а показывайте. И старайтесь делать демонстрацию занимательной. Если вам удастся сделать юнит-тест правильно, его будут читать даже чаще, чем код самого класса.

Лучший совет, как писать хорошие, качественные юнит-тесты: уделяйте им такое же внимание, как и основному коду. Есть много других хороших советов по созданию юнит-тестов и обеспечению их качества. В частности, хотелось бы выделить книги «Эффективная работа с унаследованным кодом»[1] и «Чистый код. Создание, анализ и рефакторинг»[2].

Некоторые рецензенты попросили меня привести пример хорошего юнит-теста. Я выполню просьбу, прежде всего чтобы проиллюстрировать эту главу примером кода. Проблема написания хороших юнит-тестов выходит за рамки книги. Вот что я назвал бы хорошим юнит-тестом для класса `Cash` (тест написан с использованием JUnit и Hamcrest):

```
class CashTest {
  @Test
  public void summarizes() {
    assertThat(
      new Cash("$5").plus(new Cash("$3")),
      equalTo(new Cash("$8"))
    )
```

[1] *Физерс М.* Эффективная работа с унаследованным кодом. — Вильямс, 2004.

[2] *Мартин Р. С.* Чистый код. Создание, анализ и рефакторинг. — Питер, 2018.


```
  }
  @Test
  public void deducts() {
    assertThat(
      new Cash("$7").plus(new Cash("-$11")),
      equalTo(new Cash("-$4"))
    )
  }
  @Test
  public void multiplies() {
    assertThat(
      new Cash("$2").mul(3),
      equalTo(new Cash("$6"))
    )
  }
}
```

Много хороших советов по поводу написания юнит-тестов можно найти в книге *Growing Object-Oriented Software, Guided by Tests*[1].

2.8. Используйте fake-объекты вместо mock-объектов

Обсуждение на http://goo.gl/OF3Cev.

Еще один раздел о юнит-тестировании — и хватит. На этот раз речь пойдет о *мокинге* как инструменте оптимизации тестов. Вот как это работает. Допустим, у нас есть класс `Cash`, умеющий конвертировать себя в другую валюту:

```
class Cash {
  private final Exchange exchange;
  private final int cents;
  public Cash(Exchange exch, int cnts) {
    this.exchange = exch;
```

[1] *Freeman S. et al.* Growing Object-Oriented Software, Guided by Tests. — Addison-Wesley Professional, 2009.

```
    this.cents = cnts;
  }
  public Cash in(String currency) {
    return new Cash( this.exchange,
      this.cents *
      this.exchange.rate(
        "USD", currency
      )
    );
  }
}
```

Этот класс зависит от класса `Exchange`, который знает конкретный курс конверсии долларов, скажем, в евро. Чтобы использовать класс `Cash`, мы должны передать в его конструктор экземпляр класса `Exchange`:

```
Cash dollar = new Cash(new NYSE("secret"), 100);
Cash euro = dollar.in("EUR");
```

В данном случае класс `NYSE` знает, как получить курс обмена евро на доллары с Нью-Йоркской фондовой биржи с помощью, к примеру, HTTP-запроса к ее серверу. Здесь я использую строку `"secret"` в качестве пароля к рабочему серверу биржи. Так класс `Cash` работает в реальном окружении, но мы не хотим, чтобы при каждом запуске юнит-тестов происходили обращения к рабочему серверу биржи. Мы также не хотим, чтобы программисты знали *реквизиты* этого сервера. Нам нужно найти способ протестировать класс `Cash`, не привлекая к этому сервер биржи.

Традиционный подход называется *мокингом*. Вместо того чтобы использовать сервер биржи `NYSE`, мы создаем имитацию интерфейса `Exchange` и передаем ее в качестве аргумента конструктору класса `Cash` (я применяю Mockito[1]):

```
Exchange exchange = Mockito.mock(Exchange.class);
Mockito.doReturn(1.15)
  .when(exchange)
```

[1] Насколько мне известно, он находится здесь: http://mockito.org/.


```
  .rate("USD", "EUR");
Cash dollar = new Cash(exchange, 500);
Cash euro = dollar.in("EUR");
assert "5.75".equals(euro.toString());
```

Уверен, вы знаете о таком приеме. Но я все равно решил его объяснить, чтобы было проще понять, почему я считаю его использование плохой практикой. Да, я утверждаю, что мокинг — плохая практика, применять его можно только в самых крайних случаях. Впрочем, если вы разрабатываете объекты в соответствии с рекомендациями, приводимыми в данной книге, мокинг вам не понадобится.

Я рекомендую вместо мокинга задействовать fake-объекты. Вот как интерфейс `Exchange` должен поставляться пользователям:

```
interface Exchange {
  float rate(String origin, String target);
  final class Fake implements Exchange {
    @Override
    float rate(String origin, String target) {
      return 1.2345;
    }
  }
}
```

Вложенный fake-класс — часть интерфейса и должен поставляться вместе с ним. Это важная часть интерфейса `Exchange`, поскольку он помогает применять его в юнит-тестах. Это еще не все — подробности рассмотрим позже. Теперь разберем юнит-тест, который использует fake-классы вместо мокинга:

```
Exchange exchange = new Exchange.Fake();
Cash dollar = new Cash(exchange, 500);
Cash euro = dollar.in("EUR");
assert "6.17".equals(euro.toString());
```

Выглядит короче, не правда ли? Вы можете возразить, что тест стал менее очевидным. К примеру, откуда появилось число `6,17`, если мы нигде не задаем курс конверсии? Это действительно

так. Но мы можем наделить fake-классы еще бóльшими возможностями. Можно сделать так, чтобы они возвращали инкапсулированный курс вместо константного. В целом fake-классы могут и должны быть весьма функциональными. Я бы даже сказал, что иногда они должны быть сложнее настоящих классов. Кроме того, они могут реализовывать нужную функциональность совершенно иным способом, нежели настоящие классы. Они могут действовать и реагировать на действия по-другому. Это не будет большой проблемой до тех пор, пока юнит-тесты не станут слишком сильно зависеть от их поведения. Не подстраивайте тесты под fake-классы.

Убедитесь, что fake-классы должным образом соответствуют вашим тестам.

Fake-классы сокращают размер тестов, что серьезно улучшает их сопровождаемость, тогда как мокинг делает тесты чересчур многословными и сложными для отладки и рефакторинга. Для простого интерфейса, такого как `Exchange`, это неочевидно, но мы тем не менее смогли сократить тест на одну строку. Когда тест включает пять объектов разных классов, у каждого из которых есть по нескольку методов, то спустя пару месяцев нагромождение вызовов Mockito перестанет быть понятным даже автору тестов.

Но проблема не только в объеме тестов. Она намного шире. Мокинг усложняет сопровождение тестов, поскольку *превращает предположения в факты*. Позвольте объяснить, что я имею в виду. Взгляните еще раз на эти строки:

```
Exchange exchange = Mockito.mock(Exchange.class);
Mockito.doReturn(1.15)
  .when(exchange)
  .rate("USD", "EUR");
```

Что конкретно мы хотим ими сказать? Мы буквально говорим: «Предположим, что класс `Cash` вызывает `Exchange.rate()`».

Весь юнит-тест построен на этом предположении. Мы не знаем этого наверняка, поскольку с точки зрения юнит-теста класс `Cash` — черный ящик. Мы не знаем, как именно реализован метод `Cash.in()` и как именно он использует экземпляр `Exchange`. Возможно, не использует его вообще. Мы не знаем этого, но делаем *предположение* и строим вокруг него весь тест. Мы превращаем предположения в факты. Говорим: «Вот как должен работать класс `Cash`!»

Это плохо. Очень плохо. Почему? Потому что противоречит общей цели юнит-тестирования — подстраховать процесс рефакторинга.

Юнит-тест помогает рефакторингу класса, поскольку дает негативный результат, когда что-то в поведении класса изменилось (истинное срабатывание). Но в то же время он не дает негативного результата, если я не менял поведение (ложное срабатывание). Это чрезвычайно важная вторая половина принципа в целом: тест *не должен давать отрицательный результат*, если видимое поведение класса не изменилось. Он не должен давать ложных срабатываний.

Однако наш тест может быть провален без всяких на то причин. И вот как. Допустим, мы хотим изменить интерфейс `Exchange` так, чтобы он выглядел следующим образом:

```
interface Exchange {
  float rate(String target);
  float rate(String origin, String target);
}
```

Первый метод (с одним аргументом) возвращает курс конверсии из долларов в целевую валюту, а второй (с двумя аргументами) позволяет указать исходную и целевую валюты.

Затем мы укажем классу `Cash`, что нужно использовать новый метод с одним аргументом, когда исходная валюта — доллары. Что произойдет с нашим юнит-тестом? Правильно, он покажет

ложное срабатывание. И укажет мне на сбой, которого на самом деле нет. Класс `Cash` по-прежнему работает и конвертирует валюты. Все в полном порядке, но тест показывает ошибку.

Это чертовски раздражает и полностью подрывает веру в полезность собственных юнит-тестов. И является одной из ключевых причин того, что очень многие программисты не любят и не применяют их. Они слишком хрупки и нестабильны, в основном из-за мокинга. Посмотрим, что произойдет в точно такой же ситуации, но при использовании fake-класса `Exchange.Fake` вместо мокинга.

При изменении интерфейса `Exchange` мы автоматически меняем реализацию класса `Exchange.Fake`, и теперь она выглядит следующим образом.

```
interface Exchange {
  float rate(String target);
  float rate(String origin, String target);
  final class Fake implements Exchange {
    @Override
    float rate(String target) {
      return this.rate("USD", target);
    }
    @Override
    float rate(String origin, String target) {
      return 1.2345;
    }
  }
}
```

Нужно ли изменять юнит-тест? Нет. Ломается ли он? Нет. Мы не поменяли поведение класса `Cash`, а юнит-тест не дал ложного срабатывания. Это хороший юнит-тест, я могу ему доверять.

Суть в том, что мокинг — изначально *плохой подход*. Его изобрели, чтобы помочь с юнит-тестированием, но эта помощь сомнительна. Он привязывает тесты к внутренним деталям реализации класса. Мы делаем предположения, жестко фиксируем их в коде и на этом закругляемся. Когда приходит время рефакторинга,

нам приходится удалять свои тесты, поскольку они связаны с уже несуществующими деталями реализации.

Напротив, fake-классы делают тесты полностью сопровождаемыми, поскольку нас не заботит то, как класс `Cash` взаимодействует с реализацией интерфейса `Exchange`. Взаимодействие этих двух классов не должно касаться юнит-теста класса `Cash`. Это личное дело класса `Cash`. Он может взаимодействовать с `Exchange`, а может и не взаимодействовать. Он может использовать метод с одним аргументом, а может — с двумя. Решать классу `Cash`. Мы не имеем права делать предположения о его личных решениях. Все, что нас интересует, — это то, как `Cash` взаимодействует с нами, а не то, как он взаимодействует с другими классами.

Вы можете возразить, что раз мы передаем `Cash` экземпляр `Exchange`, то имеем право знать, как он его применяет. Нет, не имеем. Мы не имеем права знать, *как* реализован объект. Привязка тестов к внутренним деталям реализации делает тест хрупким и несопровождаемым.

Мокинг — источник проблемы.

Повторюсь: мокинг — ужасный подход к юнит-тестированию.

Кроме того, большинство mock-фреймворков дает нам возможность узнать, осуществилось ли заданное взаимодействие с mock-объектом и сколько раз это произошло. На первый взгляд это удобная возможность, но по той же причине очень вредная. Ставя тесты в зависимость от взаимодействия классов, мы делаем рефакторинг болезненным, а иногда и невозможным. Мы не должны проверять, как объект работает со своими зависимостями. Эта информация инкапсулируется объектом. Иными словами, она спрятана от наших глаз. Секрет.

Но что делать, если с интерфейсом не поставляется fake-объект? Конечно, было бы идеально иметь fake-классы для всех интерфейсов, но в реальности это не так, правда? Да, именно. Действительность в общем случае намного менее элегантна, чем приемы,

описанные в данной книге. Но мы можем ее изменить (действительность, а не книгу). Начните со своих интерфейсов — оснастите их fake-классами. Убедитесь в том, что каждый создаваемый вами класс не имеет методов, не реализующих интерфейс (см. раздел 2.3). Предусмотрите fake-классы для всех написанных вами интерфейсов. Так вы начнете изменять мир. Пользователи ваших классов станут писать более качественные тесты, а количество мокинга в мире будет уменьшаться.

У fake-классов есть еще одно важное преимущество, которое я обещал описать, — они помогают лучше продумать и спроектировать интерфейс. Работая с интерфейсом и создавая для него fake-класс, вы неизбежно вынуждены думать как пользователь интерфейса, а не только как его разработчик. Посмотрите на него под другим углом и попытайтесь реализовать ту же функциональность, используя тестовые ресурсы.

Возьмем, к примеру, интерфейс `WebPage`. Его реализация по умолчанию должна делать HTTP GET-запрос для загрузки страницы и HTTP PUT-запрос для ее обновления. Но как реализовать для нее fake-класс? Где будет храниться содержимое страницы? Как обеспечить потокобезопасность операций чтения и обновления? Как работать с разными кодировками? Вопросов будет много. Суть в том, что, отвечая на них и находя оптимальное решение, вы непременно улучшаете интерфейс.

Поэтому держитесь подальше от мокинга и всегда создавайте fake-классы для своих интерфейсов.

Еще я могу привести несколько практических примеров больших fake-классов, которые мы применяем в своих проектах. Не просто классов, а пакетов классов и даже пакетов пакетов классов. Однажды, например, мы писали RESTful-клиент к API Github. Сам API весьма обширен — в нем порядка 150 точек входа. Для организации юнит-тестирования клиента мы создали полную копию API в виде fake-классов. Чтобы сохранять данные

и полностью имитировать GitHub, использовали XML-файл. Более 150 fake-классов обновляли это XML-хранилище, и ни один из них не подозревал, что взаимодействовал с поддельным GitHub, а не с настоящим сервером. Реализация библиотеки fake-классов заняла некоторое время, но оказалась ценным вложением, поскольку благодаря ей юнит-тесты упростились и уменьшились в размерах[1].

В другом случае уровень персистентности был реализован в СУБД AWS DynamoDB, а уровень модели реализован набором интерфейсов. Еще у нас были классы, реализующие эти интерфейсы путем реального взаимодействия с NoSQL-базой данных. Кроме того, в тестовых целях мы вложили в каждый интерфейс fake-класс, имитирующий персистентность с использованием текстовых файлов. Такой набор fake-классов сделал юнит-тесты намного короче и чище[2].

Dev Danke написал 13 февраля 2015 года:

> Я не согласен с этой статьей. Я не согласен, что многие думают, будто мокинг — зло и использовать его плохо. Хотя некоторые разработчики поддерживают это мнение, подавляющее большинство считает мокинг отличным решением. Еще я не согласен с тем, что создание собственного набора тестовых объектов лучше, чем применение любого из популярных фреймворков для мокинга. Мокинг и юнит-тестирование являются передовыми практиками. Мокинг и юнит-тестирование широко применяются практически во всех средних и крупных компаниях и организациях. Использование фреймворков для мокинга существенно расширяется. Только взгляните на статистику загрузок популярных фреймворков для мокинга, например Mockito. Их применение растет экспоненциально! Популярность

[1] http://github.com/jcabi/jcabi-github.
[2] http://github.com/yegor256/rultor.

фреймворков для мокинга увеличивается, поскольку разработчики осознают, что тем самым они экономят время и получают возможность быстро писать более качественные юнит-тесты. Юнит-тесты, которые они пишут, проще для понимания и поддержки другими разработчиками. Использование самодельных тестовых объектов для юнит-тестирования — антипаттерн. Любой, кто так делает, на самом деле создает собственный фреймворк для мокинга. Почти наверняка он будет представлять собой жалкую, плохо документированную имитацию (если она вообще будет документирована). Других разработчиков вряд ли обрадует применение причудливых самодельных тестовых объектов вместо общепринятых стандартных фреймворков для мокинга. И не забывайте, что время, затраченное на создание и документирование самодельных тестовых объектов, могло быть израсходовано на усовершенствование программного обеспечения вашей компании.

Jacob написал 7 февраля 2015 года:

Не существует решения для юнит-тестирования, которое подошло бы для любого проекта, но вот несколько аргументов.

1. Меня не очень убеждает ваш аргумент о сопровождаемости. При использовании инструмента вроде Spock вам всего лишь понадобится объявить три поля и написать одну строчку в каждой спецификации взаимодействия с коллаборатором. Похоже, что ваше решение станет порождать малоприятные тестовые классы, когда появится необходимость тестировать всевозможные сценарии, требующие собственной реализации (к примеру, проверка обработки исключительных ситуаций). Фреймворки для мокинга позволяют создавать управляемые заглушки для коллабораторов с минимальными накладными расходами.
2. Возможно, я что-то упустил, но то, что вы предлагаете, — не совсем юнит-тестирование, а нечто похожее на заглушечное интеграционное тестирование. Одна из важнейших частей использования фреймворков для мокинга — создание mock-объектов, которые записывают и подтверждают взаимодействие объектов в рамках управляемых сценариев.

Invisible Arrow спросил 13 декабря 2014 года:

Не приведет ли включение mock-объектов в тот же модуль к непреднамеренному использованию их в рабочем коде? Я предпочел бы, чтобы они находились в отдельном модуле, который я мог бы применять в тестовой области сборки зависимого модуля.

Егор Бугаенко:

Именно так я и делал некоторое время назад — создавал дополнительный модуль, используемый в тестовой области. Около года тому назад я понял, что такой подход сложнее, чем просто расположить fake-классы рядом с настоящими. Я понял ваш довод — и он совершенно верен: fake-объекты могут быть применены в рабочем коде по ошибке. На это у меня нет ответа. Возможно, потом я найду какие-то доводы и напишу здесь о них.

Invisible Arrow написал 13 декабря 2014 года:

В случае применения некоторых сторонних библиотек, которые не поставляются с mock-объектами, мокинг, вероятно, окажется быстрее создания fake-класса. Например, класс `ServletContext`, имеющий более 30 методов, объявленных в интерфейсе. Тот ли это случай, когда предпочтительно использовать мокинг вместо fake-классов?

Егор Бугаенко:

`ServletContext` — и вправду хороший пример, поскольку это монстрообразный интерфейс и поэтому потребует создания

монстрообразного fake-класса. Такого монстра необходимо создать однажды, и сделать это должны разработчики Servlet API. Такой класс, как FakeServletContext, должен поставляться вместе с ServletContext. Это сделало бы жизнь намного проще. К сожалению, они так не делают. Если mock-реализация контекста сервлета вам нужна во многих местах в приложении, создавайте fake-класс. Если только однажды — задействуйте Mockito.

2.9. Делайте интерфейсы краткими, используйте smart-классы

Обсуждение на http://goo.gl/1Zos9r.

Я уже упоминал в разделе 1.2, что качественно спроектированный целостный класс должен иметь всего несколько публичных методов. Мы обсудим это еще подробнее в разделе 3.1, но уверен, что вы уже понимаете важность поддержания небольших размеров классов. Еще важнее делать небольшими интерфейсы. Почему это приоритетнее? Потому что класс может реализовывать несколько интерфейсов. Если каждый из двух интерфейсов реализуют по пять методов, то класс, реализующий оба интерфейса, должен иметь десять методов. Такой класс элегантным не назовешь. Помните интерфейс Exchange из предыдущего раздела? Вот этот:

```
interface Exchange {
  float rate(String target);
  float rate(String source, String target);
}
```

Он хорош как объект обсуждения того раздела, но спроектирован отвратительно, поскольку *требует* слишком многого. Интерфейс — это контракт, который должен соблюдать реализующий его класс. Этот интерфейс возлагает слишком много обязанностей на реализующий его класс. Такой контракт способствует нарушению известного принципа единственной ответственности, иными словами, созданию рыхлых классов. Контракт требует от обменника вычислять курс и подставлять валюту по умолчанию, если она не была указана. Это две разные функции, хотя и очень близкие друг другу. Я веду к тому, что метода rate() с одним аргументом в этом интерфейсе быть не должно.

Должны ли мы определить для него еще один интерфейс? Нет. Мы должны создать smart-класс прямо внутри интерфейса:

```
interface Exchange {
  float rate(String source, String target);
  final class Smart {
    private final Exchange origin;
    public float toUsd(String source) {
      return this.origin.rate(source, "USD");
    }
  }
}
```

В этом smart-классе может быть намного больше методов, делающих нечто очевидное и очень общее. Данный smart-класс не знает, как реализован обменник и как вычисляется курс, но он реализует поверх этого некоторую функциональность. Эти возможности можно сделать общими для разных реализаций Exchange.

Вот еще одна причина создавать smart-классы и поставлять их вместе с интерфейсами: не хотелось бы, чтобы разные реализации интерфейса снова и снова переписывали одну и ту же функциональность. Загрузка обменных курсов с Нью-Йоркской фондовой биржи — уникальная функция класса NYSE, который реализует интерфейс Exchange. Но функционал, подставляющий валюту «доллар США» в случае, когда она не была указана, с легкостью может быть использован совместно несколькими классами.

Вот как будет применяться вложенный класс `Exchange.Smart` в сочетании с классом `NYSE`:

```
float rate = new Exchange.Smart(new NYSE())
  .toUsd("EUR");
```

Скажем, мы хотим добавить функциональности классу `NYSE` и в то же время другим реализациям интерфейса `Exchange`. Допустим, мы часто выполняем преобразование из долларов в евро и хотим избежать дублирования кода. И не хотим повсюду использовать строковый литерал `"EUR"`. Нам нужен метод наподобие `eurToUsd()`. Мы не будем добавлять его к интерфейсу `Exchange`. Вместо этого поместим его в smart-класс. Теперь в нем два метода:

```
interface Exchange {
  float rate(String source, String target);
  final class Smart {
    private final Exchange origin;
    public float toUsd(String source) {
      return this.origin.rate(source, "USD");
    }
    public float eurToUsd() {
      return this.toUsd("EUR");
    }
  }
}
```

Мы можем получить курс конверсии евро в доллары следующим образом:

```
float rate = new Exchange.Smart(new NYSE())
  .eurToUsd();
```

Smart-класс увеличивается в размерах, а интерфейс `Exchange` остается небольшим и целостным. В нем есть всего один метод, который реализован классами `NYSE`, `XE`, `Yahoo` и другими источниками информации о курсах обмена валюты. Функциональность smart-класса не специфична для конкретного обменника. Она является общей для всех обменников. Нет необходимости

требовать ее реализации от каждого обменника. Не нужно делать интерфейс `Exchange` слишком требовательным.

Вот почему заголовок и тема данного раздела — «Делайте интерфейсы краткими». Интерфейсы — контракты между нами, пользователями обменника, и программистами, реализующими класс `NYSE`. Чем больше интерфейс, тем более он требователен и тем больше проблем создаст тому, кто будет реализовывать класс `NYSE`. И не только потому, что его реализация потребует больших усилий. Дело в серьезной потере цельности и надежности класса. Предполагается, что класс `NYSE` будет выполнять некие сетевые вызовы к Нью-Йоркской фондовой бирже, и на этом все. Все остальные возможности — знать о валюте евро и конвертации в нее — могут быть реализованы кем-то другим. Этот кто-то — smart-класс — не должен ничего знать о сетевых вызовах. Мы, по сути, извлекаем общую функциональность и избегаем дублирования кода, делая интерфейсы краткими и поставляя с ними smart-классы.

Этот подход очень похож на компонуемые декораторы, рассматриваемые в разделе 3.2. Разница между декоратором и smart-классом в том, что smart-класс увеличивает количество методов объекта, а декоратор усиливает существующие методы. Рассмотрим следующий пример:

```
interface Exchange {
  float rate(String origin, String target);
  final class Fast implements Exchange {
    private final Exchange origin;
    @Override
    public float rate(String source, String target) {
      final float rate;
      if (source.equals(target)) {
        rate = 1.0f;
      } else {
        rate = this.origin.rate(source, target);
      }
      return rate;
```

```
    }
    public float toUsd(String source) {
      return this.origin.rate(source, "USD");
    }
  }
}
```

Вложенный класс `Exchange.Fast` одновременно является и декоратором, и smart-классом. Во-первых, он переопределяет метод `rate()`, тем самым усиливая его. Он пропустит обращение к сервису обмена валют, если валюты совпадают. Во-вторых, он добавляет новый метод `toUsd()`, который упрощает конверсию в доллары.

Bassspieler написал 9 февраля 2018 года:

> По поводу использования smart-классов: данные примеры не соответствуют элегантному принципу «не должно существовать публичных методов без аннотации `@Override`». Я понимаю стоящие за этим причины. Но, может быть, есть способ улучшить их? Или мы вынуждены смириться с этим? Я разрываюсь между теоретическими и практическими доводами.

Егор Бугаенко:

> Да, вы правы, эти smart-классы — неидеальные объекты. Скорее они являются дополнительными инструментами, помогающими создавать хорошие объекты. Поэтому данный принцип к ним неприменим.

Yev the dev написал 30 мая 2016 года:

> Почему бы вместо использования smart-классов не определить дополнительные методы как методы по умолчанию, которые применяли бы настоящие интерфейсные методы. Да, это противоречивая возможность, но она решает проблему открытости и, в отличие от smart-классов, позволяет программисту выполнить собственную реализацию дополнительных методов.

Егор Бугаенко:

> Методы по умолчанию — хорошая возможность, но я думаю, что smart-классы лучше. В основном потому, что у нас может быть несколько smart-классов, а методы по умолчанию должны оставаться в рамках одного интерфейса. Гибкость снижается.

David Raab написал 2 мая 2016 года:

> Хотел бы я посмотреть на ваше лицо в тот день в будущем, когда вы осознаете, что все ваши классы/интерфейсы с одним методом на самом деле являются статическими методами! Жду не дождусь!

Егор Бугаенко:

> Надеюсь, что в будущем статические методы исчезнут.

3 Работа

Главное различие между ООП и его процедурными предками в том, кто стоит у руля. В процедурном программировании заправлять будет код с операторами и инструкциями. Инструкции управляют и манипулируют данными, модифицируют и читают их. Данные — пассивный компонент, который спокойно сидит, ожидая, когда код считает или запишет его. Подпрограммы и структуры данных — два основных инструмента декомпозиции задачи на подзадачи.

ООП переворачивает все с ног на голову. В ООП управляют объекты — умные представители данных. Инструкции и операторы больше не у дел. По-хорошему, в идеально чистом ООП-языке их вообще не должно существовать. В нем не должно быть операторов — только классы и их экземпляры. В ООП мы компонуем меньшие объекты в больший, размером с приложение, объект и передаем ему управление.

Я понимаю, что все это может звучать очень абстрактно и теоретически, но уверяю вас, что это предельно практично. В нескольких последующих разделах я объясню и покажу на примерах, что я имею в виду. Короче говоря, данная глава посвящена аргументам против крупных объектов, статических объектов, `NULL`-ссылок, геттеров, сеттеров и оператора `new`.

3.1. Предоставляйте менее пяти публичных методов

Маленький объект — наиболее элегантный, сопровождаемый, цельный и верифицируемый объект. В разделе 2.6 я уже предложил ограничивать размер класса 250 строками, но это не самый важный показатель. У нас может быть класс из 50 строк и 20 методов. Это маленький класс? На самом деле нет. Как насчет другого примера: класс с одним публичным и 20 приватными методами? Это маленький класс? Не сказал бы, что он очень большой.

Поэтому в качестве главного показателя размера класса предлагаю использовать количество публичных (и защищенных) методов. Чем больше публичных методов, тем больше класс. Чем больше класс, тем слабее его сопровождаемость. На уме у меня число *пять*. Если в классе меньше пяти публичных методов, то это приемлемо. Если их больше, класс нуждается в рефакторинге. С ним что-то не так.

Обратите внимание на то, что я говорю о публичных методах, а не о конструкторах и приватных методах. Защищенные методы также попадают в эту категорию.

Тогда почему пять? Нет никакой особой причины — мне просто так кажется. Можем ли мы определить правильное количество? Не думаю. Должны ли принять число пять как абсолютную и непоколебимую константу? Нет. Это число нужно, чтобы помочь вам осознать, что есть верхняя граница количества методов, и она невелика. И это не десять, не двадцать, даже не семь методов. Оно очень невелико. Объявите несколько методов — и вот вы уже близки к пяти. Остановитесь и подумайте. Вы все еще пишете цельный, целостный класс, имеющий единственную область ответственности? Возможно, пришло время разделить его

на части. Я хочу, чтобы в промежутке между написанием четвертого метода и объявлением пятого вы остановились и подумали.

Что мы получим от того, что сделаем классы небольшими? Отвечаю: элегантность, сопровождаемость, целостность и верифицируемость.

Меньшие по размеру классы более *элегантны* просто потому, что в этом случае меньше вероятность сделать ошибку. Три метода согласовать между собой проще, чем десять. Они будут лучше сочетаться.

Меньшие классы лучше *сопровождаются*, потому что они... меньше по размеру. В них меньше кода, меньше методов, проще найти ошибку, их проще модифицировать. Проще изолировать проблему, когда в объекте минимум точек входа, при этом каждый метод есть точка входа в объект.

Меньшие классы более *цельные*, то есть их методы и свойства находятся, так сказать, ближе друг к другу. Проще говоря, каждый метод использует все свойства — вот суть цельности. Если одно свойство применяется только в двух методах, а другое — в трех других, мы можем с уверенностью сказать, что класс состоит из двух частей, едва связанных друг с другом. Цельность такого класса низкая. Если класс невелик, то повышается вероятность того, что все его методы будут взаимодействовать со всеми свойствами.

Меньшие классы более *верифицируемы*, так как проще воспроизвести все их сценарии использования. Прежде всего потому, что сценариев не так уж и много. Если у класса есть только один публичный метод, мы можем с легкостью написать все необходимые для них тесты. Если у класса десять методов, тесты либо будут слишком велики, либо вообще никогда не будут написаны.

Мне больше нечего сказать. Следите за количеством методов в классе и не позволяйте ему превысить число пять. Вот и все.

3.2. Не используйте статические методы

Обсуждение на http://goo.gl/8ql2ov.

Ах, статические методы... Одна из моих любимых тем. Мне понадобилось несколько лет, чтобы осознать, насколько важна эта проблема. Теперь я сожалею обо всем том времени, которое потратил на написание процедурного, а не объектно-ориентированного программного обеспечения. Я был слеп, но теперь прозрел. Статические методы — настолько же большая, если не еще бóльшая проблема в ООП, чем наличие константы `NULL`. Статических методов в принципе не должно было быть в Java, да и в других объектно-ориентированных языках, но, увы, они там есть. Мы не должны знать о таких вещах, как ключевое слово `static` в Java, но, увы, вынуждены. Я не знаю, кто именно привнес их в Java, но они — чистейшее зло. Статические методы, а не авторы этой возможности. Я надеюсь.

Посмотрим, что такое статические методы и почему мы до сих пор создаем их. Скажем, мне нужна функциональность загрузки веб-страницы посредством HTTP-запросов. Я создаю такой «класс»:

```
class WebPage {
  public static String read(String uri) {
    // выполнить HTTP-запрос
    // и конвертировать ответ в UTF8-строку
  }
}
```

Пользоваться им очень удобно:

```
String html = WebPage.read("http://www.java.com");
```

Метод `read()` относится к тому классу методов, против которого я выступаю. Предлагаю вместо этого использовать объект

(также я поменял имя метода в соответствии с рекомендациями из раздела 2.4):

```
class WebPage {
  private final String uri;
  public String content() {
    // выполнить HTTP-запрос
    // и конвертировать ответ в UTF8-строку
  }
}
```

Вот как им пользоваться:

```
String html = new WebPage("http://www.java.com")
  .content();
```

Вы можете сказать, что между ними нет особой разницы. Статические методы работают даже быстрее, потому что нам нет необходимости создавать новый объект каждый раз, когда нужно скачать веб-страницу. Просто вызовите статический метод, он сделает дело, вы получите результат и будете работать дальше. Нет необходимости возиться с объектами и сборщиком мусора. Кроме того, мы можем сгруппировать несколько статических методов в класс-утилиту и назвать его, скажем, `WebUtils`.

Эти методы помогут загружать веб-страницы, получать статистическую информацию, определять время отклика и т. п. В них будет много методов, а использовать их просто и интуитивно понятно. Кроме того, как применять статические методы, тоже интуитивно понятно. Все понимают, как они работают. Просто напишите `WebPage.read()`, и — вы догадались! — будет прочитана страница. Мы дали компьютеру инструкцию, и он ее выполняет. Просто и понятно, так ведь? А вот и нет!

Статические методы в любом контексте — безошибочный индикатор плохого программиста, понятия не имеющего об ООП. Для применения статических методов нет ни единого оправдания ни в одной ситуации. Забота о производительности не считается. Статические методы — издевательство над объектно-ориентированной парадигмой. Они существуют в Java, Ruby, C++, PHP и других языках. К несчастью. Мы не можем их оттуда выбросить, не можем переписать все библиотеки с открытым исходным кодом, полные статических методов, но можем прекратить использовать их в своем коде.

Мы должны прекратить применять статические методы.

Теперь посмотрим на них с нескольких разных позиций и обсудим их практические недостатки. Я могу заранее обобщить их для вас: статические методы ухудшают сопровождаемость программного обеспечения. Это не должно вас удивлять. Все сводится к сопровождаемости.

Объектное мышление против компьютерного

Изначально я назвал этот подраздел «Объектное мышление против процедурного», но потом переименовал. «Процедурное мышление» означает почти то же самое, но словосочетание «мыслить как компьютер» лучше описывает проблему. Мы унаследовали этот образ мышления из ранних языков программирования, таких как Assembly, C, COBOL, Basic, Pascal, и многих других. Основа парадигмы в том, что компьютер работает на нас, а мы указываем ему, что делать, давая ему явные инструкции, например:

```
  CMP AX, BX
  JNAE greater
  MOV CX, BX
  RET
greater:
  MOV CX, AX
  RET
```

Это ассемблерная «подпрограмма» для процессора Intel 8086. Она находит и возвращает большее из двух чисел. Мы помещаем их в регистры AX и BX соответственно, а результат попадает в регистр CX. Вот точно такой же код на языке C:

```
int max(int a, int b) {
  if (a > b) {
    return a;
  }
  return b;
}
```

«Что же с этим настолько не так?» — спросите вы. Ничего. Все с этим кодом в порядке — он работает, как и положено. Именно так работают все компьютеры. Они ожидают, что мы дадим им инструкции, которые они будут исполнять одну за другой. Многие годы мы писали программы именно так. Преимущество данного подхода в том, что мы остаемся вблизи процессора, направляя его дальнейшее движение. Мы у руля, а компьютер следует нашим инструкциям. Мы указываем компьютеру, как найти большее из двух чисел. Мы принимаем решения, он им следует. Поток исполнения всегда последователен, от начала сценария до его конца.

Такой *линейный* тип мышления называется «думать как компьютер». Компьютер в какой-то момент начнет исполнять инструкции и в какой-то момент закончит делать это. При написании кода на языке C мы вынуждены думать таким образом. Операторы, разделенные точками с запятыми, идут сверху вниз. Такой стиль унаследован из ассемблера.

Хотя языки более высокого уровня, чем ассемблер, имеют процедуры, подпрограммы и другие механизмы абстракции, они не устраняют последовательный образ мышления. Программа все равно проходится сверху вниз. В таком подходе нет ничего зазорного при написании небольших программ, но в более крупных масштабах так мыслить трудно.

Взглянем на тот же код, записанный на *функциональном* языке программирования Lisp:

```
(defun max (a b)
  (if (> a b) a b))
```

Можете ли вы сказать, где начинается и заканчивается исполнение этого кода? Нет. Мы не знаем, ни каким образом процессор получит результат, ни то, как конкретно будет работать функция `if`. Мы очень отстранены от процессора. Мы мыслим как функция, а не как компьютер. Когда нам нужна новая вещь, мы определяем ее:

```
(def x (max 5 9))
```

Мы *определяем*, а не даем инструкции процессору. Этой строчкой мы привязываем `x` к `(max 5 9)`. Мы не просим компьютер вычислить большее из двух чисел. Мы просто говорим, что `x` есть большее из двух чисел. Мы не управляем тем, как и когда это будет вычислено. Обратите внимание, это важно: `x` есть большее из чисел. Отношение «есть» («быть», «являться») — то, чем отличается функциональная, логическая и объектно-ориентированная парадигма программирования от процедурной.

При компьютерном образе мышления мы находимся у руля и контролируем поток исполнения инструкций. При объектно-ориентированном образе мышления мы просто определяем, кто есть кто, и пусть они взаимодействуют, когда это им понадобится. Вот как вычисление большего из двух чисел должно выглядеть в ООП:

```
class Max implements Number {
  private final Number a;
  private final Number b;
  public Max(Number left, Number right) {
    this.a = left;
    this.b = right;
  }
}
```

А так я буду его использовать:

```
Number x = new Max(5, 9);
```

Смотрите, я не вычисляю большее из двух чисел. Я определяю, что `x` есть большее из двух чисел. Меня не особо беспокоит, что находится внутри объекта класса `Max` и как именно он реализует интерфейс `Number`. Я не даю процессору инструкции относительно этого вычисления. Я просто инстанцирую объект. Это очень похоже на `def` в Lisp. В этом смысле ООП очень похоже на функциональное программирование.

Напротив, статические методы в ООП — то же самое, что подпрограммы в C или ассемблере. Они не имеют отношения к ООП и заставляют нас писать процедурный код в объектно-ориентированном синтаксисе. Вот код на Java:

```
int x = Math.max(5, 9);
```

Это совершенно неправильно и не должно использоваться в настоящем объектно-ориентированном проектировании.

Декларативный стиль против императивного

Императивное программирование «описывает вычисления в терминах операторов, изменяющих состояние программы». *Декларативное* программирование, с другой стороны, «выражает логику вычисления, не описывая поток его выполнения» (я цитирую «Википедию»). Об этом мы, по сути, говорили на протяжении нескольких предыдущих страниц. Императивное программирование похоже на то, что делают компьютеры, — последовательное выполнение инструкций. Декларативное программирование ближе к естественному образу мышления, в котором у нас есть сущности и отношения между ними. Очевидно, что декларативное программирование — более мощный подход, но императивный подход понятнее процедурным программистам. Почему декларативный подход более мощный? Не переключайтесь, и через несколько страниц мы доберемся до сути.

Какое отношение все это имеет к статическим методам? Неважно, статический это метод или объект, мы все еще должны где-то написать `if (a > b)`, так ведь? Да, именно так. Как статический метод, так и объект — всего лишь обертка над оператором `if`, который выполняет задачу сравнения `a` с `b`. Разница в том, как эта функциональность *используется* другими классами, объектами и методами. И это существенная разница. Рассмотрим ее на примере.

Скажем, у меня есть интервал, ограниченный двумя целыми числами, и целое число, которое должно в него попадать. Я должен убедиться, что это так. Вот что мне *придется* сделать, если метод `max()` — статический:

```
public static int between(int l, int r, int x) {
  return Math.min(Math.max(l, x), r);
}
```

Нужно создать еще один статический метод, `between()`, который использует два имеющихся статических метода, `Math.min()` и `Math.max()`. Есть только один способ это сделать — императивный подход, поскольку значение вычисляется сразу же. Когда я делаю вызов, я немедленно получаю результат:

```
int y = Math.between(5, 9, 13); // возвращает 9
```

Я получаю число `9` сразу же после вызова `between()`. Когда будет сделан вызов, мой процессор тут же начнет работать над этим вычислением. Это *императивный* подход. А как тогда выглядит декларативный подход?

Вот, взгляните:

```
class Between implements Number {
  private final Number num;
  Between(Number left, Number right, Number x) {
    this.num = new Min(new Max(left, x), right);
```

```
  }
  @Override
  public int intValue() {
    return this.num.intValue();
  }
}
```

Вот как я его буду использовать:

```
Number y = new Between(5, 9, 13); // еще не вычисляется!
```

Чувствуете разницу? Она чрезвычайно важна. Такой стиль будет *декларативным*, поскольку я не указываю процессору, что вычисления нужно выполнить сразу. Я просто определил, *что это такое*, и оставил на усмотрение пользователя решение о том, когда (и нужно ли вообще) вычислять переменную `y` методом `intValue()`. Может, она никогда не будет вычислена и мой процессор никогда не узнает, что это число `9`. Все, что я сделал, — объявил, что такое `y`. Просто объявил. Я еще не дал никакой работы процессору. Как указано в определении, выразил логику, не описывая процесс.

Я уже слышу: «О'кей, понял вас. Есть два подхода — декларативный и процедурный, но почему первый лучше второго?» Ранее я упомянул, что очевидно, что декларативный подход более мощный, но не объяснил почему. Теперь, когда мы рассмотрели оба подхода на примерах, обсудим преимущества декларативного подхода.

Во-первых, он быстрее. На первый взгляд он может показаться более медленным. Но если присмотреться внимательнее, станет видно, что на деле он *быстрее*, поскольку оптимизация производительности полностью в наших руках. Действительно, на создание экземпляра класса `Between` потребуется больше времени, чем на вызов статического метода `between()`, по крайней мере в большинстве языков программирования, доступных на момент написания этой книги. Я очень надеюсь на то, что в ближайшем будущем у нас появится язык, в котором инстанцирование объекта будет столь же быстрым, как и вызов метода. Но мы еще не пришли к нему. Вот почему декларативный подход медленнее... когда путь исполнения прост и прямолинеен.

Если речь идет о простом вызове статического метода, то он, безусловно, будет быстрее, нежели создание экземпляра объекта и вызов его методов. Но если у нас много статических методов, они будут последовательно вызываться при решении задачи, а не только для того, чтобы работать над действительно нужными нам результатами. Как насчет этого:

```
public void doIt() {
  int x = Math.between(5, 9, 13);
  if (/* Надо ли? */) {
    System.out.println("x=" + x);
  }
}
```

В данном примере мы вычисляем `x` вне зависимости от того, нужно нам его значение или нет. Процессор в обоих случаях найдет значение `9`. Будет ли следующий метод, использующий декларативный подход, работать так же быстро, как предыдущий?

```
public void doIt() {
  Integer x = new Between(5, 9, 13);
  if (/* Надо ли? */) {
    System.out.println("x=" + x);
  }
}
```

Я думаю, что декларативный код окажется быстрее. Он лучше оптимизирован. И не указывает процессору, что ему делать. Напротив, он позволяет процессору решить, *когда и где* действительно понадобится результат, — вычисления выполняются по требованию.

Суть в том, что декларативный подход быстрее, поскольку он *оптимален*. Это первый аргумент в пользу декларативного подхода по сравнению с императивным в объектно-ориентированном

программировании. Императивному стилю однозначно не место в ООП, и первая причина этого — оптимизация производительности. Не стоит говорить о том, что чем больше вы контролируете оптимизацию кода, тем более он сопровождаемый. Вместо того чтобы оставить оптимизацию процесса вычисления на откуп компилятору, виртуальной машине или процессору, мы делаем ее самостоятельно.

Второй аргумент — *полиморфизм*. Если говорить просто, то полиморфизм — это возможность разрывать зависимости между блоками кода. Допустим, я хочу поменять алгоритм определения того, попадает ли число в определенный интервал. Он довольно примитивен сам по себе, но я хочу его изменить. Я не хочу использовать классы `Max` и `Min`. А хочу, чтобы он выполнял сравнение с применением операторов `if-then-else`. Вот как сделать это декларативно:

```
class Between implements Number {
  private final Number num;
  Between(int left, int right, int x) {
    this(new Min(new Max(left, x), right));
  }
  Between(Number number) {
    this.num = number;
  }
}
```

Это тот же класс `Between`, что и в предыдущем примере, но с дополнительным конструктором. Теперь я могу использовать его с другим алгоритмом:

```
Integer x = new Between(
  new IntegerWithMyOwnAlgorithm(5, 9, 13)
);
```

Это, наверное, не лучший пример, поскольку класс `Between` очень примитивен, но, надеюсь, вы понимаете, о чем я. Класс `Between` очень просто отделить от классов `Max` и `Min`, поскольку *они являются классами*. В объектно-ориентированном программирова-

нии объект является полноправным гражданином, а статический метод — нет. Мы можем передать объект в качестве аргумента конструктору, но не можем сделать то же самое со статическим методом[1]. В ООП объекты связаны с объектами, общаются с объектами, обмениваются с ними данными. Чтобы полностью отвязать объект от остальных объектов, мы должны убедиться, что он не использует оператор `new` ни в одном из своих методов (см. раздел 3.6), а также в главном конструкторе.

Позвольте повторить: чтобы полностью отвязать объект от других объектов, вы всего лишь должны убедиться, что оператор `new` не применяется ни в одном из его методов, включая главный конструктор.

Можете ли вы проделать такую же отвязку и рефакторинг с императивным фрагментом кода?

```
int y = Math.between(5, 9, 13);
```

Нет, не можете. Статический метод `between()` использует два статических метода, `min()` и `max()`, и вы ничего не сможете сделать, пока не перепишете его полностью. А как вы сможете его переписать? Передадите четвертым параметром новый статический метод?

Насколько уродливо это будет выглядеть? Думаю, весьма.

Вот мой второй аргумент в пользу декларативного стиля программирования — он снижает сцепленность объектов и делает это очень элегантно. Не говоря уже о том, что меньшая сцепленность означает большую сопровождаемость.

---

[1] Мы, конечно, можем сделать это на разных языках, включая Java8, Ruby, PHP и Python, но такая возможность не имеет ничего общего с объектно-ориентированным программированием. Это суррогат процедурного и функционального программирования, который существует во всех популярных языках в силу его «удобства». На самом деле это только запутывает ситуацию. — *Примеч. авт.*

Третий довод в пользу превосходства декларативного подхода над императивным — декларативный подход говорит о результатах, а императивный объясняет единственный способ их получения. Второй подход намного менее интуитивно понятен, чем первый. Я должен сперва «выполнить» код в голове, чтобы понять, какого результата ожидать. Вот императивный подход:

```
Collection<Integer> evens = new LinkedList<>();
for (int number : numbers) {
  if (number % 2 == 0) {
    evens.add(number);
  }
}
```

Чтобы понять, что делает данный код, я должен пройти по нему, визуализировать этот цикл. По сути, я должен сделать то, что делает процессор, — пройтись по всему массиву чисел и поместить четные в новый список. Вот этот же алгоритм, записанный в декларативном стиле:

```
Collection<Integer> evens = new Filtered(
  numbers,
  new Predicate<Integer>() {
    @Override
    public boolean suitable(Integer number) {
      return number % 2 == 0;
    }
  }
);
```

Этот фрагмент кода намного ближе к английскому языку, чем предыдущий. Он читается следующим образом: «`evens` — это фильтрованная коллекция, включающая только те элементы, которые являются четными». Я не знаю, как именно класс `Filtered` создает коллекцию — использует ли он оператор `for` или что-то еще. Все, что я должен знать, читая этот код, — то, что коллекция была отфильтрована. Детали реализации скрыты, а поведение выражено.

Я осознаю, что некоторым читателям данной книги проще было воспринять первый фрагмент. Он немного короче и очень похож на то, что вы ежедневно видите в коде, с которым имеете дело. Я уверяю вас, что это дело привычки. Это обманчивое ощущение. Начните думать в терминах *объектов* и их *поведения*, а не *алгоритмов* и их *исполнения*, и вы приобретете истинное восприятие. Декларативный стиль непосредственно касается объектов и их поведения, а императивный — алгоритмов и их исполнения.

Если вы считаете этот код уродливым, попробуйте, например, Groovy:

```
def evens = new Filtered(
  numbers,
  { Integer number -> number % 2 == 0 }
);
```

Четвертый довод — *цельность* кода. Взгляните еще раз на предыдущие два фрагмента. Обратите внимание на то, что во втором фрагменте мы объявляем `evens` одним оператором — `evens = Filtered(…)`. Это значит, что все строки кода, ответственные за вычисление данной коллекции, находятся рядом друг с другом и не могут быть по ошибке разделены. Напротив, в первом фрагменте нет очевидной «склейки» строк. Можно с легкостью поменять их порядок по ошибке, и алгоритм сломается.

В таком простом фрагменте кода это небольшая проблема, поскольку алгоритм очевиден. Но если фрагмент императивного кода более крупный — скажем, 50 строк, может оказаться трудно понять, какие строки кода связаны друг с другом. Мы обсудили проблему темпорального сцепления чуть раньше — во время обсуждения неизменяемых объектов. Декларативный стиль программирования также помогает устранить это сцепление, благодаря чему улучшается сопровождаемость.

Вероятно, есть еще доводы, но я привел самые важные, с моей точки зрения, из относящихся к ООП. Надеюсь, я смог убедить вас в том, что декларативный стиль — это то, что надо. Некоторые из вас могут сказать: «Да, я понимаю, о чем вы. Я буду совмещать декларативный и императивный подходы там, где это уместно. Я буду использовать объекты там, где это имеет смысл, а статические методы — тогда, когда мне надо быстро сделать что-то несложное вроде вычисления большего из двух чисел». «Нет, вы неправы!» — отвечу вам я. Вы не должны их совмещать. Никогда не применяйте императивный стиль. Это не догма. У этого есть вполне прагматичное объяснение.

Императивный стиль нельзя совместить с декларативным чисто технически. Когда вы начинаете использовать императивный подход, вы обречены — постепенно весь ваш код станет императивным.

Допустим, у нас есть два статических метода — `max()` и `min()`. Они выполняют небольшие быстрые вычисления, поэтому мы делаем их статическими. Теперь нам нужно создать больший алгоритм, чтобы определить, принадлежит ли число интервалу. На сей раз мы хотим пойти декларативным путем — создать класс `Between`, а не статический метод `between()`. Можем ли мы так сделать? Наверное, да, но суррогатным способом, а не так, как положено. Мы не можем использовать конструкторы и инкапсуляцию. И вынуждены делать непосредственные, явные вызовы статических методов прямо внутри класса `Between`. Иными словами, мы не сможем написать чисто объектно-ориентированный код, если повторно применяемые компоненты представляют собой статические методы.

Статические методы напоминают *раковую болезнь* объектно-ориентированного ПО: однажды позволив им поселиться в коде, мы не сможем избавиться от них — их колония будет только расти. Просто обходите их стороной в принципе.

«Но они у меня повсюду! — воскликнете вы. — Что же делать?» Что я могу сказать... у вас проблемы, как и у всех нас. Существуют тысячи объектно-ориентированных библиотек, практически полностью состоящих из классов-утилит (мы обсудим их в следующем разделе). Здесь, как и с опухолью, лучшее средство — нож. Не используйте такие программы, если можете это себе позволить. Однако в большинстве случаев вы не сможете позволить себе воспользоваться ножом, поскольку эти библиотеки весьма популярны и предоставляют полезную функциональность. В данном случае лучшее, что вы можете сделать, — изолировать опухоль, создав собственные классы, которые оборачивают статические методы так, чтобы ваш код работал исключительно с объектами. К примеру, в библиотеке Apache Commons есть статический метод `FileUtils.readLines()`, который считывает все строки из текстового файла. Вот как мы можем превратить его в объект:

```
class FileLines implements Iterable<String> {
  private final File file;
  public Iterator<String> iterator() {
    return Arrays.asList(
      FileUtils.readLines(this.file)
    ).iterator();
  }
}
```

Теперь, чтобы прочесть все строки из текстового файла, наше приложение должно будет сделать следующее:

```
Iterable<String> lines = new FileLines(f);
```

Вызов статического метода произойдет только внутри класса `FileLines`, и со временем мы сможем от него избавиться. Либо этого не произойдет никогда. Но суть в том, что в нашем коде статические методы не будут вызываться нигде, за исключением одного места — класса `FileLines`. Так мы изолируем усопших, что позволяет нам разбираться с ними постепенно.

Классы-утилиты

Так называемые классы-утилиты на самом деле являются не классами, а лишь набором статических методов, используемых другими классами для удобства (они известны также как методы-помощники). К примеру, класс `java.lang.Math` — классический образец класса-утилиты. Такие порождения очень популярны в Java, Ruby и, к сожалению, почти во всех современных языках программирования. Почему они не являются классами? Потому что из них нельзя инстанцировать объекты. В разделе 1.1 мы обсудили разницу между объектом и классом и пришли к тому, что класс — это фабрика объектов. Класс-утилита не является фабрикой, например:

```
class Math {
  private Math() {
    // намеренно пустой
  }

  public static int max(int a, int b) {
    if (a < b) {
      return b;
    }
      return a;
  }
}
```

Хорошей практикой для тех, кто использует классы-утилиты, является создание приватного конструктора, как в примере, во избежание создания экземпляра класса. Поскольку конструктор приватный, никто, кроме методов класса, не может создать экземпляр класса.

Классы-утилиты — триумф процедурных программистов в области объектно-ориентированного программирования. Класс-утилита — не просто ужасная вещь вроде статического метода — это скопище ужасных вещей. Все плохие слова, сказанные о статических методах, могут быть повторены с многократным

усилением. Классы-утилиты — ужасный антипаттерн в ООП. Держитесь от них подальше.

Паттерн «Синглтон»

Паттерн «Синглтон» — популярный прием, претендующий на то, чтобы стать заменой статических методов. Действительно, в классе будет только один статический метод, а синглтон при этом будет выглядеть почти как настоящий объект. Однако он им не является:

```
class Math {
  private static Math INSTANCE = new Math();
  private Math() {}
  public static Math getInstance() {
    return Math.INSTANCE;
  }
  public int max(int a, int b) {
    if (a < b) {
      return b;
    }
    return a;
  }
}
```

Выше приведен типичный пример синглтона. Существует единственный экземпляр класса `Math`, который называется `INSTANCE`. Каждый может получить к нему доступ, просто вызвав `getInstance()`. Конструктор сделан приватным, чтобы предотвратить прямое инстанцирование объектов данного класса. Единственный способ получить доступ к `INSTANCE` — вызвать `getInstance()`.

«Синглтон» известен как паттерн проектирования, но в действительности это ужасный *антипаттерн*. Есть масса причин того, почему это плохой прием программирования. Я приведу лишь некоторые из них, касающиеся статических методов. Было бы, конечно, проще, если бы мы сначала обсудили то, чем синглтон

отличается от класса-утилиты, о котором мы только что говорили. Вот как выглядел бы класс-утилита Math, который делает то же, что и приведенный ранее синглтон:

```
class Math {
  private Math() {}
  public static int max(int a, int b) {
    if (a < b) {
      return b;
    }
    return a;
  }
}
```

Вот так будет использоваться метод max():

```
Math.max(5, 9); // класс-утилита
Math.getInstance().max(5, 9); // синглтон
```

В чем разница? Выглядит, будто вторая строка просто длиннее, а делает то же самое. Зачем было изобретать синглтон, если у нас уже были статические методы и классы-утилиты? Я часто задаю этот вопрос на собеседованиях с Java-программистами. Первое, что я обычно слышу в ответ: «Синглтон позволяет инкапсулировать состояние». Например:

```
class User {
  private static User INSTANCE = new User();
  private String name;
  private User() {}
  public static User getInstance() {
    return User.INSTANCE;
  }
  public String getName() {
    return this.name;
  }
  public String setName(String txt) {
    this.name = txt;
  }
}
```

Это ужасный фрагмент кода, но я вынужден привести его в качестве иллюстрации к своим доводам. Этот синглтон значит буквально «пользователь, в данный момент применяющий систему». Этот подход очень популярен во многих веб-фреймворках, где существуют синглтоны пользователей, веб-сессий и т. п. Итак, типичный ответ на мой вопрос о разнице между синглтоном и классом-утилитой: «Синглтон инкапсулирует состояние». Но это неверный ответ. Цель синглтона не в хранении состояния. Вот класс-утилита, который делает то же, что и упомянутый ранее синглтон:


```
class User {
  private static String name;
  private User() {}
  public static String getName() {
    return User.name;
  }
  public static String setName(String txt) {
    User.name = txt;
  }
}
```

Этот класс-утилита хранит состояние, и между ним и упомянутым синглтоном нет никакой разницы. Итак, в чем же проблема? И каков же правильный ответ? Единственно верный ответ состоит в том, что синглтон — это зависимость, которую можно разорвать, а класс-утилита — жестко запрограммированная тесная связь, которую разорвать невозможно. Иными словами, преимущество синглтонов в том, что в них можно добавить метод setInstance() наряду с getInstance(). Этот ответ верен, хотя я слышу его нечасто. Допустим, я использую синглтон следующим образом:

```
Math.getInstance().max(5, 9);
```

Мой код сцеплен с классом Math. Иными словами, класс Math — *зависимость*, на которую я полагаюсь. Без этого класса код не будет работать, и для его тестирования мне придется оставлять класс Math доступным, чтобы иметь возможность выполнять запросы. В случае с данным конкретным классом эта проблема невелика, поскольку он весьма примитивен. Однако

если синглтон большой, то мне, возможно, придется применять мокинг или заменять его чем-то, что лучше подходит для тестирования. Проще говоря, я не хочу, чтобы метод `Math.max()` выполнялся во время работы юнит-теста. Как мне это сделать? А вот как:

```
Math math = new FakeMath();
Math.setInstance(math);
```

Паттерн «Синглтон» обеспечивает возможность заменить инкапсулированный статический объект, что позволяет тестировать объект. Правда в следующем: синглтон намного лучше класса-утилиты только потому, что позволяет заменить инкапсулируемый объект. В классе-утилите нет объекта — мы не можем ничего изменить. Класс-утилита — неразрывная жестко запрограммированная зависимость — чистейшее зло в ООП.

Итак, о чем я? Синглтон лучше класса-утилиты, но все же является антипаттерном, причем довольно плохим. Почему? Потому, что логически и технически синглтон — *глобальная переменная*, ни больше, ни меньше. А в ООП нет *глобальной области видимости*. Поэтому глобальным переменным здесь не место. Вот программа на C, в которой переменная объявлена в глобальной области видимости:

```
#include <stdio>
int line = 0;
void echo(char* text) {
  printf("[%d] %s\n", ++line, text);
}
```

Всякий раз когда мы вызываем `echo()`, инкрементируется глобальная переменная `line`. Чисто технически переменная `line` видна из каждой функции и каждой строки кода в `*.c`-файле. Она видна *глобально*. Хвала разработчикам Java за то, что они не скопировали эту возможность из языка C. В Java, как и в Ruby и во многих других недо-ООП-языках, глобальные переменные запрещены. Почему? Потому что они не имеют никакого отношения к ООП. Это чисто процедурная возможность. Глобальные

переменные однозначно нарушают принцип инкапсуляции. Они просто ужасны. Надеюсь, мне больше не придется объяснять это в данной книге. Мне кажется очевидным, что глобальные переменные настолько же плохи, насколько плох оператор `GOTO`.

Однако, несмотря на все доводы против глобальных переменных, кто-то[1] нашел способ привнести их в Java, создав тем самым паттерн «Синглтон». Это попросту *издевательство* над принципами объектно-ориентированного проектирования, ставшее возможным благодаря наличию статических методов. Эти методы технически позволяют такое жульничество.

Никогда не используйте синглтоны. Даже не думайте.

«Чем их заменить? — спросите вы. — Если нам нужно, чтобы нечто было доступно многим классам в рамках всего программного продукта, что мы можем сделать?» Скажем, нам очень надо, чтобы большинство классов знало о том, какой пользователь в данный момент вошел в систему. У нас нет классов-утилит и синглтонов. Что у нас есть? Инкапсуляция!

Просто инкапсулируйте пользователя во все объекты, в которых он может пригодиться.

Все, что нужно вашему классу для работы, должно быть передано посредством конструктора и инкапсулировано внутри класса. Вот и все. Без исключения. Объект не должен затрагивать ничего, кроме своих инкапсулированных свойств. Вы можете сказать, что придется инкапсулировать слишком много: подключения к базам данных, вошедшего в систему пользователя, аргументы командной строки и т. п. Да, действительно, всего этого может оказаться слишком много, если класс чересчур большой и недостаточно цельный. Если вам нужно инкапсулировать слишком

[1] Я не знаю, чьих это рук дело, но синглтон описан в книге «Паттерны проектирования» «банды четырех» как паттерн проектирования. Я бы рекомендовал вам прочесть эту книгу, но со здоровой долей скептицизма.

много, переработайте класс — уменьшите его, о чем говорилось в разделе 2.1.

Но никогда не применяйте синглтон. Для этого правила нет исключений.

Функциональное программирование

Я часто слышу такой довод: если объекты небольшие и неизменяемые и при этом не задействуются статические методы, то почему бы не использовать функциональное программирование (ФП)? Действительно, если объекты элегантны настолько, насколько рекомендуется в данной книге, то они весьма похожи на функции. Итак, зачем нам нужны объекты? Почему бы просто не использовать Lisp, Clojure или Haskell вместо Java или C++?

Вот класс, представляющий алгоритм определения большего из двух чисел:

```
class Max implements Number {
  private final int a;
  private final int b;
  public Max(int left, int right) {
    this.a = left;
    this.b = right;
  }
  @Override
  public int intValue() {
    return this.a > this.b ? this.a : this.b;
  }
}
```

Вот как мы должны его применять:

```
Number x = new Max(5, 9);
```

А вот как мы задали бы в Lisp функцию, которая делала бы то же самое:

```
(defn max
   (a b)
   (if (> a b) a b))
```


Итак, зачем же использовать объекты? Код на Lisp намного короче.

ООП более выразительно и имеет большие возможности, поскольку оперирует объектами и методами, тогда как ФП — лишь функциями. В некоторых ФП-языках тоже есть объекты, но я бы назвал их ООП-языками с ФП-возможностями, а не наоборот. Я также считаю, что лямбда-выражения в Java, будучи подвижкой в сторону ФП, делают Java более рыхлым, сбивая нас с истинного ООП-пути. ФП — отличная парадигма, но ООП лучше. Особенно при правильном применении.

Мне кажется, в идеальном ООП-языке у нас были бы классы с функциями внутри. Не методы-микропроцедуры, как сейчас в Java, а настоящие (в смысле функциональной парадигмы) функции, имеющие единственную точку выхода. Это было бы идеально.

Компонуемые декораторы

Кажется, этот термин я придумал. Компонуемые декораторы — просто объекты-обертки над другими объектами. Они являются декораторами — известным паттерном объектно-ориентированного проектирования, — но становятся *компонуемыми*, когда мы объединяем их в многослойные структуры, к примеру:

```
names = new Sorted(
  new Unique(
    new Capitalized(
      new Replaced(
        new FileNames(
          new Directory(
            "/var/users/*.xml"
          )
        ),
        "([^.]+)\\.xml",
        "$1"
      )
    )
  )
);
```

Такой код, с моей точки зрения, выглядит очень чисто и объектно-ориентированно. Он исключительно декларативен, как объяснялось в разделе 3.2. Он ничего не делает, а лишь объявляет объект `names`, который *является* отсортированной коллекцией уникальных строк верхнего регистра, представляющих имена файлов в каталоге, измененных определенным регулярным выражением. Я просто объяснил, чем является этот объект, не говоря ни слова о том, как он устроен. Я просто *объявил* его.

Считаете ли вы этот код чистым и простым для понимания? Надеюсь, что да, с учетом всего того, о чем мы с вами говорили ранее.

Это то, что я называю компонуемыми декораторами. Классы `Directory`, `FileNames`, `Replaced`, `Capitalized`, `Unique` и `Sorted` — декораторы, поскольку их поведение полностью обусловлено инкапсулируемыми ими объектами. Они добавляют некоторое поведение инкапсулированным объектам. Их состояние совпадает с состоянием инкапсулированных объектов.

Иногда они предоставляют тот же интерфейс, что и инкапсулируемые ими объекты (но это не обязательно). К примеру, `Unique` — это `Iterable<String>`, также инкапсулирующий итератор по строкам. Однако `FileNames` — это итератор по строкам, инкапсулирующий итератор по файлам.

Большая часть кода в чистом объектно-ориентированном ПО должна быть похожа на приведенный ранее. Мы должны композировать декораторы друг в друга, и даже чуть более того. В какой-то момент мы вызываем `app.run()`, и вся пирамида объектов начинает реагировать. В коде совсем не должно быть процедурных операторов вроде `if`, `for`, `switch` и `while`. Звучит как утопия, но это не утопия.

Оператор `if` предоставляется языком Java и используется нами в процедурном ключе, оператор за оператором. Почему бы

не создать на замену Java язык, в котором был бы класс `If`? Тогда вместо следующего процедурного кода:

```
float rate;
if (client.age() > 65){
  rate = 2.5;
}
else {
  rate = 3.0;
}
```

мы бы писали такой объектно-ориентированный код:

```
float rate = new If(
  client.age() > 65,
  2.5, 3.0
);
```

А как насчет такого?

```
float rate = new If(
  new Greater(client.age(), 65),
  2.5, 3.0
);
```

И наконец, последнее улучшение:

```
float rate = new If(
  new GreaterThan(
    new AgeOf(client),
    65
  ),
  2.5, 3.0
);
```

Так выглядит чистый объектно-ориентированный и декларативный код. Он не делает ничего — просто объявляет, чем является `rate`.

С моей точки зрения, в чистом ООП не нужны операторы, унаследованные из процедурных языков вроде C. Не нужны `if`, `for`, `switch` и `while`. Нам нужны классы `If`, `For`, `Switch` и `While`. Чувствуете разницу?

Мы еще не дошли до таких языков, но рано или поздно обязательно дойдем. Я в этом уверен. А пока что старайтесь держаться подальше от длинных методов и сложных процедур. Проектируйте микроклассы так, чтобы они были компонуемыми. Убедитесь, что они могут повторно использоваться как элементы композиции в более крупных объектах.

Я бы сказал, что объектно-ориентированное программирование — это сборка крупных объектов из более мелких.

Какое отношение это имеет к статическим методам? Я уверен, вы уже поняли: статические методы не могут быть скомпонованы никоим образом. Они делают невозможным все то, о чем я говорил и что показывал ранее. Мы не можем собирать крупные объекты из более мелких с применением статических методов. Эти методы противоречат идее компоновки. Вот вам еще одна причина того, что статические методы — чистое зло.

В заключение: нигде и никогда не задействуйте в своем коде ключевое слово `static` — этим вы окажете себе и тем, кто будет использовать ваш код, большую услугу.

Matan Perelmuter написал 19 декабря 2017 года:

> Все это очень удобно, когда вы одна команда, работающая над одним проектом. Но что, если, к примеру, вы разрабатываете библиотеку для работы со строками, которая должна применяться в нескольких проектах? Другим разработчикам будет намного проще использовать API одного класса `StringUtils`, для которого есть автодополнение и всплывающая документация, поддержанная средой разработки, чем изучать все строковые классы. Даже если все они будут в одном пакете, то все равно теряется удобство автодополнения. Возьмем, к примеру, библиотеку cactoos. Мне кажется, разработчику намного проще изучить императивные библиотеки наподобие Apache Commons или Guava. Как бы вы рекомендовали публиковать программные интерфейсы библиотек?

Егор Бугаенко:

> Вы правы, современные среды разработки заточены под классы-утилиты, а не под объекты. Не могу порекомендовать никакой альтернативы. Возможно, нам стоит чаще использовать объекты и реже — статические методы, а там уже и средства разработки подтянутся.

Zack Macomber написал 26 августа 2016 года:

> В объектно-ориентированном мире нет данных — только объекты и их поведение! В Apache `FileUtils` нет данных (я не нашел ни одного нестатического поля в классе). Все, что делает этот класс, — отвечает на запросы клиентов и при этом не хранит состояния. Мне кажется, это подходит под описание класса, предоставляющего только поведение (функции). Каким образом использование оператора `new` лучше наличия публичных статических методов? Оператор `new` добавляет накладные расходы на создание объектов.

Егор Бугаенко:

> Словосочетание «нет данных» имеет в данном случае диаметрально противоположное значение! В `FileUtils` вы постоянно работаете с данными. Вы отвечаете за то, чтобы предоставлять и получать данные. Они всегда в ваших руках. При наличии настоящих объектов ситуация становится прямо противоположной. Вы не трогаете данные — вы просто общаетесь с объектами. Поэтому-то и нет данных.

Tor Djarv написал 1 июня 2016 года:

> Это, наверное, самое дурацкое применение ООП, которое я только видел. Слышали ли вы когда-нибудь старую поговорку «Каждому

делу — свой инструмент»? Вы, по сути, растащили вполне понятный код на набор классов (скорее всего, по разным файлам), что в итоге только добавляет сложности и делает простой код сложным для понимания, да еще и медленным — за счет создания одноразовых объектов. Объектная ориентация занимает свое место в мире программирования, но если вы используете ее только ради того, чтобы она была, то вы — плохой программист.

Bruno Martins написал 2 декабря 2014 года:

Егор, ваши статьи — восхитительное чтиво, я чувствую вашу приверженность ООП. Что касается темы: я понимаю, о чем вы, но пуританский взгляд на ООП затуманивает взгляд на другие важные аспекты. Есть причины того, что эта и другие проблемы, о которых вы пишете (почему плохо использовать NULL, объекты должны быть неизменяемыми), существуют и не имеют однозначного решения. Разработчики склонны уделять много внимания читабельности и паттернам проектирования, и из-за этого растет разрыв в понимании разницы между читабельностью кода человеком и машиной. Чистое ООП уверяет вас в том, что создание объектов для решения любых проблем — это хорошо. Это можно понять, потому что такой код лучше читается человеком. Но при этом мы упускаем из виду то, что у такого подхода существуют далеко идущие последствия (относительно памяти и производительности). Это также может вызвать проблемы, когда разработчику вдруг придется взаимодействовать с платформенно-зависимым кодом (или даже кодом с других платформ), например через JNI. Мне кажется, это должен учитывать каждый системный архитектор. Иметь красивый, чистый код, безусловно, хорошо, но пользователям в конечном итоге нужны надежные и эффективные системы. А это требует более глубоких размышлений, нежели использование лучших паттернов для создания самого читаемого кода.

Егор Бугаенко:

Спасибо за прочтение. Я понимаю ваши доводы, но позволю себе не согласиться. Мне кажется, что сейчас заботиться о памяти и процессорном времени намного менее важно, чем о читаемости и сопровождаемости. Почему? Потому что компьютеры дешевле программистов. Час моего времени, потраченного на разбор кода 2000-строчного класса, стоит больше, чем новая карта памяти для сервера. Поэтому мы должны задумываться о производительности только тогда, когда код стал понятным и чистым. Понимаете, о чем я?

Bruno Martins:

Действительно, программисты довольно часто приводят такие доводы. Но код, написанный в соответствии с современными стандартами программирования, не слишком сложен для понимания и сопровождения. Фактически любой средний Java-программист понимает проверки на NULL, классы-утилиты и изменяемые объекты, раз уж вы о них пишете. У всех них есть свои недостатки, но все в итоге сводится к контексту, с которым имеет дело программист. Они существуют потому, что ООП-языки создавались, чтобы обеспечить наибольшую гибкость. Кроме того, компенсировать низкую производительность программ более высокой производительностью машин, на мой взгляд, несколько недостойно (говорю как инженер, а не как управленец). Вы наверняка регулярно сталкиваетесь с низкой производительностью и неразумным управлением памятью в современных программных продуктах. И да, аппаратная часть сегодня все дешевле и все лучше, но и требования к программному обеспечению тоже повышаются. А как насчет программ и систем, в которых чрезвычайно важно эффективно использовать ресурсы, — прошивок для периферийных и носимых устройств, мобильных телефонов, игр, экспертных систем? Если все, что вам нужно, — простые чистые приложения, а ваши ресурсы неограниченны, то такой подход мне понятен. Но при

реализации решений, требующих производительности и надежности, необходимо понимать, что то, что вы пишете в коде, имеет глубокое влияние на то, что сгенерирует компилятор, и то, что по факту будет делать процессор. Безусловно, важно учить людей проектированию и реализации систем, учитывающих все эти соображения вне зависимости от того, можно ли им продешевить, а также написанию максимально читаемых, хорошо спроектированных программ. Но это не самое важное. Я восхищен вашей приверженностью вопросам проектирования и архитектуры ПО. Со многими вашими статьями я согласен. Но в некоторых случаях, мне кажется, вы упускаете из виду инженерию в прикладном и практическом аспекте, а не только в аспекте написания читаемого, красивого кода с применением (анти)паттернов.

3.3. Не допускайте аргументов со значением NULL

Обсуждение на http://goo.gl/TzrYbz.

NULL (также известный как null в Java, nil в Ruby, NULL в C++, None в Python и т. п.) — еще одна большая проблема в объектно-ориентированном мире наряду со статическими методами (см. раздел 3.2) и изменяемостью (см. раздел 2.6). По сути, вы делаете большую ошибку, если где-либо в своем коде используете константу NULL. Где бы то ни было — я серьезно. Здесь же поговорим о NULL как об аргументе метода. Затем в разделе 4.1 рассмотрим NULL как возвращаемый результат.

Посмотрим на следующий метод:

```
public Iterable<File> find(String mask) {
  // Обойти каталог
  // и найти все файлы, которые соответствуют
  // некоторой маске, например "*.txt".
  // Если маска == NULL, вернуть все файлы.
}
```


Весьма распространенный подход — разрешать пользователям передавать NULL как способ сказать: «У меня нет объекта, так что считайте, что он отсутствует». Действительно, он представляет собой удобную альтернативу этим двум методам:

```
public Iterable<File> findAll();
public Iterable<File> find(String mask);
```

Один метод выглядит более компактным и простым для запоминания пользователем, так ведь? Не нужно помнить, что, если я хочу отфильтровать файлы по маске, то надо вызывать find(), а если мне нужны все файлы, то findAll(). Если метода всего два, то их не так уж сложно запомнить. Но что, если у метода три аргумента, причем каждый из них может быть равен NULL? Мне придется создавать девять разных методов. Использование NULL кажется более удобным и компактным.

Звучит логично, но это противоречит объектно-ориентированной парадигме, где каждый объект полностью отвечает за свое поведение.

Чтобы реализовать метод find(), принимающий NULL в качестве аргумента, нам придется сделать что-то подобное следующему:

```
public Iterable<File> find(String mask) {
  if (mask == null) {
    // найти все файлы
  } else {
    // найти файлы по маске
  }
}
```

Дурным тоном здесь является сравнение mask==NULL. Вместо того чтобы *поговорить* с объектом mask, мы проходим мимо, *игнорируя* его. Мы спрашиваем его в лоб: «Стоит ли с тобой общаться?» Или даже: «Стоит ли с ним общаться?» Мы даже не обращаемся к объекту. Мы спрашиваем кого-то, кто должен знать, достоин объект общения или нет. Так общаться не очень-то вежливо, не правда ли?

Если мы *уважаем* объект, мы сделаем что-то вроде:

```
public Iterable<File> find(Mask mask) {
  if (mask.empty()) {
    // найти все файлы
  } else {
    // найти файлы по маске
  }
}
```

А еще лучше вот так:

```
public Iterable<File> find(Mask mask) {
  Collection<File> files = new LinkedList<>();
  for (File file : /* все файлы */)
    if (mask.matches(file)) {
      files.add(file);
    }
  }
  return files;
}
```

Если бы мы уважали объект `mask`, то позволили бы ему решить, есть ли у него для нас что-нибудь или же он пуст. Мы не должны судить о нем по его внешности. Не должны говорить, что если кто-то `NULL`, то он ненастоящий объект и мы не станем его использовать, а вот если он настоящий, тогда поговорим.

То, что мы принимаем `NULL` в качестве корректного аргумента, неизбежно вынуждает нас применять сравнение `mask==null`. Мы просто не можем поступить иначе. Всякий раз перед использованием объекта мы должны проверять его на «настоящесть». Выполняя такую проверку, мы снимаем с объекта значительную долю ответственности. Превращаем его в тупую структуру данных, которая неспособна позаботиться о себе и ожидает, что кто-то в нее что-то положит или из нее достанет.

В мире процедурного программирования, где подпрограммы манипулируют данными, факт существования `NULL` плох, но

по крайней мере хоть чем-то обусловлен. Я даю вам какие-то данные и не ожидаю, что вы будете с ними *общаться*. Они недостаточно умны, чтобы поддерживать разговор. Они просто биты и байты. Чисто технически, когда я *даю* вам данные, я на самом деле даю адрес, по которому вы можете их найти. Такой адрес, например `0x89f4a328`, называется *указателем*. Все байты в памяти пронумерованы, а это число является номером ячейки памяти, в которой содержится первый байт передаваемой структуры данных:

```
#include <stdio.h>
void foo(char* p) {
  printf("Пятый байт равен: %x", *(p + 5));
}
```

Подпрограмма `foo()` попросит процессор обратиться к этому адресу в памяти и прочитать пятый байт. Но мы можем договориться, что, когда я передаю вам число `0x00000000` в качестве адреса, *вы* не будете просить процессор обратиться по нему. Просто потому, что маловероятно, что там окажется моя структура данных. Впрочем, в современных компьютерных архитектурах она там не окажется никогда. Вот почему программисты много лет назад договорились, что, если указатель равен нулю, мы называем его `NULL` и никогда не используем как адрес в памяти. Мы не можем попросить процессор прочитать что-либо по этому адресу:

```
#include <stdio.h>
void foo(char* p) {
  if (p == 0) {
    printf("NULL — данных нет.");
  } else {
    printf("Пятый байт равен: %x", *(p + 5));
  }
}
```

Помните, что это всего лишь договоренность. Чисто технически нет никакой разницы между настоящим указателем `0x89f4a328`

и не очень настоящим `0x00000000`, который мы договорились называть `NULL`.

Что случится, если я забуду о давней договоренности и попрошу процессор считать данные по адресу `0x00000000`? В языке C результат непредсказуем[1], но в большинстве случаев процессор меня остановит и завершит выполнение процесса с сообщением «Ошибка сегментации». Так это работает в мире императивного процедурного программирования. Попробуйте сами:

```
#include <stdio.h>
int main(int argc, char** argv) {
  char* p = 0;
  printf("Байт по адресу 0 равен: %x", *p);
  // здесь программа упадет
}
```

К сожалению, объектно-ориентированный мир унаследовал эту «идею», даже притом что большинство современных языков не имеют указателей. В Java нет указателей, и нам нет необходимости их *разыменовывать*. Так называется конструкция `*p` из приведенного ранее примера. Указатель — всего лишь число, положение нужных мне данных в памяти. Чтобы сказать компилятору, что я хочу работать с данными, а не с адресом, я должен разыменовать указатель.

Хотя указатели считаются одной из болевых точек языка C, прежде всего из-за своей контринтуитивности, работать с ними проще, чем кажется. Довольно легко представить, что объекты разыменовываются автоматически, а не являются структурами данных, размещенными где-то в памяти. Если у нас есть объекты, но нет указателей, то зачем нам в Java нужен `null`? Честно говоря, не знаю. Кроме того, я думаю, что это *большая ошибка*

[1] Я не большой специалист в этом, но некоторые рецензенты говорили мне, что результат вполне предсказуем — исполнение программы остановится. — *Примеч. авт.*

разработчиков языка Java, как и Ruby, JavaScript и даже самых современных объектно-ориентированных языков.

«Что же делать, если нам нечего передавать в качестве аргумента метода `find()`? — спросите вы. — Что, если маски имени файла нет и мы просто хотим передать "ничего"? Почему бы не использовать `null`?»

В ООП проблема отсутствующего аргумента должна решаться с применением так называемого нулевого объекта. Вам нечего мне дать? Дайте мне объект, который ведет себя так, будто он пустой. Не перекладывайте проблему на мои плечи, не заставляйте меня проверять, дали вы мне объект или `NULL`. Вместо этого всегда передавайте мне объект, а в некоторых случаях — такой, который откажется со мной говорить, если я захочу от него слишком многого.

Скажем, у нас есть интерфейс `Mask`, который мы должны передавать методу `find()`, чтобы сообщить ему, какие файлы соответствуют маске, а какие — нет:

```
interface Mask {
  boolean matches(File file);
}
```

Надлежащая реализация такого интерфейса должна инкапсулировать glob-шаблон (например, `"*.txt"`) и сопоставлять с ним имена файлов. Напротив, нулевой объект будет выглядеть следующим образом:

```
class AnyFile implements Mask {
  @Override
  boolean matches(File file) {
    return true;
  }
}
```

Это граничный случай маски, не имеющий никакой логики. Он просто возвращает `true`, какое бы имя файла ему ни передали.

Теперь вместо того, чтобы передавать null как аргумент метода find(), мы просто создаем экземпляр класса AnyFile, и на этом все. Метод find() не будет иметь понятия о том, что происходит. Он все еще будет полагать, что ему передали корректную маску.

Договоримся о том, чтобы наши методы никогда не принимали NULL. Но что, если пользователи все равно передают NULL, несмотря на соглашение и документацию, гласящую: «Пожалуйста, не передавайте NULL»? Как реагировать на такое издевательское поведение? Есть два способа — оборонительный и игнорирующий. При оборонительном подходе мы проверяем аргумент на равенство NULL и бросаем исключение, если это так:

```
public Iterable<File> find(Mask mask) {
  if (mask == null) {
    throw new IllegalArgumentException(
      "Маска не может быть равна NULL, пожалуйста,
       передайте объект"
    );
  }
  // Найти файлы по маске и вернуть результат
}
```

Второй подход подразумевает игнорирование, и я склоняюсь к его использованию. Не делайте ничего, исходя из предположения, что аргумент не равен NULL. Рано или поздно, когда вы начнете манипулировать аргументом, будет выброшено исключение NullPointerException и вызывающая сторона осознает свою ошибку.

Не засоряйте код лишними проверками. NullPointerException — нормальный показатель того, что в качестве аргумента было некорректно передано значение NULL. Нет необходимости делать его более умным или более информативным. В качественно спроектированном ПО все равно не должно быть нулевых ссылок. Не защищайтесь, просто игнорируйте их — оставьте подобные ситуации на откуп JVM.

Вывод: никогда не принимайте NULL в качестве аргумента метода. Никаких исключений.

Никогда.

Kevin Rutherford написал 26 августа 2017 года:

> С моей точки зрения, NULL плох, поскольку он создает сцепление. Функция (ООП, ФП или любая другая), возвращая NULL, вынуждает кого-то выше по стеку проверять существование объекта. Такое сцепление связывает все объекты в стеке вызовов до тех пор, пока кто-то не проверит возвращаемое значение. Это говорит о зависимости по соглашению (все должны одинаково понимать семантику NULL в конкретном контексте), а часто еще и о зависимости по алгоритму (все должны возвращать NULL в одинаковых случаях). Поэтому между частями программы появляется дополнительная зависимость. А еще есть накладные расходы, связанные по меньшей мере с одним условным ветвлением. Это прибавляет работы тестировщикам и тем людям, которые будут читать этот код. Выработав привычку не возвращать NULL, вы сэкономите себе уйму времени впоследствии.

Igor спросил 13 октября 2016 года:

> Можете ли вы придумать сценарий, где использовать Optional было бы лучше, чем NULL или исключения?

Егор Бугаенко:

> Не думаю.

Madmenyo написал 8 апреля 2016 года:

> Вы говорите, что использование NULL засоряет код излишними условными операторами проверки. Но реализация вашим методом

приведет к засорению кода либо излишними строками вида `employee.isNobody()`, либо блоками `try/catch`. Я согласен с тем, что нужно прикладывать усилия к обеспечению читаемости кода.

Егор Бугаенко:

Проверки на `NULL` действительно засоряют код, поскольку они семантически не согласованы с остальным текстом программы. Блоки `try/catch` и метод `isNobody` (хотя не думаю, что я предлагал такое) семантически более близки к основной проблемной области.

Martin написал 15 октября 2014 года:

В некоторых языках есть третья и, пожалуй, лучшая альтернатива использованию исключений или нулевых ссылок — паттерн Optional. В стандартной библиотеке Java недавно появился параметрический класс `Optional` типа `T`, созданный по мотивам типа Optional в Scala (который, в свою очередь, был создан по подобию Maybe-типа из Haskell). Optional-типы лучше нулевых ссылок не только тем, что явно кодируют в системе типов, что функция может не вернуть значение, но и тем, что позволяют отложить вычисление значения.

Егор Бугаенко:

Хотя тип `Optional` выглядит удобным с точки зрения компьютерного мышления, он не имеет смысла с точки зрения объектно-ориентированного мышления. Как и в примере ранее, когда я звоню и спрашиваю Джефри, я не хочу говорить с «необязательным» Джефри. Это контринтуитивно. Я хочу поговорить с тем, кто мне может помочь, — либо с Джефри, либо с кем-то, кто представится Джефри (нулевой объект). Я не хочу спрашивать у того, кого услышал, есть ли у него внутри Джефри. Мыслите как объект, а не как программист, манипулирующий битами и байтами в компьютере.

Martin:

Мне кажется, что это наиболее интуитивный Option (без шуток). Вы спрашиваете Джефри, но нет гарантии, что вам его позовут. Говоря в терминах типов данных, вам нужно что-то, что отличает успешный вызов метода от неудачного. Если вы возвращаете нулевой объект, то фактически возвращаете пустую оболочку объекта Employee — вы считаете его Джефри, хотя он таковым не является. Только когда вы обследуете его, то обнаружите, что он не тот, о ком вы просили, — система типов не позволяет вам сделать такой вывод. Пользователь должен знать, на что посмотреть, — в данном случае на равенство его нулю. Тип возвращаемого значения `Optional<Employee>` дает знать всем пользователям API, что они могут и не получить экземпляр класса `Employee`, даже не глядя в исходный код или самописную документацию. Эта идея сильна сама по себе, но некоторые языки продвигают ее на шаг вперед (поскольку я привык к реализации `Optional` в Scala, пример будет основан на ней, но реализация в Java не должна сильно отличаться). В Scala вы будете безопасно работать с экземпляром `Optional`, не зная, вернул вызов существующий экземпляр `Employee` или нет. Делать if-проверки свойств Option-классов не принято — просто работайте с ними как с особыми списками из одного или нуля элементов. И только когда вам действительно нужен экземпляр `Employee`, вы материализуете объект вызовом `optionalEmployee.getOrElse(new Employee("Patrick Bateman"))`. Если исключения вам больше по душе, чем пустые пользователи, можно применить что-то вроде `optionalEmployee.getOrThrow(new EmployeeNotFoundException("No such employee found"))`. Заметьте, что все в руках пользователя API. Он, а не проектировщик решает, нужно ли материализовать или отбросить экземпляр `Employee`. Также невозможно преувеличить значение того, что пользователю не требуется заранее знать API, чтобы корректно получить доступ к объекту, так как состояние его существования перешло в систему типов.

Егор Бугаенко:

Понимаю ваши доводы — в них есть смысл. Позвольте все же еще раз попытаться убедить вас. Все, о чем вы говорили, весьма эффективно с точки зрения программиста, который хочет оставлять за собой управление возвращаемым ему объектом. Мне кажется, что при объектно-ориентированном программировании мы должны стремиться к чему-то противоположному. Мы должны избавить код, с которым работаем, от каких бы то ни было зависимостей настолько, насколько это возможно. Этого можно добиться при помощи абстрагирования.

Martin:

Но вы же явно просите Джефри. Не имеет смысла звать первого попавшегося работника или самозванца. Если метод гарантирует получение корректного объекта, нет нужды применять Option. В своем последнем примере вы на самом деле не используете паттерн «Нулевой объект», поскольку просто предполагаете, что объект корректен, и на основе этого делаете о нем выводы. Вернет ли метод `areYouHappy` `true\false` для нулевого объекта или бросит исключение? Почему в первом случае недоступность сотрудника интерпретируется как его удовлетворенность/неудовлетворенность жизнью? С другой стороны, бросая исключение из этого метода в нулевом объекте, вы фактически маскируете исключение `NullReferenceException` (`NullObjectEmployeeDoesntHaveFeelingsException`). Оба этих неудовлетворительных сценария отпадают при использовании Option. Это принуждает пользователя быть не столько ответственным, сколько честным и явно указывать то, что значение может отсутствовать. Если значение не может отсутствовать или если есть осмысленное значение по умолчанию, не возвращайте `Option`, подобно тому, как вы в данном случае не возвращали бы нулевой объект.

Roland Bouman написал 25 сентября 2014 года:

Мне не очень понятно, как паттерн «Нулевой объект» решает какие-либо проблемы в этом отношении. По крайней мере мне не удается придумать практический пример того, что можно было бы продолжать работать обычным образом в случае, когда возвращается особый нулевой объект. «Преимуществом» будет то, что не смогут появиться нулевые указатели, но при этом программа вынуждена будет делать бессмысленные операции (хотя, скорее, корректные операции над бессмысленным объектом). Конечно же, можно явно проверять, возвращен ли нулевой объект, но я не понимаю, почему это лучше, чем проверка на равенство `NULL`. Мне интересно, сможет ли кто-то привести реалистичный пример, демонстрирующий преимущества использования паттерна «Нулевой объект».

Егор Бугаенко:

Нулевые объекты не обязательно бросают исключения при каждом обращении к ним. Это может быть объект, который что-то может, а что-то не может. Мне не всегда нужно применять всю функциональность объекта.

3.4. Будьте лояльным и неизменяемым либо константным

Обсуждение на http://goo.gl/2UKLds.

Я уже исписал более десятка страниц на тему неизменяемости объектов в разделе 2.6, но пришло время вернуться к этой теме, прежде всего потому что существует связанное с ней большое заблуждение, которое надо попытаться развеять. Часто озвучиваемый довод против неизменяемости состоит в том, что мир по своей сути изменяем, и поэтому его невозможно представить с помощью только неизменяемых объектов. Действительно, у нас есть сущности, отвечающие за ввод-вывод, — файлы, потоки, веб-страницы, буферы и т. п. Все они, по сути, изменяемы, и ожидаемая их реализация также изменяема.

В сказанном есть изрядная доля здравого смысла, но я с этим не соглашусь. Да, мир, в котором мы живем, изменяем, но это

не значит, что мы не можем смоделировать его неизменяемыми объектами. Запутывает нас непонимание разницы между *состоянием* и *данными* — двумя разными вещами. Как обычно, начнем с примера:

```
class WebPage {
  private final URI uri;
  WebPage(URI path) {
    this.uri = path;
  }
  public String content() {
    // Делает HTTP GET-запрос, загружает веб-страницу
    // и конвертирует ее содержимое в UTF-8
  }
}
```

Как вы думаете, этот объект изменяемый или нет?

Вам он кажется изменяемым? Если да, то подумайте еще раз. Хотя метод `content()` может, по идее, возвращать разные значения при каждом вызове, сам объект *неизменяемый*. Он не меняет своего состояния в течение жизни, поэтому не имеет значения, как он себя ведет и что возвращают его методы. И это наверняка запутывает большинство из вас.

Интуитивно мы ожидаем, что неизменяемый объект будет вести себя как *константа*, возвращая одни и те же данные всякий раз, когда мы к нему обращаемся. Мы думаем, что если объект неизменяем, то он должен вести себя как строковый или числовой литерал. Действительно, большинство неизменяемых классов в Java и других языках ведут себя как константы. `String`, `URI` или `Double`, к примеру. Как только вы инстанцировали один из этих классов, объект будет предсказуем на 100 % и все его методы всегда станут возвращать одинаковые значения. Этого мы ждем от неизменяемых объектов, но ожидания не оправдываются. Не то чтобы совсем, но они формируют неполную картину. Это всего лишь граничный случай неизменяемости.

Неизменяемый объект подразумевает гораздо большее. Класс `WebPage` также неизменяем, хотя его метод `content()` всякий раз возвращает разные результаты. Мы не знаем, чего от него ожидать, поскольку он общается с сущностью реального мира — веб-страницей. То, что мы получим посредством HTTP-запроса, предсказанию не поддается. Вот почему класс `WebPage` не похож на класс `String`, хотя тоже является неизменяемым. Его поведение непредсказуемо, но объект все равно неизменяемый. Пускай объект не является константным, но он неизменяем, поскольку «верен» сущности, которую представляет.

Достаточно ли я вас запутал? Чтобы прояснить ситуацию, начнем сначала и определим, что такое состояние, а что такое объект. Потерпите немного. В этот раз я попробую изъясняться понятнее.

Объект — это *представитель* сущности реального мира, например файла на диске, веб-страницы, ассоциативного массива либо календаря на текущий месяц. Под реальным миром мы понимаем все то, что лежит за пределами области видимости объекта. К примеру, объект `f` представляет файл на диске:

```
public void echo() {
  File f = new File("/tmp/test.txt");
  System.out.println("Размер файла: %d", file.length());
}
```

Область видимости в данном случае определяется границами метода `echo()`. Чтобы пообщаться с файлом на диске и спросить, каков его размер, мы должны коммуницировать с объектом `f` посредством метода `length()`. Объект `f` — представитель файла `/tmp/test.txt`. Он представляет его интересы при взаимодействии с нами. Настолько, насколько это нас касается, в рамках метода `echo()` он *является* файлом.

Чтобы коммуницировать с файлом на диске, объект должен знать его *координаты*. Они еще называются *состоянием* объекта.

К примеру, состоянием объекта класса `WebPage` будет URI страницы. Чтобы загрузить его содержимое, объект свяжется с внешним миром посредством протокола HTTP, используя URI в качестве координат HTTP-службы. Состоянием класса `File` будет полный путь к файлу в файловой системе, например `/tmp/test.txt`.

У каждого объекта, по сути, есть три элемента: идентичность, состояние и поведение. Идентичность — то, что отличает `f` от других объектов, состояние — то, что `f` знает о файле на диске, а поведение — то, что `f` может сделать по нашему запросу. Основное различие между изменяемыми и неизменяемыми объектами состоит в том, что неизменяемые объекты не имеют идентичности и их состояние никогда не изменяется. Точнее, идентичность неизменяемого объекта совпадает с его состоянием.

Взгляните на класс `WebPage` еще раз. Если я инстанцирую два экземпляра с одним и тем же `uri`, будут ли они отличаться друг от друга? Будут ли они демонстрировать разное поведение? Нет. Они будут *идентичны*, поскольку их инкапсулированные состояния равны друг другу. Они оба представляют одну и ту же веб-страницу реального мира. Вот почему не будет никакой разницы в том, с которым из объектов я буду общаться, — они будут коммуницировать с той же веб-страницей. Координаты веб-страницы одинаковы, и поэтому объекты будут идентичными, хотя инстанцировались раздельно. Идеальная реализация класса как фабрики объектов (см. раздел 1.1) должна понимать это и избегать дублирующихся экземпляров, инкапсулирующих одинаковое состояние.

Однако в большинстве ООП-языков, включая Java, это не так. По умолчанию каждый объект имеет уникальную идентичность, которая может быть переопределена. К примеру, для класса `WebPage` я могу определить ее следующим образом (здесь приводится псевдореализация — настоящая реализация метода `equals()` несколько сложнее):


```
class WebPage {
  private final URI uri;
  WebPage (URI path) {
    this.uri = path;
  }
  @Override
  public void equals(Object obj) {
    return this.uri.equals(
      WebPage.class.cast(obj).uri
    );
  }
  @Override
  public int hashCode() {
    return this.uri.hashCode();
  }
}
```

Как видите, и метод `equals()`, и метод `hashCode()` рассчитывают на инкапсулированное свойство `uri`, что делает объекты класса `WebPage` прозрачными — они больше не имеют собственной идентичности. Они представляют веб-страницу, и единственное их состояние — координаты страницы в форме URI.

Но изменяемые объекты — совсем другая история. Они позволяют модифицировать свое состояние, что требует идентичности, отдельной от состояния. В настоящем объектно-ориентированном мире у нас были бы только неизменяемые объекты и нам не понадобились бы методы `equals()` и `hashCode()`. Они были бы одинаковыми во всех классах. Не было бы необходимости определять или переопределять их. В неизменяемом классе все объекты идентифицируются инкапсулированным ими состоянием. Состояние объекта является необходимым и достаточным для идентификации неизменяемого объекта.

Неизменяемый объект знает, где находится объект реального мира и как его использовать. Вот и все. Он знает координаты,

которые мы называем *состоянием*. Надеюсь, это логично, по крайней мере с точки зрения приведенного примера. Когда речь идет о веб-странице или файле, все просто, поскольку реальный мир вправду реален. Его сущности находятся за пределами нашего программного обеспечения. Вот почему несложно разделить сущность и ее представителя.

Иными словами, неизменяемый объект *верен* сущности реального мира, которую он представляет. Он никогда не меняет ее координаты. Он всегда работает с одной и той же сущностью, несмотря ни на что. Вот почему я говорю, что он *верен*. В то же время изменяемый объект может менять координаты сущности, с которой работает. Вот почему он неверен.

Что нам делать, если мы работаем с набором чисел? Задача тривиальна: нужен набор целых чисел, из которого можно удалять элементы, добавлять их, перебирать существующие элементы, пересчитывать и т. п. Как я могу реализовать все это, используя только неизменяемые объекты? Есть два возможных варианта: *константный* список либо *неизменяемый* список. Вот пример константного списка:

```
class ConstantList<T> {
  private final T[] array;
  ConstantList() {
    this(new T[0]);
  }
  private ConstantList(T[] numbers) {
    this.array = numbers;
  }
  ConstantList with(T number) {
    T[] nums = new T[this.array.length + 1];
    System.arraycopy(
      this.array, 0, nums,
      0, this.array.length
);
    nums[this.array.length] = number;
    return new ConstantList(nums);
  }
```



```
  Iterable<T> iterate() {
    return Arrays.asList(this.array);
  }
}
```

Вот как я буду его использовать:

```
ConstantList list = new ConstantList()
  .with(1)  // новый объект
  .with(15) // еще объект
  .with(5); // и еще один объект
```

Надеюсь, вы поняли, как это работает. При каждой попытке изменить список или добавить к нему новый элемент будет создаваться новый список, куда станут копироваться все элементы существующего.

Это классический неизменяемый объект, но я предлагаю называть его константным, потому что это всего лишь граничный случай неизменяемости, при котором его состояние равно сущности реального мира. Именно так, состояние `this.array` *совпадает* с сущностью, которую представляет объект `list`. Объект представляет массив, а его состояние *является* массивом. Сравните этот класс с классом `WebPage`, приведенным несколькими страницами ранее. В его случае `this.uri` — всего лишь координата сущности реального мира — веб-страницы. В то же время в `ConstantList` представляемая нами сущность и есть состояние.

Повторяю, это всего лишь граничный случай.

Вот как я бы делал список неизменяемым:

```
class ImmutableList<T> {
  private final List<T> items = LinkedList<T>();
  void add(T number) {
    this.items.add(number);
  }
  Iterable<T> iterate() {
    return Collections.unmodifiableList(this.items);
  }
}
```

Похож ли он, по-вашему, на неизменяемый? Похоже, что объекты данного класса можно модифицировать, поэтому они являются неизменяемыми? Нет, не совсем так. Попробуем проанализировать ситуацию. Модифицировать-то мы можем, правда, не сам объект. Взглянем на класс `WebPage` еще раз: вот что будет, если к нему добавить новый метод:

```
class WebPage {
  private final URI uri;
  WebPage(URI path) {
    this.uri = path;
  }
  public void modify(String content) {
    // Выполняет HTTP PUT-запрос и модифицирует
    // содержимое веб-страницы.
  }
}
```

Сделали ли мы его тем самым изменяемым? Определенно нет. Что происходит, когда мы используем его следующим образом?

```
WebPage page = new WebPage("http://localhost:8080");
page.modify("<html/>");
```

Изменяем ли мы состояние объекта `page`? Нет. Объект все еще неизменяемый? Несомненно. Неизменяема ли веб-страница, которую он представляет? Мы не знаем, но, скорее всего, нет.

Этот случай очень похож на то, что наблюдается в `ImmutableList`, но есть небольшое отличие — сущность реального мира находится *в памяти*, а не во Всемирной паутине. Если бы язык Java был спроектирован по-другому, мы бы никогда не увидели этой разницы. Если бы в Java был класс `Memory`, мы бы запрограммировали класс `ImmutableList` следующим образом:

```
class ImmutableList<Integer> {
  private final Memory total =
    new Memory(2);   // 2 байта в куче
  private final Memory items =
    new Memory(100); // 100 байт в куче
  void add(Integer number) {
    int pos = this.total.read();
    this.items.store(pos, number);
    this.total.store(pos + 1);
  }
}
```


Данный пример весьма примитивен, но, надеюсь, вы понимаете, что в нем происходит.

Что скажете теперь? Похож ли он на `WebPage`? Думаю, да. Инкапсулированные объекты `this.total` и `this.items` являются состоянием. Они представляют собой координаты нескольких байтов в памяти для счетчика элементов списка и еще какого-то количества байтов для хранения собственно элементов. По идее, и память, и диск, и Сеть для нас одинаковы. Наши объекты представляют их, и ничего более. Это очень похоже на указатель в C/C++. Вот как неизменяемый список выглядел бы в C++:

```
#include <stdlib.h>
class ImmutableList {
public:
  ImmutableList() :
    total((int*) calloc(1, sizeof(int))),
    items((int*) malloc(100)) { }
  ~ImmutableList() {
    free(total);
    free(items);
  }
  void add(int number) {
    int pos = *total;
    items[pos] = number;
  *total = pos + 1;
  }
private:
  int* const total;
  int* const items;
};
```

Обратите внимание на то, что указатели `total` и `items` являются константными. Они инициализируются в конструкторе путем выделения участков памяти и освобождаются, когда освобождаются соответствующие участки памяти.

Я считаю, что память должна рассматриваться нами так же, как диск, сеть или любое другое внешнее хранилище. Язык должен предоставлять встроенные инструменты для работы с памятью, но они должны быть намного более гибкими и функциональными, чем указатели в C/C++. Проблема указателей в том, что они чересчур просты. Они просто перенаправляют нас на некоторый участок памяти, а выделение памяти — наша проблема и забота. Как видно из приведенного примера, мы должны выделить с помощью функции `malloc()` фиксированное количество байтов. Что делать, когда весь выделенный блок заполнится элементами? Нужно увеличить емкость блока, но у нас это не получится. Мы должны выделить функцией `malloc()` новый блок памяти, скопировать туда содержимое существующего, а затем освободить его функцией `free()`.

Такая трехшаговая процедура должна быть реализована во встроенном классе `Memory`. К сожалению, такого класса в Java нет.

Блок памяти для нас — такой же внешний ресурс, как и файл на диске. С точки зрения архитектуры программы между ними нет абсолютно никакой разницы. Учитывая этот принцип, мы можем использовать неизменяемые объекты где угодно. Некоторые из них будут константными, некоторые — неизменяемыми, представляющими фрагменты памяти.

Очевидно, что лучше использовать константные объекты, поскольку они проще для проектирования, поддержки и понимания. Почти все, что говорилось о неизменяемых объектах в разделе 2.6, относилось к константным объектам, являющимся частными случаями неизменяемых.

Таким образом, любая система, независимо от ее производственной и технической области применения, включая игры, настольные приложения, мобильные приложения, веб-приложения, корпоративные системы и т. п., может и должна быть реализована целиком из неизменяемых объектов.

Jacob Zimmerman написал 18 марта 2017 года:

> Вы правы во всем, но я понимаю, почему люди говорят то, что говорят о неизменности результатов, возвращаемых методами. Они приравнивают неизменяемость к идеалам функционального программирования, включая идемпотентность — принцип, согласно которому вызовы метода с одними и теми же аргументами должны возвращать одинаковые результаты. К этому идеалу нужно по возможности стремиться, но, когда объект представляет нечто вроде файла, это, очевидно, невозможно.

Егор Бугаенко:

> Вот именно!

Ben Nadel написал 4 июня 2016 года:

> Я сейчас читаю «Элегантные объекты», в частности раздел о неизменяемых данных, и у меня, как и у многих, появляется масса вопросов и непонятных моментов. Имея опыт веб-разработки, я часто думаю в терминах объектной модели документа (Document Object Model, DOM) и обрабатываю события в DOM по мере того, как они поднимаются вверх по дереву элементов. Для взаимодействия с DOM объект события имеет методы, изменяющие его (события) поведение, `event.stopPropagation()` и `event.preventDefault()`. Интересно, противоречит ли написанное выше принципу неизменяемости? Как сказано в посте: «Объект является неизменяемым тогда и только тогда, когда он не меняет координаты сущности реального мира, которую он представляет». Из этого я понял, что если тип события (например, mousemove) и его цель (например, Element) не меняются, то не меняется и идентичность события. Тот факт, что другие особенности распространения и поведения события меняются, как таковой не делает класс изменяемым. Иными словами, могу ли я с учетом сказанного считать объект события неизменяемым?

Егор Бугаенко:

Хороший вопрос (спасибо, кстати, что купили и читаете книгу). Действительно, объект события является неизменяемым до тех пор, пока не меняет свою идентичность. Кроме того, я думаю, что архитектура DOM в целом не является объектно-ориентированной. Они на деле являются не объектами, а структурами данных, сцепленными друг с другом. Функциональность отделена от них и вызывается посредством механизма событий. Настоящая объектная модель должна выглядеть по-другому. Никаких событий быть не должно.

Ben Nadel:

Я все еще пытаюсь разобраться в этом всем, но не могу представить себе веб-разработку без использования событий.

Егор Бугаенко:

Вы хотите сказать: JavaScript-разработку?

Ben Nadel:

Да, именно. Но опять же я с уверенностью отношу себя к «процедурному» лагерю. Поэтому и купил вашу книгу.

Егор Бугаенко:

Да, DOM в целом проектировался с учетом процедурного подхода. У нас есть набор объектов (структур данных), сцепленных друг с другом и встроенных друг в друга. Еще есть некоторый интерпретатор (браузер), связанный с мышью и клавиатурой и имеющий полный доступ к этим структурам данных. А еще есть написанные на JavaScript процедуры, которые по мере необходимости вызываются интерпретатором и имеют доступ ко всему дереву данных. Очевидно, что ООП не должно так работать. Я не в состоянии с ходу сказать, какая архитектура была бы корректной, но интерпретатор и JavaScript-процедуры не должны находиться отдельно от дерева.

Mario T. Lanza написал 4 июня 2016 года:

Вы называете неизменяемым объект, чья внешняя оболочка неизменяема, но один из важнейших механизмов неизменяемости — то, каким образом объект работает с чистыми функциями. Если я передаю один из неизменяемых объектов в чистую функцию, которая дает некоторый результат, я ожидаю этого же результата при каждом вызове. То же верно и для методов. Однако, поскольку неизменяемый объект может манипулировать состоянием, где бы оно ни находилось, мы не можем ожидать этого. Потеря этой гарантии крайне существенна. Я не думаю, что люди возражают по поводу того, что вы не понимаете, что после того, как класс помечен final, у вас появляется неизменяемая оболочка. Я думаю, они имеют в виду, что называть неизменяемую оболочку неизменяемым объектом — значит рушить доводы в тех обсуждениях, где неизменяемость связана с определенными гарантиями. Ваши посты часто великолепны, ваши идеи достойны распространения, но, имея опыт функционального программирования, я не вижу ценности в неизменяемом объекте, чье видимое поведение не связано напрямую с некоторым неизменным состоянием.

Егор Бугаенко:

В общем-то вы правы, но я предлагаю переосмыслить традиционную интерпретацию неизменяемости в ООП. Она может выглядеть противоречиво, но я считаю, что она осмысленна. Как минимум с моей точки зрения.

Martin написал 22 декабря 2014 года:

Вы по-прежнему не правы. «Это живой организм, представляющий сущность реального мира в некоторой окружающей среде (компьютерной программе)». Как же это живые организмы

неизменяемы? Проблема не в том, что люди вас не понимают. С неизменяемостью все в порядке. Мы все понимаем. Проблема в том, что вы утверждаете, что все объекты всегда должны быть неизменяемыми. Все. Всегда. Точка. А тот, кто не согласен, просто не понял вас. Ваше утверждение о том, что объекты есть сущности реального мира, напрямую противоречит утверждению о том, что они не изменяются. Почему? Потому что сущности реального мира постоянно меняются. Возможно, вы хотели сказать, что объект является представлением сущности реального мира, что ближе к истине. Но, согласно такому определению, объект может и должен меняться. Возможно, вы имели в виду, что объект является представлением состояния сущности реального мира в данный конкретный момент времени. А вот это уже закрывает проблему с неизменяемостью. О неизменяемом объекте имеет смысл говорить тогда и только тогда, когда он представляет собой нечто неизменяемое (например, состояние объекта в прошлом, каким бы недалеким оно ни было). Недостаток вашего примера с классом `File` (как и примера с собакой, используемого в других постах) в том, что объект класса `File` не является представителем собственно файла. Он представляет ссылку на файл (она, в свою очередь, может быть изменяемой, но для данного обсуждения представим, что не может). Если он действительно был представителем файла, а не ссылки на него, то его состоянием было бы не имя файла — это были бы байты, находящиеся в файловой системе, права доступа, дата создания, дата изменения, информация об аудите и все то, что мы считаем относящимся к файлу. Позвольте выразиться по-другому. В Java у вас может быть два объекта класса `File`, ссылающихся на один и тот же файл в файловой системе. Но это проблема с реализацией. Разработчик класса `File` говорит, что объекты класса представляют собой ссылки на файлы в файловой системе. В объектно-ориентированной архитектуре у вас вполне могли бы быть объект класса `ActualFile` и несколько объектов класса `File`, выступающих ссылками на него. `ActualFile` может и наверняка должен быть неизменяемым. Вы можете добавлять содержимое в `ActualFile`, не создавая нового экземпляра. Очевидно, что изменение `ActualFile`, на который ссылаются несколько экземпляров класса `File`, никоим образом не меняет их внутреннего состояния. То, что вы можете запрограммировать класс `ActualFile` как неизменяемый, совершенно не значит, что вы должны так делать. Есть разница между объектно-ориентированным программированием и архитектурой. В рамках объектно-ориентированной архитектуры вы можете и должны иметь неизменяемые объекты.

Егор Бугаенко:

Позвольте объяснить. Во-первых, я считаю, что объект действительно является живым организмом, живущим в некоторой среде обитания, например в методе. Во-вторых, он представляет сущность реального мира, которая находится где-то вне его среды обитания — неважно, в другом ли методе, на диске, или в Буэнос-Айресе. В-третьих, неизменяемый объект — бескорыстный представитель сущности реального мира, у него нет никаких личных вещей. Все, что у него есть, — набор координат той сущности, которую он представляет. И наконец, бескорыстие объекта не означает его глупость — он прекрасно может передавать наши запросы сущности реального мира, а ее ответы — нам. Но себе он ничего не оставляет, хотя все проходит через его руки. Он не жадный.

3.5. Никогда не используйте геттеры и сеттеры

Обсуждение на http://goo.gl/LSyvo9.

Геттеры и сеттеры. Я не знаю, паттерн это или просто договоренность. Я думаю, вы знаете, о чем я, но все же позвольте напомнить. Вот как они выглядят:

```
class Cash {
  private int dollars;
  public int getDollars() {
```

```
    return this.dollars;
  }
  public void setDollars(int value) {
    this.dollars = value;
  }
}
```

Итак, что у нас здесь? Изменяемый класс с единственным приватным свойством, доступным через *геттер* `getDollars()` и изменяемым с помощью *сеттера* `setDollars()`. В разделе 2.6 мы уже говорили, что все классы должны быть неизменяемыми. Этот же — изменяемый. Кроме того, в разделе 2.4 мы обсуждали, как должны называться методы. В этом классе два метода названы некорректно. А еще у него нет ни одного конструктора, что противоречит принципам, озвученным в разделе 2.1. Я к тому, что этот класс уже противоречит советам, приведенным в данной книге.

Но это еще не все. Изменяемость, названия методов и полное отсутствие конструкторов — лишь малые прегрешения по сравнению с намного большим грехом, в котором повинен этот класс. Это не класс, а *структура данных*. И этот грех не может быть прощен. Аминь.

Объекты против структур данных

Какая разница между объектом и структурой данных? Почему быть структурой данных — грех в ООП.

Сначала обсудим их разницу. Вот структура данных, описанная на C:

```
struct Cash {
  int dollars;
}
```

А вот похожая вещь — объект, описанный на C++:

```
#include <string>
class Cash {
```



```
public:
  Cash(int v): dollars(v) {};
  std::string print() const;
private:
  int dollars;
};
```

В чем разница? Давайте взглянем. Так мы используем структуру данных `cash` в языке C:

```
printf("Cash value is %d", cash.dollars);
```

А так делаем нечто подобное с объектом класса `Cash` в C++:

```
printf("Cash value is %s", cash.print());
```

Чувствуете разницу? Работая со *структурой*, мы получаем доступ к ее полю `dollars` и работаем с ним как с целым числом. С самой же структурой ничего не делаем. Не общаемся с ней. Мы напрямую получаем доступ к какому-то ее полю. Структура для нас всего лишь *мешок с данными*, не имеющий никакой индивидуальности.

Класс — это нечто другое. Он не позволяет получать доступ к своим полям. Кроме того, он их нам даже не показывает. Мы даже не знаем, что внутри него есть поле `dollars`. Все, о чем мы можем его попросить, — *вывести* себя на экран. Мы понятия не имеем, как это происходит. Будут ли как-то использоваться инкапсулированные поля? Неизвестно. Это называется *инкапсуляцией*, и в этом суть ООП.

Структуры данных прозрачны, а объекты — нет. Структуры данных — прозрачные ящики, а объекты — черные ящики. Структуры данных пассивны, объекты активны. Структуры данных мертвы, а объекты живы. Хорошие слоганы, не так ли? Здесь я хотел бы остановиться и продолжить разговор о геттерах и сеттерах в предположении, что объекты лучше структур данных. Этот факт очевиден всем. Однако я взял паузу и задумался о том, что не так со структурами данных. Почему мы не можем

сочетать объекты и структуры данных? Да, пускай объекты лучше, но почему только они? Ведь иногда нам нужны старые добрые структуры данных с парочкой полей. Зачем строить объект с поведением, состоянием и идентичностью? Мы же не ООП-фанатики, правда?

Нет, конечно, но хотелось бы работать только с объектами, а не со структурами данных, чему есть весьма рациональное и практичное объяснение.

Как обычно, все упирается в сопровождаемость. Главная цель любой парадигмы программирования, какой бы она ни была — процедурной, функциональной или объектно-ориентированной, — *упрощать* вещи, *сужая* область видимости. Чем меньше область, которую вам надо понимать в каждый конкретный момент времени, тем проще понимать, модифицировать и сопровождать программное обеспечение.

Когда в процедурном и императивном программировании код манипулирует данными, лучший способ упростить вещи — использовать подпрограммы и агрегацию данных. Вместо того чтобы продираться сквозь тысячи операторов, мы откладываем некоторые из них в сторонку и называем их подпрограммой. Вместо того чтобы управлять сотнями байт, мы группируем их в массивы и структуры данных и ссылаемся на них с помощью единственного указателя.

Группа находящихся рядом байтов удобна тем, что, когда мы хотим адресовать ее элемент, мы добавляем его смещение относительно начала группы к адресу этого начала. Такую группу проще передать в качестве аргумента к подпрограмме. Вместо того чтобы передавать десять аргументов структуры, мы передаем указатель на нее, а подпрограмма с легкостью находит нужные ей байты.

Движущими силами в данном случае являются код, подпрограммы и инструкции процессора. Они манипулируют данными,

а данные просто сидят и ждут, пока их кто-нибудь не изменит или не прочитает.

В разделе 3.2 мы обсуждали разницу между процедурным и объектно-ориентированным программированием и пришли к выводу, что ООП было изобретено в первую очередь для того, чтобы упростить вещи по сравнению с процедурным миром. Объекты перевернули все *с ног на голову*. Код стал пассивным, а данные — активными. Если я правильно понимаю ООП, то в этом его суть. Данные больше не сидят и ничего не ждут. Теперь они инкапсулированы внутрь живых объектов. Они связаны друг с другом и, когда приходит время что-либо сделать, инициируют исполнение посредством сообщений, известных также как вызовы методов. В ООП код не преобладает над данными. Вместо этого объекты инициируют исполнение кода при необходимости. Звучит слишком абстрактно, но лучше я объяснить не смогу. Важно понимать фундаментальное различие между процедурным и объектно-ориентированным стилями программирования. Код больше не рулит. В ООП код вторичен. Объекты — полноправные граждане кода, а программное обеспечение *есть* их *инициализация* посредством конструкторов.

Ни операторы, ни выражения — конструкторы.

Каждый раз, когда мы пытаемся применить в ООП что-то сложнее, чем байт, мы делаем шаг назад к процедурному программированию. Когда мы группируем несколько байтов в структуру данных и начинаем использовать ее для коммуникации между объектами, мы серьезно подрываем объектную модель приложения, и назад дороги (почти) нет. Мы начинаем думать в терминах выражений и операторов, а не объектов и конструкторов. В разделе 3.2 мы уже обсудили разницу между императивным и декларативным стилями. Пришло время к ней вернуться. Когда данные становятся сложнее одного байта, мы возвращаемся к императивному программированию. Мы просто должны

писать инструкции и операторы, которые будут манипулировать байтами, — сами манипуляции неизбежно окажутся императивными.

Чтобы оставаться декларативными и объектно-ориентированными, мы должны прятать данные в объектах и никогда не выставлять их наружу. Только объект должен знать, что именно инкапсулировано и насколько сложна структура данных. Я бы даже сказал, что мы не должны оставлять данные *голыми*. Мы всегда должны их как следует одевать.

Никто не должен видеть их голыми или трогать их.

Голые данные склоняют нас к применению процедурного стиля программирования, которого в ООП следует избегать любой ценой, — таково прагматическое обоснование использования объектов вместо структур данных.

Благими намерениями вымощена дорога в ад

Геттеры и сеттеры были созданы, чтобы *нарушать* принцип инкапсуляции, хотя обычно декларируется обратное.

В Java они были введены, чтобы превращать классы в структуры данных, поскольку там структур данных не создано умышленно. В C++ есть структуры, поэтому геттеры и сеттеры в нем не требуются. В Java они нужны, чтобы создавать объекты, которые выглядят как объекты, но на деле являются пассивными структурами данных, подобно `struct` в C++.

Мы можем превратить класс в структуру данных, сделав его поля публичными (`public` в Java):

```
class Cash {
  public int dollars;
}
```


Однако это нарушает базовые правила программирования на Java так сильно, что любой скажет вам, что вы понятия не имеете об ООП. Так вот, чтобы избежать такого публичного унижения, мы договорились делать поля приватными и прикреплять к ним геттеры и сеттеры. В каждой современной среде разработки есть возможность генерировать геттеры и сеттеры к существующим приватным полям. Вы просто ставите курсор на поле класса, нажимаете кнопку и получаете два новых метода: один с префиксом `get`, другой — с префиксом `set`.

В Ruby есть встроенная возможность автоматически создавать геттеры и сеттеры. Они называются аксессорами и мутаторами. О том, что они нам нужны, говорят два ключевых слова — `attr_reader` и `attr_writer`:

```
class Cash
  attr_reader :dollars
  attr_writer :dollars
end
```

Это всего лишь удобная замена следующей развернутой конструкции:

```
class Cash
  def dollars
    @dollars
  end
  def dollars=(value)
    @dollars = value
  end
end
```

Те, кто проектирует языки и среды разработки, подталкивают нас к обертыванию приватных полей в геттеры и сеттеры.

Я считаю, что геттеры и сеттеры — удобный инструмент нарушения приниципа инкапсуляции в ООП. Они выглядят как методы, но в действительности маскируют тот неприятный факт, что мы получаем прямой доступ к данным. Данные *обнажены*.

Вы можете возразить, что данные скрыты, поскольку геттеры и сеттеры являются методами. Можно добавлять в них дополнительную логику, проверять данные на корректность и даже изменять способ хранения и считывания данных, но все это не имеет значения. С точки зрения пользователя объекта геттеры и сеттеры выглядят точно так же, как точки доступа к данным. Объект выглядит как структура данных с битами и байтами. Независимо от способа реализации геттеры и сеттеры являются данными и представляют данные, а не поведение.

Все дело в префиксах

Важно упомянуть, что порочной составляющей в антипаттерне «Геттер — сеттер» являются префиксы `get` и `set`. Они четко дают нам знать, что объект на самом деле не объект, а структура данных, не ожидающая к себе никакого уважения. Она ожидает, что мы будем обращаться с ней как с набором байтов, голыми данными. Она не хочет, чтобы с ней общались. А хочет, чтобы мы ввели в нее или получили из нее какие-то данные.

Вполне нормально иметь метод, возвращающий некоторые данные, например:

```
class Cash {
  private final int value;
  public int dollars() {
    return this.value;
  }
}
```

Но такое имя недопустимо:

```
class Cash {
  private final int value;
  public int getDollars() {
    return this.value;
  }
}
```


Не слишком ли я зациклился на именовании? Вовсе нет. Разница существенна и очень важна. Вызывая `getDollars()`, мы говорим: «Залезь в свои данные, найди там поле dollars и верни его значение». Вызывая же `dollars()`, мы спрашиваем: «Сколько у вас долларов?» Чувствуете разницу? Во втором случае я не рассматриваю объект как хранилище данных. Я уважаю его. Мне нужно знать, сколько долларов есть у объекта, но я не рассчитываю на то, что их количество хранится в приватном поле. Я не делаю предположений о его внутренней организации и уж точно не думаю о нем как о структуре данных.

В первом случае данные скрыты, во втором — нет. Они выставлены напоказ — любой пользователь класса видит их.

Вывод здесь один: геттеры и сеттеры — ужасный антипаттерн ООП.

Никогда не называйте так свои методы.

Dog написал 21 апреля 2017 года:

> «В настоящем объектно-ориентированном программировании объекты — такие же живые существа, как вы и я». Я еще могу представить себе такое в Erlang, но уж точно ни в каком другом языке. Знать о том, что объекты содержат слоты и vtable (или как они там у вас в языке называются), жизненно важно, чтобы их корректно использовать. Вам также необходимо знать, как они управляют памятью, как взаимодействуют с потоками и много других технических заморочек. В отличие от собаки они большей частью за собой следить не будут. Если бы мне давали доллар всякий раз, когда я пытался использовать класс из *стандартной библиотеки* в потоке, отличном от главного, а он при этом обрушивал программу, у меня было бы намного больше денег. Было бы замечательно, если бы объекты были живыми

существами и я мог бы доверять им следить за собой. Однажды это может сбыться, но сегодня это совсем не так. Это тупая свалка байтов, с которой надо работать строго определенным образом, иначе она уничтожит Вселенную. Собака ест, когда ей хочется, и ходит в туалет, когда ей хочется, даже если ей об этом не говорить. Если вы долго будете игнорировать свою собаку, она может съесть ваш обед и нагадить на ковер, но не совершит самоубийство, не начнет рожать тысячу щенков в секунду и не сожжет ваш дом.

Егор Бугаенко:

Вы правы. Java и другие недо-ООП-языки не считают объекты живыми существами. Да, мы должны следить за ними, иначе они сожгут наше жилище. Но кто создал эти языки? Мы, программисты, не понимающие, кто такие объекты. Одна из основных целей данной заметки, помимо прочего, состоит в том, чтобы поменять мировоззрение тех самых людей, которые проектируют языки, библиотеки и т. п.

Ivan P. написал 22 ноября 2016 года:

В реальном мире не все будет объектом, а только то, что имеет поведение. Стол, к примеру, не имеет собственного поведения. Следовательно, его не стоит описывать как объект. Какой именно — DTO-объект[1] или какой-то еще, — не имеет значения. Описывать сущности без поведения как объекты — значит усложнять код.

Егор Бугаенко:

У чего нет поведения, так это у данных.

---

[1] Data Transfer Object (объект для передачи данных) — паттерн проектирования, используемый при передаче данных между подсистемами приложения или уровнями архитектуры (уровень бизнес-логики, хранения данных и т. п.). — *Примеч. пер.*

Lewis Cowles написал 17 сентября 2016 года:

Что интересно, речь скорее о том, что API объектов должно иметь более естественный синтаксис, а не о паттерне «Геттер — сеттер» как таковом. Я могу принять такой ход мыслей, и мне кажется, что это заставит более рационально мыслить о проблемной области, но беспокоит то, что это будет сложнее объяснить и понять, особенно в мультикультурных командах, в частности, тем, кто «пересекает вертикальную линию»[1].

AlexKubicalL написал 27 июля 2016 года:

Здравствуйте, Егор. А как насчет представления данных в виде форм с множеством полей, не имеющих объектного поведения как такового? С помощью этих полей я должен иметь возможность сохранять, редактировать и фильтровать данные. Такое, мне кажется, трудно сделать без геттеров и сеттеров.

Егор Бугаенко:

Похоже, вы используете DTO-объекты. Не делайте этого. Есть много других альтернатив.

Ivan Stankov написал 12 июля 2016 года:

Допустим, у нас есть проект со всеми этими правильными собаками и прочим. Проект находится в разработке уже более года,

---

[1] Завуалированная ссылка на азиатов, в частности китайцев. Здесь, вероятно, подразумеваются региональные различия в написании арабской цифры 4. В европейских языках обычно не имеет значения, пересекать ли вертикальную линию при написании цифры 4, но в китайском языке есть похожий по написанию иероглиф 4, поэтому китайцы всегда пересекают вертикальную линию при написании цифры 4, см. https://en.wikipedia.org/wiki/Regional_handwriting_variation#Arabic_numerals. — *Примеч. пер.*

и вы наняли нового программиста. Он получает задачу подыскать какой-то семье пса. Разработчику нужно создать новую собаку. Он ищет фабрику собак, генератор собак и т. д. и т. п., но ничего похожего не находит. Через некоторое время ему говорят, что есть классы вроде `DogShelter`, `ZooShop` и другие подходящие для решения задачи. Итак, `DogShelter.getDog` — нет такого, `createDog` — нет такого, `constructDog` — нет такого, `newDog` — нет такого, да боже мой, может, `getFlyweightDog` — конечно же, и его нет. Ага, в итоге он находит метод `adoptDog`. В этом и есть суть заметки?

Егор Бугаенко:

Да, именно! Существующие на сегодня соглашения (например, геттеры — сеттеры) плохи, так как базируются на неправильных принципах. Да, они весьма популярны и позволяют программистам быстрее делать работу. Но в большинстве случаев из-за этого страдает качество работы. Я пытаюсь изменить принципы. Неизбежно мне приходится менять соглашения.

Sagiri написал 23 марта 2016 года:

Для примера с классом Dog не существует концептуально корректного варианта геттера. Take не подойдет, потому что, когда я беру что-то у кого-то, у него этого чего-то больше нет. Это противоречит тому, что ball не может быть равен `NULL` (предположительно, потому, что у каждой собаки есть мяч), не говоря уже о том, что класс Dog должен быть неизменяемым. Может быть, можно наблюдать за мячом или анализировать его, но тогда возникает еще бóльшая путаница, поскольку наблюдение и анализ — более активные действия, чем то, что происходит на самом деле. С основной идеей, впрочем, я согласен. Если мыслить в терминах объектов, то мы не устанавливаем имя или размер — мы переименовываем или меняем размер чего-либо.

Scott Palmer написал 13 ноября 2015 года:

Эта заметка — практически полная бессмыслица. Она начинается с того, что предлагает притвориться, будто структур данных не существует, а оттуда катится вниз по наклонной.

Егор Бугаенко:

В объектно-ориентированном программировании нет структур данных. Мы оставили их позади 20 лет назад в языке C. Идите в ногу со временем!

Scott Palmer:

Когда в руках молоток, все вокруг кажется гвоздями. Объектно-ориентированное программирование не означает, что нужно забыть все, что есть, и подгонять все подряд, без разбору под объектную парадигму. Это глупо. Для каждой задачи нужно использовать подходящий инструмент. Иногда все, что вам нужно, — простая структура данных. Как бы то ни было, в заметке масса других проблем. Идея о том, что слова get и set волшебным образом порочат имена методов, абсурдна. «Собака — неизменяемый живой организм, который не позволяет никому извне менять ее вес, рост, кличку и т. п.» — очевидная ложь. Я могу поменять кличку собаки. И мой питомец не имеет права голоса по этому вопросу. Если у меня есть собака, то я могу поменять ее кличку так: `dog.setName("Spot")` или `dog.name("Fido")`, и, откровенно говоря, вариант с префиксом `set` более очевиден. Не нужно мне указывать, что я не могу поменять кличку собаки, не заводя новую (неизменяемые собаки, вы это серьезно?).

Егор Бугаенко:

Менять кличку собаки и устанавливать ее — разные вещи. В заметке речь об этом. Кличку можно поменять, но устанавливать ее не стоит.

bedobi написал 16 августа 2015 года:

Мне нравится ваш блог, но мне кажется, что переименованием методов многого не добиться. Независимо от названий, основной проблемой остается инкапсуляция и то, как она подталкивает разработчиков относиться к объектам как к тупым мешкам данных и тем самым использовать процедурный подход вместо объектно-ориентированного. Вместо того чтобы просто переименовывать методы, можно и нужно полностью запретить возможность изменения объектов, в частности, с помощью set-методов. Get-методы должны предоставляться только при необходимости и возвращать только неизменяемые копии своих значений, которые бросают исключения при любых попытках их изменить.

Егор Бугаенко:

Действительно, в идеале нам необходимо избавиться от сеттеров и сделать объекты неизменяемыми.

Esteban написал 21 июня 2015 года:

Допустим, у меня есть графическое приложение. Справа у него панель инструментов, позволяющая менять свойства объектов. Допустим, мы выбрали объект кисти, используемый в трех изображениях цыплят на ферме. На панели справа пользователь может поменять значения красного, зеленого, синего и альфа-каналов с помощью четырех ползунков. Когда пользователь перемещает один из ползунков, цвет перьев должен меняться у всех цыплят одновременно. Так должно быть, потому что все три рисунка внутри ссылаются на этот объект кисти. Как вы реализуете объект кисти, не добавляя в класс сеттеры и геттеры, изменяющие значения красного, зеленого, синего и альфа-каналов? Эти свойства могут быть независимо изменены пользователем с помощью элементов управления.

Егор Бугаенко:

Есть несколько возможных решений. Первый и самый простой — создать класс `Brushes`, возвращающий кисть по ее идентификатору. Все изображения будут знать только идентификатор кисти и ссылаться на класс `Brushes`. Получать необходимую кисть они смогут по требованию.

Greg Bonney написал 21 июля 2015 года:

Приятно видеть, что не один я думаю, что «Геттер — сеттер» — это антипаттерн. К сожалению, он настолько укоренился в API Java и других похожих языков, что его практически невозможно избежать.

Matt написал 9 июля 2015 года:

Может быть, я что-то упускаю, но по большей части данные примеры все так же используют геттеры и сеттеры, но дают им неочевидные имена. `dog.weight()` — метод, ничем не отличающийся от `dog.getWeight()`. Для тех, кто не осилил[1]: геттеры и сеттеры — всего лишь соглашение об именовании, их не имеет смысла называть антипаттерном и некорректно утверждать, что это плохая практика программирования.

hasufell написал 27 мая 2015 года:

Весьма интересный взгляд. Мне кажется, что вы на верном пути и что с вами приятно работать как с ОО-программистом,

[1] В оригинале TL;DR (too long;didn't read) — таким сокращением в интернет-сленге обозначают упрощенное изложение длинной статьи или комментария. В русскоязычном сегменте Интернета с похожей целью используется фраза «Не осилил — много букв». — *Примеч. пер.*

но после прочтения я подумал, что вы упустили из виду чисто функциональное программирование. Многое из того, о чем вы говорите, уже включено по умолчанию в языки вроде Haskell (неизменяемость, прямолинейный код, инкапсуляции, отсутствие нулевых ссылок). Есть даже дополнительные преимущества в виде отсутствия побочных эффектов. В таких языках не получится мыслить по-старому, императивным способом. Конечно, от ООП тоже есть отличия, но с учетом приведенных пожеланий по улучшению современного образа мышления следующим логичным шагом будет, как мне кажется, чисто функциональное программирование.

Егор Бугаенко:

Действительно, функциональное программирование весьма близко по духу правильно реализованному ООП. Но я все же считаю, что они различаются и что ООП мощнее.

Nikola Boricic спросил 14 февраля 2015 года:

Как насчет сценария, когда ресурсы ограничены, а производительность критична? К примеру, в мобильных Android-приложениях объекты `ListView` не будут неизменяемыми, поскольку длинные списки станут приводить к отказу приложений. Вместо этого есть пара объектов, которые при прокрутке списка используются повторно. Считаете ли вы это плохой практикой? Как бы вы работали с таким случаем, не применяя методы-мутаторы[1]?

Егор Бугаенко:

Я бы по-прежнему использовал неизменяемые объекты, но на основе изменяемого хранилища в памяти.

---

[1] Mutators (англ.) — обобщенное название геттеров и сеттеров. — *Примеч. пер.*

Hans-Peter Storr написал 2 января 2015 года:

Меня несколько смущает то, что вы говорите о неизменяемых живых организмах. Я понимаю, что живой организм будет удачной метафорой, например, акторов в Scala. Но если «оно» не изменяется, то не будет ли «оно» статуей собаки, а не самой собакой? Или фотографией собаки, притом что действия порождают новые фотографии? Я несколько затрудняюсь придумать хорошую метафору для неизменяемости.

Егор Бугаенко:

Неизменяемый — не значит тупой или безжизненный.

---

oddparity написал 1 ноября 2014 года:

Нет, программирование не религия, как вы пишете ниже. И ваша заметка, мне кажется, поддерживает эту мысль. Программирование — процесс создания инструмента. Инструмент должен работать, должен быть сопровождаемым, расширяемым, отлаживаемым, верифицируемым. Он будет работать и с геттерами, и с сеттерами.

Егор Бугаенко:

Да, он будет работать с геттерами, сеттерами, синглтонами, статическими методами, без юнит-тестов, с божественными объектами, спагетти-кодом и без всякой документации. Будем честными: бóльшая часть кода, с которым мы сталкиваемся, работая в этой индустрии, делается именно так. Нравится ли нам это? Хотим ли мы это улучшить? Любим ли мы свою профессию или работаем за еду? Я думаю, что программирование — это образ жизни, религия, искусство, но никак не процесс создания инструмента. Вы проводите 1 % жизни на свиданиях, а 80 % — за компьютером. Почему мы должны встречаться с красивыми

мужчинами/женщинами, но при этом не беспокоимся о красоте собственного кода?

oddparity:

Мне тоже нравится красивый код. Но переименование `getBall()` в `giveBall()` много проблем не решит. К тому же я не особо люблю сюрпризы. Некоторые из моих коллег могут реализовать метод `spitBall()`, некоторые даже `getBall()` (те, что всегда говорят: «Принеси мне мяч»). Хотелось бы как можно быстрее находить метод, возвращающий мяч, а поэтому я придерживался бы общепринятых соглашений об именовании, как это делают остальные. Я скорее хотел бы, чтобы мои коллеги вносили больше креатива в хорошо спроектированные классы (учитывающие, к примеру, ваши рекомендации по другим темам) и хорошее микропрограммирование. Кроме того, мои объекты не цифровые воплощения живых существ, а запрограммированные проекции файлов, папок и их содержимого. Бумага и то, что на ней написано, не обладают интеллектом, а процесс, работающий на ней, — да. Мне нравятся идеи мистера Фаулера[1], но я, похоже, предпочитаю «безжизненный» взгляд на проектирование.

Bruno Skvorc написал 31 октября 2014 года:

Это все придирки ради славы и денег. Делайте качественные приложения и называйте методы как угодно.

Егор Бугаенко:

Да, будь хорошим мальчиком, слушай маму — и все будет хорошо. Для детей это подойдет, но в серьезной разработке ПО нужны правила, принципы, дисциплина. ООП дает нам дисциплину, если мы ее правильно понимаем.

---

[1] Мартин Фаулер (Martin Fowler) — автор ряда книг, посвященных архитектуре приложений. — *Примеч. пер.*

3.6. Не используйте оператор new вне вторичных конструкторов

Обсуждение на http://goo.gl/U8F8nq.

Поговорим о внедрении зависимостей. Честно говоря, мне нравятся это название и шумиха вокруг него. Знаете, не будем об этом. Поговорим о чистом, дисциплинированном ООП. Мы не избежим разговора о внедрении зависимостей, инверсии управления и других паттернах проектирования, имеющих отношение к зависимостям.

В небольших и молодых приложениях проблема не слишком очевидна, но она становится важна, иногда даже жизненно важна, в крупных системах. Например:

```
class Cash {
  private final int dollars;
  public int euro() {
    return new Exchange().rate("USD", "EUR")
      * this.dollars;
  }
}
```

Вот так выглядят проблемы. Мы создаем экземпляр класса `Exchange`, используя оператор `new` прямо внутри метода `euro()`. Почему это вызывает проблемы? Не обязательно вызывает (при условии, что классы небольшие, простые и не задействуют дорогих ресурсов вроде сети, диска, базы данных и т. п.).

Проблемы вызывает нечто, называющееся *жестко запрограммированной зависимостью*. Действительно, класс `Cash` связан с классом `Exchange`, в результате чего мы не можем ликвидировать такую зависимость, не редактируя код *внутри* класса `Cash`.

Представьте себе ситуацию, когда исходник класса `Cash` недоступен, а класс все равно приходится использовать. Или код доступен, но его нельзя изменять. У нас просто есть библиотека

в двоичном формате, и мы ее обязаны применять. Код может выглядеть так:

```
Cash five = new Cash("5.00");
print("$5 соответствует %d", five.euro());
```

Я проверяю метод `print()` и не хочу, чтобы класс при каждом запуске юнит-теста обращался в Нью-Йоркскую фондовую биржу. Меня не волнует, как работает метод `five.euro()`. Все, что мне нужно, — результат. Я не хочу тестировать класс `Cash`. Я хочу тестировать собственный код и чтобы класс `Cash` создавал как можно меньше шума. Если каждый раз при запуске тестов он будет подключаться к NYSE по HTTP, это станет сильно раздражать и моим первым вопросом к разработчику этого класса будет: «Как мне настроить класс `Cash` так, чтобы он перестал обращаться к бирже?»

В текущем варианте реализации класса `Cash` такое абсолютно невозможно. Связь между классами `Cash` и `Exchange` *нерушима*. Чтобы их расцепить, придется изменять исходный текст класса `Cash`. Эта проблема незначительна тогда и только тогда, когда класс невелик, но в более глобальном масштабе жестко запрограммированные зависимости мешают тестированию и сопровождению ПО.

Корень зла — оператор `new`.

Поскольку мы позволяем объектам инстанцировать другие объекты где и когда им удобно, то почему жалуемся, когда они делают это где хотят? `Cash` может порождать экземпляры `Exchange` — вот в чем проблема. Представьте ситуацию, когда оператор `new` запрещен внутри методов. Объекты не смогут порождать новые объекты. Они смогут только принимать их в качестве аргументов конструктора и инкапсулировать в приватных полях. Класс `Cash` будет выглядеть примерно так:

```
class Cash {
  private final int dollars;
  private final Exchange exchange;
  Cash(int value, Exchange exch) {
    this.dollars = value;
    this.exchange = exch;
  }
  public int euro() {
    return this.exchange.rate("USD", "EUR")
      * this.dollars;
  }
}
```


Проблема решена. Вот как *должен* будет выглядеть наш код:

```
Cash five = new Cash(5, new FakeExchange());
print("$5 соответствует %d", five.euro());
```

Мы должны передавать экземпляр `Exchange` в качестве второго аргумента конструктора. Класс `Cash` не может инстанцировать его самостоятельно. Он работает только с тем обменником, который ему предоставят. Он больше не зависит от `Exchange`. Вообще, конечно, зависит, но теперь зависимость контролируем мы, а не он. Он не решает, где ему взять курс обмена долларов на евро. Он полагается на наше решение и работает с тем объектом, который мы ему дадим.

Иными словами, вместо того, чтобы позволять объекту создавать зависимость по своему усмотрению, мы *внедряем* ее посредством конструктора.

Такое внедрение — хорошая практика. Класс `Cash` разработан таким образом, что его конструктор ожидает *все* необходимые зависимости, и такое поведение — образец для подражания. Так нужно разрабатывать все объекты. Для удобства мы можем добавить несколько вторичных конструкторов, как описано в разделе 1.2:

```
class Cash {
  private final int dollars;
  private final Exchange exchange;
  Cash() { // вторичный
    this(0);
  }
  Cash(int value) { // вторичный
```

```
    this(value, new NYSE());
  }
  Cash(int value, Exchange exch) { // основной
    this.dollars = value;
    this.exchange = exch;
  }
  public int euro() {
    return this.exchange.rate("USD", "EUR")
      * this.dollars;
  }
}
```

Одноаргументный конструктор внедряет экземпляр класса NYSE. Но это вторичный конструктор. Первичный конструктор позволяет нам полностью контролировать то, с какими зависимостями работает объект.

Я предлагаю простое правило, обеспечивающее высокое качество всех ваших объектов: не используйте оператор new нигде, кроме вторичных конструкторов. Взгляните на приведенный ранее код еще раз. Как видите, оператор new применяется только во вторичном конструкторе и нигде больше. Если вы запретите использование оператора new где-либо еще, ваши объекты будут полностью расцеплены и их верифицируемость и сопровождаемость повысятся.

Вы можете спросить, что делать, если объект должен инстанцировать другие объекты. Допустим, у нас есть объект, который представляет поток запросов, скажем, от сетевого сокета:

```
class Requests {
  private final Socket socket;
  public Requests(Socket skt) {
    this.socket = skt;
  }
  public Request next() {
    return new SimpleRequest(
      /* прочесть данные из сокета */
    );
  }
}
```


Каждый вызов метода `next()` должен создавать объект типа `Request` и возвращать его. Действительно, здесь нам нужен оператор new, и это не конструктор. Да, этот код нарушает правило, о котором идет речь в данном разделе. Вот как мы решаем эту проблему:

```
class Requests {
  private final Socket socket;
  private final Mapping<String, Request> mapping;
  public Requests(Socket skt) {
    this(
      skt,
      new Mapping<String, Request>() {
        @Override
        public Request map(String data) {
          return new SimpleRequest(data);
        }
      }
    );
  }
  public Requests(Socket skt,
    Mapping<String, Request> mpg) {
    this.socket = skt;
    this.mapping = mpg;
  }
  public Request next() {
    return this.mapping.map(
      /* прочесть данные из сокета */
    );
  }
}
```

Мы инкапсулируем экземпляр класса `Mapping`, который отвечает за конвертирование текстовых данных в экземпляр класса `Request`. Как видите, оператор new используется только во вторичном конструкторе. В методе `next()` его больше нет. Такой подход делает класс `Requests` конфигурируемым и избавляет его от жестко запрограммированных зависимостей. В коде больше нет жестко прописанных зависимостей. Мы можем внедрить собственную реализацию класса `Mapping`, который не применяет `SimpleRequest`, но возвращает, к примеру, что-то пригодное для тестирования.

Было бы хорошо, если бы такое правило было встроено в язык и строго соблюдалось, но это вопрос завтрашнего дня. А пока что имейте в виду: всякий раз, когда вы используете оператор new в методах или основных конструкторах, вы делаете что-то не так. Единственное законное место оператора new — вторичные конструкторы.

Думаю, это все, что вам нужно знать о внедрении зависимостей и инверсии управления. Это простое правило в совокупности с неизменяемыми объектами сделает ваш код чистым и подготовит его к внедрению зависимостей.

Bartosz Miera написал 8 января 2018 года:

> Я считаю, что основная проблема с оператором new() состоит в том, что он за кадром использует синглтон (!), который почти все считают вселенским злом. Будем честными: куча (heap), оказывается, является синглтоном, так как она доступна глобально и существует в каждом приложении в единственном экземпляре. Разумно применять оператор new для создания объектов только на верхнем уровне кода. Остальные объекты нужно порождать посредством других объектов, таких как фабрики и т. п.

3.7. Избегайте интроспекции и приведения типов

Обсуждение на http://goo.gl/BoQ2iq.

Время от времени весьма соблазнительно использовать интроспекцию и приведение типов, однако держитесь от них подальше, чего бы это ни стоило. С технической точки зрения речь идет об операторе `instanceof` и методе `Class.cast()` в Java или их аналогах в других языках. Задействуя этот оператор, мы

можем проверять тип объекта во время исполнения программы, например:

```
public <T> int size(Iterable<T> items) {
  if (items instanceof Collection) {
    return Collection.class.cast(items).size();
  }
  int size = 0;
  for (T item : items) {
    ++size;
  }
  return size;
}
```

Интроспекция — один из приемов, известных под общим названием *«рефлексия»*. Рефлексия тоже зло, но зло, не имеющее прямого отношения к ООП. Вот почему мы не будем его подробно обсуждать. Используя рефлексию, вы можете изменять методы, инструкции, выражения, классы, объекты, типы и т. п. во время исполнения. Вы модифицируете код прежде, чем его достигнет процессор. Это очень мощный и в то же время очень грязный прием, который сводит на нет сопровождаемость кода. Думаю, очевидно, что трудно читать код, когда нужно держать в голове, что он может быть модифицирован другим кодом во время исполнения. Чтение такого кода превращается в кошмар. Короче говоря, рефлексия — хороший инструмент для плохих программистов.

Приведенный ранее Java-метод вычисляет размер итерируемого объекта. Прежде чем перебрать и пересчитать элементы, он проверяет, относится ли объект `items` к типу `Collection`, в котором уже есть метод `size()`. Это явная оптимизация, верно? Нет нужды перебирать элементы, если есть короткий путь. Мы проверяем тип во время исполнения и действуем соответствующим образом.

Этот подход кажется удобным и оптимальным, но на деле он отвратителен.

Этот подход серьезно нарушает принципы ООП путем *дискриминации* объектов по типу. Действительно, мы взаимодействуем с объектом `items` по-разному в зависимости от его типа. Вместо того чтобы позволить объекту решать, как выполнить то, что от него требуется, мы принимаем решение без его участия, сегрегируя тем самым объекты на плохие и хорошие. С философской точки зрения это категорически неправильно. Это выглядит агрессивно и неуважительно. И напоминает расовую, гендерную, этническую, возрастную и любую другую дискриминацию в мире людей. Когда вы принимаете человека на работу, то не обращаете внимания на его пол. Вы говорите ему или ей, что необходимо сделать, и ожидаете, что результат будет удовлетворять заявленным вами требованиям. Не будет ли странным, если инструкции для мужчин и женщин будут различаться? То же самое применимо и к объектам. Мы должны избегать дискриминации объектов и позволять им делать свою работу без оглядки на то, *кем* они являются.

С технической точки зрения интроспекция типов во время исполнения — тоже плохой прием, поскольку он усиливает сцепленность классов. Взгляните на приведенный ранее пример еще раз. Наш метод зависит от двух интерфейсов — `Iterable` и `Collection`, а не просто от `Iterable`. Большее число зависимостей означает более тесную связь и худшую сопровождаемость. Что особенно плохо — эти зависимости скрыты. Мы не знаем, что метод использует интроспекцию. Зависимость между методом и классом `Collection` скрыта.

Кроме того, чтобы эффективно применять этот метод, мы должны знать, как он устроен. Мы должны будем заглянуть в исходный код, чтобы убедиться, что он действительно ведет себя по-другому, если ему передать экземпляр класса `Collection`. Намного лучше будет сделать так:

```
public <T> int size(Collection<T> items) {
  return items.size();
}
public <T> int size(Iterable<T> items) {
  int size = 0;
  for (T item : items) {
    ++size;
  }
  return size;
}
```


Этот прием известен как *перегрузка метода* и доступен не во всех языках. В Ruby, например, он не поддерживается, но есть возможность создать два метода с разными именами:

```
def sizeOfIterable(items)
  # ...
end
def sizeOfCollection(items)
  # ...
end
```

Теперь пользователь класса вынужден решать, какой метод использовать. В Java решение принимает компилятор, в Ruby это надо делать вручную, имея на руках информацию о типе объекта.

То же справедливо и в отношении *преобразования классов*, когда мы принуждаем объект подчиняться контракту, под выполнением которого он не подписывался:

```
return Collection.class.cast(items).size();
```

Эта строка может выглядеть следующим образом:

```
return ((Collection) items).size();
```

С технической точки зрения строки практически идентичны. Конечный результат — то, что объект `items` *становится* типа `Collection`. Это как если бы вы вызвали сантехника и сказали: «Я полагаю, что вы еще и компьютерщик, — почините мне принтер». Погодите, вот более подробный пример:

```
if (items instanceof Collection) {
  return ((Collection) items).size();
}
```

Это звучит как «Если вы еще и компьютерщик, то почините мне принтер». Это уже намного лучше, чем делать необоснованное предположение, на основе которого просить сантехника отремонтировать принтер. Тем не менее это все равно плохо, прежде всего из-за скрытого сцепления. В следующий раз, прежде чем отправлять к вам сантехника, фирма будет пытаться подобрать того, кто по совместительству является компьютерщиком, поскольку они помнят, что в прошлый раз вы заплатили еще и за наладку принтера. Контракт между вами и фирмой будет официально включать починку стока, как и раньше, однако будет подразумевать еще и ремонт принтера.

Если вы завтра решите сменить фирму по ремонту сантехники, вам снова придется искать сантехника-компьютерщика. Но эта информация в контракте не записана. Та же проблема возникнет, если теперешняя фирма решит поменять сотрудников. Скажем, парня, который с вами работал, уволили, а вам прислали нового. В вашем контракте прописана починка стока. Вам предоставят хорошего сантехника, но вы останетесь недовольны, поскольку хотите сантехника с дополнительной квалификацией, которая в контракте не указана.

Иными словами, вы выражаете свои *ожидания* относительно объектов, явно не документируя их. Некоторые клиенты *запомнят* ваши потребности и будут предоставлять вам более подходящие объекты, а некоторые — нет. Такие непрозрачные, скрытые отношения, основанные на неписаных соглашениях, серьезно влияют на сопровождаемость вашего продукта.

В общем, необходимо избегать любого использования оператора `instanceof` или приведения типов. Они не приносят никакой пользы вашему ПО, несмотря на то что их предоставляют почти все ООП-языки в рамках механизма рефлексии. Они только усугубляют беспорядок.

Lily спросил 8 января 2018 года:

Считаете ли вы, что приводить к более частному типу в Java плохо?

Егор Бугаенко:

Да, безусловно.

Andriy спросил 12 июля 2017 года:

Будет ли антипаттерном приведение Object к интерфейсу?

Егор Бугаенко:

Определенно да.

Ross William Drew написал 3 апреля 2015 года:

Если у вас есть `Collection`, который рассматривается как `Iterable` и в таком виде передается методу, то тогда метод интерфейса `Iterable` станет выполнять лишние действия над `Collection`, считая его `Iterable`. Следуя вашей аналогии, которую я нахожу несколько примитивной, вы все равно сегрегируете, но в итоге у вас негры делают то, что делают белые. Я бы здесь посоветовал решение, состоящее в том, что `Iterable` (или `Iterator`) должен иметь метод `size()`, чтобы его можно было спросить о количестве элементов, а не подсчитывать их количество извне.

Егор Бугаенко:

Вы правы, но это только подтверждает мою мысль. Решение о том, как вы будете обращаться с объектом, приходящим в метод, должно быть отражено в сигнатуре метода и нигде больше. Если я объявляю `java.util.List` как `java.lang.Iterable` и передаю ее методу, ожидающему `List` или `Iterable`, то я хочу,

чтобы вы считали меня `Iterable` и не обращались к методу `size()`. Это мое решение, не ваше.

Ross William Drew:

Может быть, но ваше решение подвержено риску ошибок в том смысле, что пользователь метода/класса не должен забывать приводить передаваемые коллекции к более общему типу, иначе исполнение программы замедлится или вовсе произойдет отказ. Если в них есть элементы, приведенные к частным типам, которые, в свою очередь, должны быть переданы классу/методу, то ваше решение просто сдвигает выбор типа на основе `instanceof` вверх по стеку вызовов. Мое предложение состоит в следующем: идеальным решением будет принимать только объекты, обернутые в интерфейс, поведение которого единообразно во всех реализациях (например, наличие метода `size()`). Пользователь в таком случае будет вынужден писать хороший с точки зрения ООП код — у него не возникнет желания каждый раз делать `instanceof` при использовании ваших классов. Всегда пишите код так, будто тот, кто его будет читать, — агрессивный психопат, который знает, где вы живете[1]. Я бы посоветовал писать интерфейсы точно так же.

[1] Перифраз известного высказывания, оригинал см.: https://groups.google.com/forum/#!msg/comp.lang.c++/rYCO5yn4lXw/oITtSkZOtoUJ. — *Примеч. пер.*

4 Уход на пенсию

Жизненный цикл объекта начинается с оператора `new` и заканчивается тогда, когда он больше никому не нужен. Обычно объекты делают свое дело и мы на них не жалуемся. Хотя иногда они выбрасывают исключения, когда им не нравится то, что они видят. Исключения — хороший ООП-прием, не имеющий, однако, никакой связи с объектной парадигмой, при этом они здорово помогают в обработке ошибок и оптимизации кода. Благодаря исключениям у нас нет необходимости решать проблемы в каждом методе по отдельности. Мы просто можем передать их на уровень выше.

Позже мы обсудим, что с ними делать после этого.

Однако исключения легко использовать не по назначению. Совсем не по назначению. Нет ничего хуже для сопровождаемости, чем некорректная обработка исключений.

Глава посвящена возврату `NULL` из методов, обработке исключений и получению ресурсов. Этот материал на данный момент весьма спорный. На данный момент — значит, в существующей реализации объектно-ориентированных языков. Я очень надеюсь, что в ближайшем будущем некоторые из озвученных в этой главе идей будут реализованы в ООП-языках.

4.1. Никогда не возвращайте NULL

Обсуждение на http://goo.gl/TzrYbz.

Использовать `NULL` в качестве аргумента метода — плохо, как мы уже обсуждали в разделах 2.6 и 3.3. Очень плохо. Не знаю, как насчет других парадигм программирования, но в объектно-ориентированном и процедурном — точно. Теперь обсудим, почему возвращать `NULL` — тоже плохо. Как обычно, начнем с примера на Java:

```
public String title() {
  if (/* нет заголовка */) {
    return null;
  }
  return "Элегантные объекты";
}
```

Это настолько отвратительно, но при этом настолько распространено, что я даже не знаю, с чего начать. Начнем с того, почему это отвратительно, а затем проанализируем, почему этот прием получил такое широкое распространение в мире ООП.

Во-первых, такой подход заставляет нас делать то, против чего агитирует раздел 3.3, — считать объекты существами с ограниченными возможностями. Мы попросту не можем доверять объекту, который был возвращен из метода `title()`. Не можем доверять его способностям. Он инвалид. Он нуждается в особом к себе обращении:

```
String title = x.title();
print(title.length());
```

Мы не можем вызвать `title.length()`, не боясь получить `NullPointerException`. Проблема не в самом исключении. Исключение — всего лишь техническое неудобство. Истинная проблема крупнее. Проблема в потере нашего *доверия*.

Мы не можем утверждать, что наши объекты самодостаточны, целостны, уважаемы, умны и т. п. Они не являются таковыми. Мы не можем попросить их сделать что-то и надеяться на результат. Мы должны проверить, является ли объект объектом в принципе.

Такие проверки — ужасное нарушение объектно-ориентированной парадигмы:

```
String title = x.title();
if (title == null) {
  print("Не могу вывести - не название.");
  return;
}
print(title.length());
```

Суть объекта в том, что это сущность, которой мы *доверяем*. Это не фрагмент данных, который не знает о наших намерениях и просто предоставляет участок памяти с удобным доступом и набором подпрограмм. Это не маркер, который мы передаем между узлами системы. Это не конверт для данных. Нет и еще раз нет.

Объект — живой организм со своими собственными жизненным циклом, поведением и состоянием. Он либо существует и жив, либо не существует и мертв. Третьего не дано. *Переменная* — лишь псевдоним объекта:

```
String t = x.title();
```

В данном случае `t` — лишь псевдоним объекта, возвращаемого методом `title()`. Мы доверяем объекту и надеемся, что переменная означает то же, что и объект. Под доверием я понимаю то, что объект несет полную *ответственность* за свои действия и мы никоим образом не должны ему мешать. Он работает так, как вздумается. Если он хочет вывести имя, мы не возражаем. Если хочет выбросить исключение, так тому и быть. Но мы

не должны выбрасывать исключение, даже не поговорив с ним! Это неправильно и *неуважительно*:

```
if (title == null) {
  print("Не могу вывести – не название.");
  return;
}
```

Такая проверка — верный признак недоверия в приложении. Я не доверяю методу `title()`, соответственно, кто-то другой не станет впоследствии доверять мне. Использование `NULL` приведет к крупной потере доверия во всем приложении и превратит его в неподдерживаемый бардак. Да, это тоже имеет отношение к *сопровождаемости*. Нехватка доверия приводит к серьезному ухудшению сопровождаемости просто потому, что, когда я читаю код, мне приходится затрачивать больше времени на то, чтобы понять, какому из вызываемых методов я могу доверять, а какой может вернуть `NULL`. Мне также приходится дважды проверять результат, прежде чем использовать его и общаться с возвращенным объектом.

Все это очень похоже на рабочие отношения в команде. Если мне каждый раз приходится проверять составленные коллегой документы, то работа серьезно замедляется. Не поймите меня неправильно — я только за контроль качества. Мы должны проверять корректность результатов, при необходимости дважды, но за это должен отвечать кто-то другой, а не я — тот, кто получает документ от своего коллеги. Я должен иметь возможность работать с документом, как только получу его. Я должен иметь основания *доверять* своему коллеге. Это касается не личных отношений, а скорости работы всей команды. Нам *нужно* доверие, но `NULL` у нас его отнимает.

Если мои коллеги могут схитрить и вернуть `NULL`, моя работа существенно замедлится. Мне придется проверять их всех, в результате код станет намного более многословным. Рано или поздно я забуду проверить действия одного из них. Я не смогу

чувствовать себя в безопасности в собственной команде, в собственном ПО.

Короче говоря, это плохо и неприлично. Метод, возвращающий `NULL`, ведет себя неуважительно. Он не уважает меня как своего пользователя, поскольку может схитрить и вернуть мне недействительный документ.

Почему же этот прием так популярен и так часто применяется? Взгляните на метод `listFiles()` в классе `Files` из Java 1.2. Ему нужно перебрать все файлы в папке и вернуть их массив. Он не выбрасывает исключения при отсутствии каталога, а вместо этого возвращает `NULL`. Вот как я должен его использовать:

```
void list(File dir) {
  File[] files = dir.listFiles();
  if (files == null) {
    throw new IOException("Directory is absent.");
  }
  for (File file : files) {
    System.out.println(file.getName());
  }
}
```

А вот как я должен был бы его применять, если бы он бросал исключения вместо того, чтобы возвращать `NULL`:

```
void list(File dir) {
  for (File file : dir.listFiles()) {
    System.out.println(file.getName());
  }
}
```

Думаю, очевидно, что второй фрагмент короче, чище, лучше сопровождается и в целом более качественный. Почему разработчики Java решили возвращать `NULL`, вместо того чтобы выбрасывать `IOException`? Похоже, во время разработки JDK они не слышали о принципе *скорейшего отказа*. Они думали, что лучше молча возвращать `NULL` и давать возможность бросать исключение пользователю класса, вместо того чтобы бросать

исключение сразу же после обнаружения того, что каталога не существует. Они пытались угодить нам, но у них не получилось.

Отказывать как можно скорее или как можно безопаснее?

По сути, есть два противоположных подхода к отказоустойчивости ПО — отказывать *как можно скорее* или *как можно безопаснее*. Я ярый сторонник первого подхода и противник второго.

Стратегия безопасных отказов побуждает нас делать все возможное, чтобы приложение продолжало работать, даже если мы столкнулись с логической ошибкой, ошибкой ввода/вывода, переполнением памяти и т. п. Что бы ни случилось, приложение должно выжить. Возвращать `NULL` — прием выживания. К примеру, если мы выяснили, что каталог, файлы внутри которого нас попросили перечислить, отсутствует, можем ли мы вывести их? Не можем. Запрос, очевидно, некорректен. Его автор не проверил наличие каталога, прежде чем попросить вывести список его файлов. Это его или ее проблема, но мы попробуем *выровнять* ситуацию. Не станем бросать `IOException`, вместо этого вернем `NULL`, чтобы кто-то другой решил проблему. Надеемся, что никто не станет перебирать элементы возвращенного массива и мы не получим `NullPointerException`. Надеемся.

Противоположный подход состоит в том, чтобы отказывать как можно скорее. Он мотивирует нас остановить исполнение и выбросить исключение, как только мы столкнемся с проблемой (любой!). Нас не должны волновать последствия. Приложение должно быть настолько хрупким, насколько возможно, но при этом полностью покрыто юнит-тестами. Если приложение хрупкое и может отказать в любой контрольной точке, то юнит-тесты могут с легкостью воспроизвести эти ситуации, а мы исправим

их. Если приложение откажет в режиме эксплуатации, мы легко можем добавить тест, учитывающий сложившуюся ситуацию, просто потому, что все точки отказа очевидно и хорошо документированы. Мы выпячиваем их, вместо того чтобы прятать. Делаем их заметными и легко отслеживаемыми. Мы бросаем исключение `IOException` в тот же момент, когда выясняем, что каталога не существует. Мы не будем выравнивать ситуацию, наоборот, сделаем ее как можно более *вопиющей*. Если нам передали некорректный каталог, пусть разбираются с этим сами. Им придется исправить свою ошибку и быть более аккуратными в следующий раз.

Какой подход лучше? Как я уже говорил, я ярый сторонник скорейшего отказа. Я считаю, что можно добиться стабильности и устойчивости приложения, только если немедленно сообщать о выявленных ошибках. Чем раньше мы обнаружим проблему и вызовем отказ, тем лучше со временем станет приложение. Напротив, чем дольше мы скрываем проблему, тем бóльшими в итоге окажутся неприятности.

Это может показаться контринтуитивным, поскольку мы не хотим останавливать приложение. И не хотим, чтобы оно падало. Мы не хотим видеть трассировок стека. В этом-то и подвох. Мы не хотим признавать, что в приложении есть логические ошибки и их полным-полно. Некоторые из них очевидны и легко обнаруживаются, другие же хорошо спрятаны. Но они есть. Пряча голову в песок, мы оказываем себе медвежью услугу. Вместо того чтобы обнаружить рану и залечить ее, мы прячем ее и говорим пациенту, что все будет хорошо. Чем раньше мы увидим проблему, тем быстрее ее исправим. Чем раньше она проявится, тем быстрее должна быть наша реакция. Каждая исправленная ошибка делает продукт стабильнее и устойчивее.

Почему же так много Java-методов возвращают `NULL`, а не бросают исключение? Скорее всего, так происходит из-за приверженности

их разработчиков философии безопасных отказов. Я к ним не отношусь. И всячески рекомендую держаться от нее подальше. Вызывайте отказ как можно скорее, если вы беспокоитесь о качестве не только отдельного метода, но и приложения в целом.

Альтернативы NULL

Какие существуют альтернативы возврату `NULL`? Иногда заманчиво возвращать `NULL`, если искомый объект не может быть найден, например:

```
public User user(String name) {
  if (/* имя не найдено в базе данных*/) {
    return null;
  }
  return /* запись из базы данных */;
}
```

Мой опыт говорит о том, что именно в таких случаях разработчики чаще всего возвращают `NULL` вместо настоящих объектов. Мы не находим способа лучше, чем этот, чтобы сообщить клиенту, что объект, который он ищет, недоступен. Мы не бросаем исключение, так как не считаем такую ситуацию исключительной. Клиент ищет имя в базе данных, но такого пользователя там нет. Мы не хотим, чтобы приложение из-за этого падало, правда? Это вполне стандартная, даже рутинная ситуация. Если пользователь не найден, мы возвращаем `NULL` и движемся дальше.

Как вы понимаете, такой ход мыслей очень близок к философии безопасных отказов, рассмотренной ранее. Не делайте так. Я предлагаю несколько альтернатив использованию `NULL`.

Первый вариант — разбить метод на два. Первый метод будет проверять существование объекта, а второй — возвращать его.

Второй метод должен бросить исключение, если пользователь не найден:

```
public boolean exists(String name) {
  if (/* имя не найдено в базе данных*/) {
    return false;
  }
  return true;
}
public User user(String name) {
  return /* запись из базы данных */;
}
```

Проблема с этим подходом следующая: он неэффективен. Мы обращаемся к базе данных два раза: сперва проверяем существование записи в базе данных, затем обращаемся к ней, чтобы, собственно, получить ее.

Вот почему я предлагаю второй вариант. Вместо того чтобы возвращать `NULL` или бросать исключение, мы можем вернуть коллекцию объектов, например:

```
public Collection<User> users(String name) {
  if (/* имя не найдено в базе данных*/) {
    return new ArrayList<>(0);
  }
  return Collections.singleton(
    /* из базы данных */
  );
}
```

Если ничего не найдено, коллекция будет пуста. Затем клиент работает с коллекцией, чтобы получить из нее объекты. Чисто технически это незначительно отличается от использования `NULL`, но при этом выглядит несколько чище. Обратите внимание на то, что я переименовал метод. Теперь он называется `users()`, а не `user()`.

Еще одним вариантом будет применение generic-класса `java.util.Optional` из Java 8 либо аналогичного. Он похож на коллекцию, но может содержать только один элемент. Я считаю, что это

решение противоречит принципам ООП, и не рекомендую использовать его, так как оно семантически некорректно. Метод по-прежнему называется `user()`, но то, что он станет возвращать, будет не пользователем, а чем-то вроде конверта для пользователя. Это сбивает с толку и не соответствует духу объектно-ориентированного мышления. К тому же смахивает на `NULL`-ссылки. Не используйте этот вариант.

Последнее, что я могу вам предложить, — паттерн «Пустой объект». В случае, когда искомый объект не найден, возвращается объект, похожий на настоящий, но ведущий себя по-другому. Он может делать что-то одно, но не делать что-то другое. К примеру, если мы ищем пользователя по имени Джефф и не находим его, то возвращаем объект, имеющий такое же имя и возвращающий его при вызове `name()`. На все другие запросы он бросает исключения. Такой подход вполне в духе объектного мышления, но имеет ограниченное применение. Обратите внимание на то, что тип возвращаемого объекта остается неизменным. К примеру, экземпляр `NullUser` — объект того же типа, что и `SqlUser`. Оба они реализуют интерфейс `User`. К примеру, `NullUser` может выглядеть следующим образом:

```
class NullUser implements User {
  private final String label;
  NullUser(String name) {
    this.label = name;
  }
  @Override
  public String name() {
    return this.label;
  }
  @Override
  public void raise(Cash salary) {
    throw new IllegalStateException(
      "Пользователь-заглушка — невозможно повысить зарплату"
    );
  }
}
```


Короче говоря, никогда не возвращайте `NULL`. Даже не думайте. Нет никакого оправдания существованию `NULL` в ООП-языках. В Java и других языках это ключевое слово токсично. Просто держитесь от него подальше. Если вам нужно вернуть что-то, что не было найдено, то либо бросьте исключение, либо верните коллекцию или пустой объект.

Вот три возможные альтернативы.

4.2. Бросайте только проверяемые исключения

Обсуждение на http://goo.gl/5tGDEc.

Пришло время поговорить о проверяемых и непроверяемых исключениях. Хотя многие объектно-ориентированные языки поддерживают только непроверяемые исключения, Java поддерживает оба типа. Обобщу этот раздел сразу: непроверяемые исключения — недостаток языка, все исключения должны быть проверяемыми. Иметь несколько типов исключений тоже плохо.

Это очень абстрактный и непрактичный подход, поскольку он сильно противоречит реальному состоянию большинства платформ разработки. Основная часть из них, включая Ruby, C#, Python, Scala и многие другие, поддерживает только непроверяемые исключения. В них попросту нет проверяемых исключений. Вот почему почти все, о чем я говорю в этом разделе, может лишь помочь вам поменять образ мышления, но не даст никаких реальных, применимых на практике рекомендаций, если только вы не пишете на Java или не собираетесь создать собственный объектно-ориентированный язык.

Я, однако же, надеюсь, что будущие ООП-языки будут более строгими, чем существующие, и станут уделять больше внимания обработке ошибок с помощью исключений. В дальнейшем

я хотел бы предложить то, что считаю правильным способом обработки исключений. Он более логичен и чист. И сейчас я его продемонстрирую.

Вначале посмотрим, в чем различие между проверяемыми и непроверяемыми исключениями и для чего вообще нужны разные типы исключений. Вот как выглядит использование проверяемого исключения в Java:

```
public byte[] content(File file) throws IOException {
  byte[] array = new byte[1000];
  new FileInputStream(file).read(array);
  return array;
}
```

Обратите внимание на сигнатуру этого простого метода — она оканчивается на `throws IOException`. Это значит, что, если я вызываю `content()`, я должен во что бы то ни стало ловить данное исключение:

```
public int length(File file) {
  try {
    return content(file).length();
  } catch (IOException ex) {
    // Нужно что-то сделать с этим
    // исключением — либо разрешить его
    // прямо сейчас, либо передать
    // на уровень выше.
  }
}
```

Я не могу вызвать метод `content()`, не неся полной ответственности за ту проблему, которую он может вызвать. Под проблемой я имею в виду `IOException`. Этот метод небезопасен, поскольку может создать проблему. Я снова говорю об `IOException`. Он может отказать из-за некоторой проблемы в подсистеме ввода/вывода. Я полагаю, что отказ будет иметь отношение к файловой системе. Говоря `throws IOException`, метод, по сути, перекладывает *ответственность* на мои плечи. Он заставляет меня принимать решение в случае, когда с файлом что-то не так.

Я могу сделать то же и *переложить* ответственность на своих клиентов, также объявив себя небезопасным:

```
public int length(File file) throws IOException {
  return content(file).length();
}
```

В данном примере я больше не ловлю исключение. Я позволяю ему всплыть. Я выполняю эскалацию проблемы, как в управлении проектом или предприятием. С проблемой разберутся те, кто находится выше в стеке вызовов, а не я. Я просто говорю, что не знаю, что делать в такой ситуации, и прошу помощи.

Исключение `IOException` — проверяемое, поскольку его необходимо ловить. Мы не можем проигнорировать его существование в методе `length()`. И должны либо ловить его, либо обозначить себя `throws IOException`. Вот почему проверяемые исключения всегда *на виду*. Работая с методом `length()`, мы должны помнить, что работаем с *токсичным* и *небезопасным* методом под названием `content()`. Мы должны либо обозначить себя как небезопасный метод, либо снять токсичность, разрешив исключительную ситуацию.

Напротив, непроверяемые исключения можно проигнорировать и не ловить вовсе. Возникнув, они автоматически всплывают до тех пор, пока их кто-нибудь не перехватит. Язык не заставляет нас что-либо с ними делать. К примеру, исключение `IllegalArgumentException` — непроверяемое:

```
public int length(File file) throws IOException {
  if (!file.exists()) {
    throw new IllegalArgumentException(
      "Невозможно вычислить размер файла, так как его
       не существует"
    );
  }
  return content(file).length();
}
```

В данном примере сигнатура метода никак не упоминает исключение `IllegalArgumentException`. Когда кто-либо вызывает метод `length()`, то не знает, чего ожидать. Информация об `IllegalArgumentException` скрыта. Именно это я имел в виду, когда говорил о том, что проверяемые исключения всегда *на виду*.

Не ловите исключения без необходимости

При разработке метода мы рано или поздно сталкиваемся с выбором: ловить все исключения, чтобы метод был безопасным для пользователей, или осуществлять эскалацию проблемы. Я предпочитаю второй вариант. Передавайте исключения как можно выше по стеку вызовов. Для существования каждого блока `catch` должна быть веская причина. Иными словами, не ловите исключения без особой необходимости, делайте это, только если у вас нет другого выбора.

Идеальным будет приложение, в котором на каждую точку входа есть единственный блок `catch`. К примеру, если речь идет о мобильном приложении, которое взаимодействует с пользователем через экран смартфона, то у него одна точка входа и, соответственно, должен быть единственный блок `catch` на все приложение. К сожалению, это почти невозможно, поскольку сам язык и многие фреймворки для него разработаны по другим принципам.

Мы уже обсуждали разницу между быстрыми и безопасными отказами в разделе 4.1. То же почти дословно применимо и здесь. Философия, заключающаяся в том, что приложение можно сделать устойчивым, любой ценой решая проблемы в том же месте, где они происходят, делает приложение нестабильным

и сложным для сопровождения. Вот что мы можем сделать в своем методе:

```
public int length(File file) {
  try {
    return content(file).length();
  } catch (IOException ex) {
    return 0;
  }
}
```

Метод `length()` теперь совершенно безопасен. Что бы ни произошло с файловой системой, в нем не случится отказ. Он вернет целое число, и приложение продолжит работу. Это типичный пример подхода «безопасный отказ». Мы видим проблему, но не хотим расстраивать клиента. Хотим, чтобы приложение выглядело привлекательно и никогда не ломалось. Мы хотим выглядеть надежно, поэтому возвращаем нуль, даже когда в действительности в файловой системе, к примеру, заканчиваются доступные дескрипторы файлов. Файловая система не может получить длину файла, даже если он существует. Она сигналит нам, она кричит и плачет, но мы ее игнорируем. Что бы ни случилось, мы все прячем. Просто возвращаем нуль.

К тому же, пряча проблему, мы оказываем медвежью услугу всем, в том числе клиенту, вызывающему метод `length()`. Да, приложение не упадет сразу, поскольку получит нуль и продолжит что-то делать. Но со временем оно упадет, так как нуль — некорректный размер файла. Оно упадет вдали от вызова `length()`, в результате чего невозможно будет понять, что вызвало отказ. Часы отладки понадобятся на то, чтобы выяснить, что число, которое вернул метод `length()`, было лишь показателем ошибки в файловой системе.

Этот подход известен также как «использование исключений для управления потоком исполнения». Действительно, в приведенном

ранее примере мы применяем уведомление об исключительной ситуации для ветвления программы. Мы делаем нечто подобное, только при помощи исключений:

```
public int length(File file) {
  if (/* Проблема с файловой системой. */)
    return 0;
  } else {
    return content(file).length();
  }
}
```

Такое ветвление допустимо, но исключения — инструмент для другой работы. Исключения сделаны не для того, чтобы заменять условные операторы. Напротив, они должны обозначать критическую ситуацию, не допускающую восстановления, в результате возникновения которой прекращается нормальное исполнение программы и нужно принимать чрезвычайные меры. Подробнее о *восстановлении* поговорим через пару минут.

Некоторые из вас могут возразить, что вместо того, чтобы возвращать нулевой размер файла, можно вернуть `-1` или `NULL`. В разделе 4.1 мы уже говорили о том, почему возвращать `NULL` — плохая идея. Возврат `-1` немногим от нее отличается, поскольку это не пустой объект, а скалярное значение, семантически близкое к `NULL`. Практически идентичное. Полностью идентичное, раз уж на то пошло. Возвращая `-1`, мы вынуждаем наших клиентов не доверять возвращаемому результату и всегда перепроверять его:

```
int length = length(new File("test.txt"));
if (length == -1) {
  print("Хм... что-то не так.");
} else {
  print("Размер файла равен %d" + length);
}
```

Проблемой здесь является сравнение с использованием оператора `==`. Это признак недоверия объекту `length`, как говорилось в разделе 4.1. Мы ожидаем размер файла, но получаем что-то

иное. Это не размер файла, а сигнал о том, что не следует считать полученный результат числом. Мы должны помнить, что подобное предательство может случиться, и быть готовыми к нему. Если забудем выполнить сравнение оператором `==`, то можем оказаться в серьезной беде. К примеру, мы решим прочитать `length` байт из файла, а значение `length` равно `-1`. Последствия этого непредсказуемы.

И что еще важнее, будет очень трудно обнаружить причину проблемы.

Суть моих слов в том, что поимка исключения и спасение ситуации — серьезные действия, которые должны иметь под собой вескую причину. Каждый раз, когда вы спасаете ситуацию, не перебрасывая исключение, вы используете подход безопасного отказа.

Стоит ли говорить, что подход «ловить и записывать в журнал» — ужасный антипаттерн. Думаю, это уже очевидно.

Стройте цепочки исключений

Стойте, мы же еще не поговорили о том, что такое перебрасывание исключений. Вот как это выглядит, и это совершенно корректный прием:

```
public int length(File file) throws Exception {
  try {
    return content(file).length();
  } catch (IOException ex) {
    throw new Exception(
      "Невозможно определить размер файла.",
      ex
    );
  }
}
```

Я ловлю исключение, но тут же бросаю новое. Использование цепочек исключений, как продемонстрировано ранее, — хорошая

практика. Заменяя одну проблему другой, я не игнорирую факт существования первой. Напротив, я оборачиваю исходную проблему в новую и вместе бросаю их на уровень выше.

Если так сделать несколько раз, то всплывшее исключение будет выглядеть как мыльный пузырь с мыльным пузырем внутри. Внутри того тоже будет пузырь и т. д. Будет много слоев. Блок `catch`, который решит что-то сделать с этой проблемой и спасти ситуацию, проткнет пузырь и достанет из него остальные. Что именно будет делать блок `catch` для разрешения ситуации и уведомления о проблеме, не имеет значения. Что важно — мы поднимаем источник проблемы с нижнего уровня на уровень приложения в целом.

Однако приведенный далее код плох, поскольку он игнорирует источник проблемы:

```
public int length(File file) throws Exception {
  try {
    return content(file).length();
  } catch (IOException ex) {
    // Здесь я игнорирую проблему 'ex' и создаю
    // новую, нового типа, с новым
    // сообщением:
    throw new Exception("Не могу вычислить размер");
  }
}
```

В самом деле ужасная практика. Мы теряем важную информацию об источнике проблемной ситуации с вводом/выводом. Внутри у объекта `ex` наверняка было сообщение наподобие `"Слишком много открытых файлов(24)"`. Мы его игнорируем и создаем новое исключение, которое гласит: `"Не могу вычислить размер"`. Новое исключение начнет всплывать и со временем будет поймано блоком `catch` на уровне объекта приложения. Ценная низкоуровневая информация окажется потеряна. Потребуются часы или даже дни, чтобы определить, почему не удалось вычислить размер файла.

Уверен, это очевидно, но повторю еще раз: делайте цепочки исключений и никогда не игнорируйте исходную проблему.

Вы можете спросить: «Зачем нам вообще нужны исключения? Почему бы не сделать так, чтобы методы были небезопасными, а исключения просто всплывали наверх? Зачем в приведенном примере ловить `IOException` и бросать его снова, обернув в `Exception`? Что не так с уже существующим классом `IOException`?» Ответ очевиден: построение цепочек исключений семантически обогащает контекст проблемной ситуации. Иными словами, получить сообщение `"Too many open files (24)"` недостаточно. Оно слишком низкоуровневое. Вместо этого хотелось бы видеть цепочку исключений, где исходное исключение касалось бы количества открытых файлов, в следующем говорилось бы о невозможности вычислить размер файла, в третьем — о том, что файл изображения не может быть прочитан, и т. д. Если пользователь не может открыть фото в своем профиле, то сообщения «Слишком много открытых файлов» недостаточно.

В идеале каждый метод должен ловить все возможные исключения и перебрасывать их, формируя тем самым цепочки исключений. Повторюсь: ловите все, объединяйте в цепочки и немедленно перебрасывайте.

Это лучший подход к обработке исключений.

Восстанавливайтесь единожды

Есть довольно популярный прием восстановления после сбоя, который стоит обсудить. Вообще говоря, чуть раньше мы о нем уже говорили. Речь все о том же конфликте между скорейшим и безопасным отказом, но уже под другим углом. Если мы придерживаемся принципа скорейшего отказа, то не можем восстановиться после исключения. Проще говоря, речь

о восстановлении идти не может. Это лишь другое название уже известного антипаттерна «Использование исключений для управления потоком исполнения». В данном коде вы, возможно, узнаете прием восстановления после сбоя:

```
int age;
try {
  age = Integer.parseInt(text);
} catch (NumberFormatException ex) {
  // здесь мы восстанавливаемся после сбоя
  age = -1;
}
```

Насколько этот пример отличается от рассмотренных ранее? Разницы никакой. Это антипаттерн, похожий на возврат `NULL`, который рассматривался в разделе 4.1.

Но я не совсем прав. Мы все же должны восстановиться, но только единожды. Всем методам должно быть разрешено перебрасывать исключения, не обрабатывая их, как обсуждалось в предыдущем разделе. Тогда все исключения будут всплывать на верхний уровень приложения. Точнее говоря, к точкам входа в приложение. К тем точкам, через которые пользователь общается с приложением. К примеру, если речь идет о приложении командной строки, которое пользователь запускает с терминала, то соответствующий код может выглядеть следующим образом:

```
public class App {
  public static void main(String... args) {
    try {
      System.out.println(new App().run());
    } catch (Exception ex) {
      System.err.println(
        "Извините, возникла проблема:"
        + ex.getLocalizedMessage()
      );
    }
  }
}
```


Как видите, внутри блока `catch` я ничего не перебрасываю. Я решаю проблему здесь и сейчас — просто говорю пользователю о проблеме, и все. Статический метод `main` нетоксичен. Он безопасен. Он никогда не падает, поскольку это верхний уровень приложения. Выше него ничего нет.

Если я не поймаю проблему там, то она поднимется в среду исполнения. Если это произойдет, то пользователь также увидит сообщение, но оно будет совсем не дружелюбным. Пользователю будет показано системное сообщение с полной трассировкой стека. Я не хочу, чтобы так произошло. Вместо этого мне нужно восстановиться.

Верхний уровень приложения — единственное законное место восстановления.

То же должно происходить в каждой точке входа в приложение. Их не так уж много даже в сложных системах. Я к тому, что количество законных мест для восстановления в приложениях обычно невелико. Во всех прочих местах мы должны либо ловить и перебрасывать исключения, либо не ловить их вообще. Первый вариант предпочтительнее. Ловите, стройте цепочку и перебрасывайте.

Восстанавливайтесь только на верхнем уровне. Вот и все.

Используйте аспектно-ориентированное программирование

Иногда может возникнуть необходимость повторить неудачно выполненную операцию.

Скажем, мы пытаемся загрузить веб-страницу посредством HTTP-запроса. Вполне возможно, что подключение иногда будет сбоить. Было бы некрасиво показывать пользователю

сообщение об ошибке и заставлять его перезапускать приложение. Мы можем повторить запрос, правда?

Но чтобы повторить запрос, придется ловить исключение и восстанавливаться:

```
public String content() throws IOException {
  int attempt = 0;
  while (true) {
    try {
      return http();
    } catch (IOException ex) {
      if (attempt >= 2) {
        throw ex;
      }
    }
  }
}
```

Прежде чем выбросить исключение `IOException`, метод сделает не более трех попыток подключения. Такой метод небезопасен, но не сразу. Он предпринимает несколько попыток, прежде чем эскалировать проблему. Хотя этот подход весьма удобен, он противоречит всему сказанному ранее в текущем разделе, так как метод восстанавливается раньше, чем исключение попадает на уровень приложения. Это плохая практика? Да. Есть ли лучшее решение? Нет.

Вообще говоря, одно решение есть — аспектно-ориентированное программирование (АОП). Это очень простая и сильная парадигма программирования, которая хорошо сочетается с объектно-ориентированной. Точнее, не совсем парадигма — скорее базовый прием, который может существенно упростить типовые операции и снизить многословность ООП-кода. Взглянем на приведенный фрагмент кода еще раз. Он довольно многословен. Повторный вызов метода — это десять строк кода. И он еще довольно примитивен. Надлежащая реализация будет намного

объемнее. В соответствующим образом реализованном механизме повторного вызова метода исключение не игнорируется, а некоторым образом записывается в журнал. Мы также добавим интервал между попытками, который будет алгоритмически увеличиваться. А еще добавим возможность настраивать количество попыток, не ограничиваясь жестко запрограммированной константой 3. Используя АОП в Java 6, мы поступим следующим образом:

```
@RetryOnFailure(attempts = 3)
public String content() throws IOException {
  return http();
}
```

Эта аннотация `@RetryOnFailure` будет подхвачена во время исполнения и обернет метод `content()` в блок кода «повторить при ошибке»[1]. Данный блок кода называется *аспектом*. С технической точки зрения он является объектом, который получает управление и решает, как и когда вызывается метод `content()`. Это своего рода адаптер метода `content()`. Красота аспектно-ориентированного программирования в том, что мы избегаем дублирования кода, вынося вспомогательные механизмы и приемы из основных классов. Настоятельно рекомендую подробнее изучить тему АОП и использовать его в своих проектах, независимо от языка.

АОП упомянуто здесь, чтобы показать, что в ООП досрочное восстановление после сбоя — плохая практика, которую можно (и нужно) заменить другими приемами. Повторный вызов метода при ошибке — один из примеров, где АОП помогает поддержать чистоту ООП, будучи при этом удобным и позволяющим достичь цели приемом.

---

[1] Вы можете найти эту аннотацию и увидеть в действии ее АОП-аспект на сайте http://aspects.jcabi.com/.

Достаточно одного типа исключений

Если вы согласны с принципами формирования цепочек исключений и отказа от немедленного восстановления после сбоя, то поймете, почему типизация исключений избыточна. Действительно, если мы восстанавливаемся лишь единожды, то у нас будет объект исключения, который содержит все остальные исключения. Если они правильно выстроены в цепочку, то зачем знать их тип?

Кроме того, мы не применяем исключения для управления потоком исполнения программы, правда? И никогда не ловим исключения, чтобы решить, что делать дальше. Мы ловим их только для того, чтобы перебросить, так? Если это действительно так, то нас мало интересует тип исключения. Мы все равно перебросим его. Эта информация нам не нужна, потому что она никогда не используется. Мы не ловим исключения по мере их продвижения наверх. Даже если и делаем это, то с единственной целью — добавить их в цепочку и перебросить.

Timofey Solonin спросил 11 ноября 2017 года:

> Не могли бы вы уточнить, что значит «нельзя восстанавливаться после сбоев»? В одной из веток комментариев ниже вы приводите отличный пример того, как избежать раннего возврата, — преобразовывать ошибку в реакцию системы. Значит ли это, что вы восстанавливаетесь? В конце концов, восстановиться где-то придется, поскольку иначе приложение падало бы при выбрасывании любого исключения. Прием перебрасывания и выстраивания цепочки ошибок вместо раннего возврата мне нравится, но когда ошибка поднимется в один из интерфейсных объектов, нам придется ее отобразить. Это значит, что придется перехватить эту ошибку и отобразить ее в виде объекта пользовательского интерфейса. Как бы мы это ни толковали, мы восстановились после сбоя. Так надо ли

> нам решать, в каком контексте уместно восстанавливаться? Как нам определить такой контекст? Возможно, я что-то понял совсем не так.

Егор Бугаенко:

> Я бы все же советовал не восстанавливаться — пусть ошибка всплывет на поверхность, где пользователь сможет ее увидеть и сообщить разработчикам. Иногда, однако, может оказаться необходимо перехватить ошибку раньше. Мы должны избегать таких мест и ситуаций.

Kapralov Sergey написал 7 сентября 2017 года:

> Мне нравится ваша аргументация. Даже очень. Она напоминает мне языки вроде Haskell, где если функции надо бросить исключение, то это необходимо отразить в ее контракте (сигнатуре). Однако когда я пытаюсь применить это на практике в Java, то сталкиваюсь с одной проблемой, которая все ломает. Интерфейсы. Если вы утверждаете, что небезопасные методы должны явно бросать проверяемые исключения, значит, этот факт необходимо отражать и в интерфейсах. Но интерфейс — всего лишь контракт, и вы не можете знать наверняка, какого подхода будут придерживаться исполнители контракта — безопасного или небезопасного. Это их дело. Записывая `throws Exception` в контракте метода, вы даете разрешение исполнителям использовать небезопасные инструкции, опуская же это требование, вы заставляете их применять лишь безопасные конструкции. Само по себе это хорошо (заставлять использовать безопасные конструкции), но не для Java, где все, по сути, небезопасно. Java не дает много гарантий во время компиляции, так как его система типов недостаточно строга, чтобы однозначно рассматривать какой-либо фрагмент кода как безопасный. `ClassCastException` может возникнуть в любом коде, использующем обобщенные классы и стирание типов-параметров. Должен ли я считать

код, бросающий такие исключения, небезопасным? От безысходности мне снова пришлось вернуться к непроверяемым исключениям.

Adam Spofford написал 12 декабря 2016 года:

У меня есть возражения по этой части. Как проверяемым, так и непроверяемым исключениям есть место в языке. Вы говорите, что ни одно исключение не должно быть программно важнее других, но суть не в важности. Речь о том, что вы можете контролировать. Если у меня есть числовой класс и я в нем вызываю метод `divide`, то этот метод будет бросать исключение при делении на 0. Но он не становится от этого небезопасным. Передаешь 0 в качестве делителя — сам дурак. Сбой файловой операции, однако же, может произойти по нескольким причинам. Исключение, вызываемое при передаче логического значения вместо пути файла, должно быть непроверяемым — сам виноват. Но когда файл был каким-то образом изменен во время операции записи, это уже проверяемое исключение. Такую ситуацию я могу обработать, но не могу быть ее виновником. Возвращаясь к делению на 0. Когда кто-то привел подобный аргумент, вы сказали, что исключению надо позволить всплыть наверх. Но тогда ваш метод тоже придется пометить как небезопасный, несмотря на то что это вызвано вполне безопасной операцией — вызов `divide(2)` программно небезопасен, хотя безопасен логически. Метод должен быть помечен как небезопасный, хотя в нем никогда не произойдет ошибки. А если вы хотите пометить его как безопасный, вам придется ловить исключение всякий раз при вызове метода `divide`. Теоретически это значит, что раз в объектно-ориентированном подходе не должно быть особых случаев, то при каждом использовании оператора *деления* также придется ловить исключение. Что касается типов исключений: допустим, я делаю GUI-приложение и хочу, чтобы пользователь ввел число. Пользователь вводит «двадцать», я пытаюсь (неудачно) преобразовать строку в число. Если исключения не типизированы, все, что я могу сказать: «Ошибка». Типизированные исключения позволяют мне сказать: «Это не число» или «Число слишком велико». Что касается восстановления: если пользователь оставил поле пустым, мне придется выбирать, как обработать такую ситуацию. Если пользователь ничего не ввел, я могу подставить число 20. Если введено некорректное значение, я также могу подставить 20. Таким образом, исключениям придется диктовать порядок исполнения программы. Эту ситуацию можно решить разными способами, но факт остается фактом.

Andrej Zirko написал 28 мая 2016 года:

Насколько я понимаю, стандарт языка Java определяет то, как можно пометить метод как небезопасный, — ключевое слово `throws`. Я тоже придерживаюсь такого подхода. Большинство разработчиков определяют, что метод небезопасный, глядя на его сигнатуру. Такую информацию обычно не ищут в Javadoc. Чтобы не перегружать сигнатуры методов многочисленными типами исключений, я оборачиваю исключение одного определенного типа, прежде чем перебрасывать их. Я, однако, не использую базовый класс `Exception`, как предлагается в заметке. Для каждого приложения или библиотеки я делаю один класс проверяемых исключений. Таким образом я обеспечиваю то, что мне не понадобится менять сигнатуру метода, если впоследствии в нем будут возникать другие исключения. При необходимости я могу добавить дополнительную информацию в исключение. И я не требую, чтобы пользователи моей библиотеки ловили исключение класса `Exception`. Я считаю, что бросать исключения, производные от `Exception`, — значит сбивать с толку большинство разработчиков, пользующихся моей библиотекой или приложением.

Егор Бугаенко:

Вы все делаете правильно. Я бы тоже не рекомендовал бросать или ловить исключения типа `Exception`. Определяйте собственный тип.

Robert DiFalco написал 15 сентября 2015 года:

Кажется, я запутался в вашем посте. Если, по-вашему, каждое исключение должно быть типа `Exception` и каждое исключение должно быть проверяемым, то не приводит ли это нас обратно к тому, что исключения должны быть непроверяемыми? Если они все равно всплывают наверх и нигде не обрабатываются, тогда зачем объявлять их в методах? К примеру, у меня есть код, который, как я думаю, никогда не вызовет исключение. Затем я реализую его так, что он использует нечто, что бросает исключение. Если это невосстановимое исключение, то я перебрасываю его. Теперь мне придется поменять сигнатуру метода, добавив туда `throws Exception`. Если вы объявляете все методы как бросающие `Exception` (тип везде одинаков), то в чем тогда ценность такого изменения? Возможно, я упустил важную часть вашего блога.

Егор Бугаенко:

Что ж, у нас должно быть два типа методов — безопасные и небезопасные. Безопасные методы не бросают исключений. Небезопасные могут выбросить исключение. Вот и все. Понятно?

Robert DiFalco:

Безопасных методов не бывает. Это иллюзия. Вот почему вы не должны бояться того, что все исключения будут непроверяемыми. А еще, если исключение непроверяемое, то оно с большей вероятностью всплывет. Не этого ли вы хотите?

Егор Бугаенко:

Согласен, безопасных методов нет. В Java. Но, может, в этом и проблема? Наверное, должны быть безопасные методы вроде `a+b` или `string.format()`. Я понимаю, что `OutOfMemory` может случиться всюду, но, возможно, такие исключения должны иметь другой статус/вид/тип? Я думаю... Честно говоря, на данный момент я с вами почти согласен. Тем не менее я все еще пытаюсь найти аргументы в свою защиту.

Arnaud написал 1 августа 2015 года:

Можно поспорить, что безопасных методов не существует и поэтому все методы должны бросать исключение `Exception`. Но если все методы бросают исключение `Exception`, то зачем его объявлять в принципе? Мне кажется, это не противоречит вашей позиции о том, что проверяемые исключения ошибочны, и вашей точке зрения о дихотомии safe/unsafe.

Егор Бугаенко:

Именно это я и предлагаю. Должны быть безопасные и небезопасные методы. Вот и все.

Arnaud:

Мне кажется, вы не поняли, в чем разногласие. Автор утверждает, что безопасных методов не бывает и что почти любой сколько-нибудь полезный код может бросить какое-нибудь исключение. Если так и есть, то ваше предложение помечать участок кода ключевыми словами `throws Exception` — ничего не значащий, бессмысленный шум.

Stefano Masiero написал 30 июля 2015 года:

Этот спор весьма запутан. Провокационная заметка Егора призывает освободить разум, зацикленный на общепринятом мнении. Почему в Java есть понятие проверяемого исключения? Мне кажется, это попытка заставить разработчика постоянно иметь в виду обработку ошибок, попытка избежать человеческого фактора. Но, как и любое правило, навязываемое языком, оно становится обременительным для хороших программистов (они всегда думают об обработке ошибок независимо от того, проверяются исключения или нет).

Darren Hoffman написал 29 июля 2015 года:

> Люди, пожалуйста, поймите, что непроверяемые исключения — это неявно объявленные исключения. Все методы неявно помечены `throws RuntimeException`. Дело не в том, что вы не должны ловить «скрытые» исключения, — вы должны ловить исключения, объявленные в сигнатуре метода. В коде они явно не прописываются, потому что в этом нет смысла. Нет возможности сказать, что метод может бросать непроверяемые исключения. Мне кажется, людей вводит в заблуждение слово «непроверяемое». Вам стоит подумать над его значением. Непроверяемое — не значит скрытое. Это значит, что компилятор ожидает, что оно может быть брошено в любом месте кода. Нет смысла объявлять его в сигнатуре метода. Вы можете ловить его, если вам так хочется, но только для диагностических или информационных целей.

4.3. Будьте либо константным, либо абстрактным

Обсуждение на http://goo.gl/vo9F2g.

Я еще ничего не сказал о *наследовании*. Пришло время поговорить об этом очень мощном и зачастую неправильно используемом механизме ООП. Я часто слышу, что наследование — зло и его необходимо избегать. Говорят, что инкапсуляция предпочтительна в большинстве случаев. Мне кажется, я с ними согласен, но давайте проанализируем, почему наследование создает проблемы и что можно сделать для их предотвращения. Действительно, нет смысла избавляться от наследования, но его надо применять умело.

Самый сильный аргумент против наследования — то, что оно делает отношения между объектами слишком запутанными.

Очень сложно понять иерархию классов, наследующих друг друга, когда ее глубина превышает, скажем, пять уровней. В этом есть смысл, но наследование не является источником проблемы. Проблемы бывают вызваны *виртуальными методами*. Взглянем на следующий пример:

```
class Document {
  public int length() {
    return this.content().length();
  }
  public byte[] content() {
    // Загружает необработанное содержимое
    // документа как массив байтов
  }
}
```

Не лучший способ абстрагирования документа, но такого примера достаточно, чтобы продемонстрировать, как наследование затрудняет чтение кода. Попробуем расширить этот класс возможностью загрузки зашифрованного документа:

```
class EncryptedDocument extends Document {
  @Override
  public byte[] content() {
    // Загружает документ, расшифровывает его на лету
    // и возвращает расшифрованное содержимое
  }
}
```

Выглядит корректно, правда? Метод `content()` класса `EncryptedDocument` загружает содержимое и расшифровывает его на лету. Но поведение метода `length()`, унаследованного классом `EncryptedDocument` от класса `Document`, поменялось. Он больше не возвращает размер документа на диске. Он возвращает размер расшифрованного содержимого. Этого ли мы от него ожидаем? Не факт. Вероятно, мы ожидаем, что он вернет размер хранилища, занятого документом, так же, как и в классе `Document`.

Легко ли понять, что не так с методом `length()` в дочернем классе `EncryptedDocument`? Это займет какое-то время. Мы должны

помнить, что он вызывает метод `content()`, который был переопределен. Мы будем просматривать исходный код класса `Document`, в котором определяется метод `length()`, имея в виду, что некоторые из вызываемых им методов находятся в дочерних классах. Такое мышление контринтуитивно. Наследование интуитивно представляется направленным сверху вниз — дочерние классы наследуют код родительских классов. Переопределение методов дает возможность родительскому классу получать доступ к методам дочернего класса. Скажем так, такое «перевернутое» мышление противоречит здравому смыслу.

Вот где наследование из удобного инструмента ООП превращается в источник проблем с сопровождаемостью. Сложность растет, и код становится тяжело читать и понимать. Но решение есть. Просто делайте свои классы и методы либо *константными*, либо *абстрактными* — и любая возможность возникновения проблемы растворится. Действительно, если бы класс `Document` был константным, мы бы в принципе не смогли от него наследовать. В то же время, если бы его метод `content()` был абстрактным, мы не смогли бы реализовать его в `Document` и с методом `length()` не возникло бы путаницы.

У класса, по сути, может быть три статуса: константный, абстрактный или другой. Константный класс — черный ящик для его пользователей. Он не может быть изменен наследованием. Он цельный и самодостаточный. Он знает, как работать, ему не нужна помощь. Мы не можем переопределить методы в константном классе чисто технически. Они навсегда константны.

Абстрактный класс — как незавершенный прозрачный ящик. Он не может работать самостоятельно, ему нужна помощь, часть его компонентов отсутствует. Он еще не класс с технической точки зрения. Он — полуфабрикат для создания настоящего класса. Формально в абстрактном классе можно переопределить некоторые методы, остальные будут константными.

Третье состояние — когда класс не является ни абстрактным, ни константным. Я категорически против него, поскольку оно не является ни черным, ни прозрачным ящиком. Довольно запутанная ситуация, поскольку класс может стать либо тем, либо другим. Мы можем переопределить некоторые методы, и тогда он станет прозрачным ящиком, но в то же время будет считать себя черным ящиком. Класс станет предполагать, что он цельный, самодостаточный и устойчивый, в то время как другим будет позволено, вразрез с этим предположением, заменять некоторые его элементы посредством виртуальных методов.

Вот как выглядел бы класс `Document`, если бы Java не позволял создавать классы, не являющиеся ни абстрактными, ни константными:

```
final class Document {
  public int length() { /* тот же */ }
  public byte[] content() { /* тот же */ }
}
```

Обратите внимание на модификатор `final`. Он указывает на то, что ни один из методов класса не может быть переопределен дочерними классами. Теперь нам нужно создать класс `EncryptedDocument`. Он должен наследоваться от `Document`, но мы не можем наследовать от него. Таким образом, придется ввести интерфейс, что, как обсуждалось в разделе 2.3, является хорошей практикой:

```
interface Document {
  int length();
  byte[] content();
}
```

Затем нужно переименовать класс `Document` в нечто вроде `DefaultDocument` и сделать так, чтобы он реализовал интерфейс `Document`:

```
final class DefaultDocument implements Document {
  @Override
```

```
  public int length() { /* тот же */ }
  @Override
  public byte[] content() { /* тот же */ }
}
```

А теперь последний шаг: нужно создать `EncryptedDocument`, использующий функционал `DefaultDocument`. Мы применим инкапсуляцию вместо наследования, поскольку для константного класса оно невозможно:

```
final class EncryptedDocument implements Document {
  private final Document plain;
  EncryptedDocument(Document doc) {
    this.plain = doc;
  }
  @Override
  public int length() {
    return this.plain.length();
  }
  @Override
  public byte[] content() {
    byte[] raw = this.plain.content();
    return /* Расшифрованное содержимое. */;
  }
}
```

Обратите внимание на то, что как `DefaultDocument`, так и `EncryptedDocument` являются константными и от них нельзя наследовать.

Данный пример показывает, что при обязательном использовании ключевых слов `final` и `abstract` наследование в большинстве случаев станет невозможным. Если все классы константные, доступна только инкапсуляция.

Если вы придерживаетесь этого принципа и помечаете все классы либо `final`, либо `abstract`, то вам почти не придется использовать наследование. Тогда, когда это будет осмысленно, вы сможете им воспользоваться. «Когда имеет смысл пользоваться наследованием?» — спросите вы. Тогда, когда нужно *уточнить* поведение класса. Не расширить, а уточнить. Разница есть. Расширение означает, что существующее поведение дополняется новым. Уточнение означает, что не полностью определенное поведение становится полностью определенным.

Мы не должны ничего расширять в ООП, поскольку данный процесс, как показано ранее, рассматривает объекты как прозрачные ящики, что нежелательно. Объекты задуманы как черные ящики и не терпят вторжения и нарушения их личного пространства. Расширение класса есть вторжение.

Вместо этого мы должны *уточнять* абстрактные классы, что ожидаемо. К примеру, у нас есть неполный класс `Document`, который знает, как вычислить свой размер:

```
abstract class Document {
  public abstract byte[] content();
  public final int length() {
    return this.content().length;
  }
}
```

Затем нужно *уточнить* его, введя новый класс `DefaultDocument`, который знает, как загрузить содержимое, скажем, с диска:

```
final class DefaultDocument extends Document {
  @Override
  public byte[] content() {
    // Загружает содержимое с диска
  }
}
```

Затем мы создаем класс `EncryptedDocument`, который уточняет класс `Document` по-другому:

```
final class EncryptedDocument extends Document {
  @Override
  public byte[] content() {
    // Загружает содержимое с диска,
    // расшифровывает и возвращает его
  }
}
```

Вы можете возразить, что в таком случае возникнет аналогичная проблема — метод `length()` вернет размер расшифрованного документа, а не файла на диске. Да, это так, но сделано уже сознательно. Оба класса уточняют абстрактный класс. Теперь четко видно, что метод `length()` применяет методы соответствующих классов. Вот почему уточнение — более чистый подход, чем расширение.

Подведем итог: возможность создавать классы, отличные от константных и абстрактных, — недостаток языка Java и многих других языков. Мы должны явно выражать свои намерения — метод либо разработан правильно, либо не разработан вообще. Третьего не дано.

4.4. Используйте принцип RAII

Обсуждение на http://goo.gl/ULUJ8o.

Принцип «выделение ресурсов есть инициализация» (Resource Acquisition Is Initialization (RAII)) — последнее, о чем я хотел бы упомянуть, прежде чем закончить книгу. Это очень мощный прием, существующий в C++, но отсутствующий в Java в силу того, что объекты в нем уничтожаются посредством сборки мусора. Вот почему в нем нет деструкторов. Мы, конечно, можем имитировать RAII в Java, но C++ реализует его гораздо элегантнее. Взглянем, как это работает в C++. Представим, что у нас есть абстракция текстового файла:

```
#include <stdio.h>
#include <string>
class Text {
public:
  Text(const char* name) {
    this->f = fopen(name, "r");
  }
  ~Text() {
    fclose(this->f);
  }
  const std::string& content() {
    // Считывает содержимое файла
    // и возвращает его как UTF8-строку
  }
private:
  FILE* f;
};
```


Вот как мы использовали бы данный класс:

```
int main() {
  Text t("/tmp/test.txt");
  t->content();
}
```

Вначале мы создаем `t` — объект класса `Text` путем вызова его конструктора `Text()`. Затем вызываем метод `content()`, чтобы прочесть файл. Потом покидаем область видимости объекта `t`, в результате чего вызывается его деструктор `~Text()`. Он закрывает файл.

Фокус в том, что ресурс захвачен на время жизни объекта. В данном примере дескриптор файла `f` будет захвачен до вызова деструктора. Отсюда прием и получил свое название — выделение (захват) ресурсов есть инициализация. Мы захватываем ресурс при инициализации объекта и освобождаем его, когда объект больше не нужен и будет уничтожен. Этот прием очень удобен. Рекомендую использовать его при любой возможности.

В Java и во многих других языках прием RAII применить нельзя, поскольку в них нет деструкторов. В Java, к примеру, объекты уничтожаются в фоновом режиме, когда в них больше нет необходимости.

Этот процесс называется *сборкой мусора*. Формально в Java объект `t` будет все еще жив после окончания выполнения метода `main()`. Нужен ли он нам после завершения метода `main()`?

Нет, но Java не уничтожает его автоматически. Вместо этого он долго находится в памяти, после чего объявляется мусором. Сборщик мусора удаляет объект лишь тогда, когда для новых объектов оказывается недостаточно памяти.

Вот почему в Java нет деструкторов. К несчастью.

Однако в Java 7 появилась возможность, похожая на RAII. Теперь мы можем использовать блок `try` со связанным ресурсом:

```
int main() {
  try (Text t = new Text("/tmp/test.txt")) {
    t.content();
  }
}
```

Объект `t` не будет уничтожен по завершении блока `try`, но будет вызван его метод `close()`, близкий по смыслу к деструктору в C++. Нам всего лишь нужно сделать так, чтобы класс `Text` реализовывал интерфейс `AutoCloseable`.

Я настоятельно рекомендую использовать RAII всюду, где приходится работать с настоящими ресурсами — файлами, потоками, подключениями к БД и т. п. В C++ применяйте деструкторы, в Java — интерфейс `AutoCloseable`.

Эпилог

Я свято верю, что объектно-ориентированное программирование ожидает светлое будущее. Java, C#, C++, Ruby, Python и другие псевдо-ООП-языки будут заменены более строгими, чистыми и элегантными языками. Я не знаю, когда это произойдет, но это непременно случится.

Проблема даже не в отсутствии хороших языков. Проблема в нас, нашем образе мышления, понимании ООП, в том, как мы продумываем и проектируем программное обеспечение, в нашем менталитете и наших принципах. Мы должны изменить подход к написанию кода, и программы ответят нам взаимностью. Языки начнут меняться, когда мы станем по-другому их использовать.

Я хочу, чтобы мы поменяли свой образ мышления. Вот почему написал эту книгу.

Конец.

Калифорния, Мальта, Украина
2015–2017

Егор Бугаенко

Элегантные объекты. Java Edition

Перевел с английского К. Русецкий

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художественный редактор	*С. Заматевская*
Корректоры	*Е. Павлович, Т. Радецкая*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 06.2018. Наименование: книжная продукция.
Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева,
д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 20.06.18. Формат 60×90/16. Бумага офсетная. Усл. п. л. 15,000.
Тираж 1000. Заказ № ВЗК-02843-18.

Отпечатано в АО «Первая Образцовая типография», филиал «Дом печати — ВЯТКА»
в полном соответствии с качеством предоставленных материалов
610033, г. Киров, ул. Московская, 122. Факс: (8332) 53-53-80, 62-10-36
http://www.gipp.kirov.ru; e-mail: order@gipp.kirov.ru