*Конспект книги Роберта Мартина «Чистый код. Создание, анализ и рефакторинг»*

# Чистый код

Чтобы написать чистый код, необходимо сознательно применять множество приемов, руководствуясь приобретенным усердным трудом чувством «чистоты». Ключевую роль здесь играет «чувство кода».

## Мнения, что такое «чистый код»:

* Код должен был элегантным и эффективным. Логика должны быть достаточно прямолинейной, чтобы ошибкам было трудно спрятаться; зависимости — минимальными, чтобы упростить сопровождение; обработка ошибок — полной в соответствии с выработанной стратегией; а производительность — близкой к оптимальной, чтобы не искушать людей загрязнять код беспринципными оптимизациями. Чистый код хорошо решает одну задачу.
* Чистый код прост и прямолинеен. Чистый код читается, как хорошо написанная проза. Чистый код никогда не затемняет намерения проектировщика; он полон четких абстракций и простых линий передачи управления.
* Чистый код может читаться и усовершенствоваться другими разработчиками, кроме его исходного автора. Для него написаны модульные и приемочные тесты. В чистом коде используются содержательные имена. Для выполнения одной операции в нем используется один путь (вместо нескольких разных). Чистый код обладает минимальными зависимостями, которые явно определены, и четким, минимальным API. Код должен быть грамотным, потому что в зависимости от языка не вся необходимая информация может быть четко выражена в самом коде.
* В порядке важности, простой код:
  + проходит все тесты;
  + не содержит дубликатов;
  + выражает все концепции проектирования, заложенные в систему;
  + содержит минимальное количество сущностей: классов, методов, функций и т.д.
* Вы работаете с чистым кодом, если каждая функция делает примерно то, что вы ожидали. Код можно назвать красивым, если у вас также создается впечатление, что язык был создан специально для этой задачи.

## Правило бойскаута:

Хорошо написать код недостаточно. Необходимо поддерживать чистоту кода с течением времени. Все мы видели, как код загнивает и деградирует с течением времени. Если мы все будем оставлять свой код чище, чем он был до нашего прихода, то код попросту не будет загнивать. Чистка не обязана быть глобальной. Присвойте более понятное имя переменной, разбейте слишком большую функцию, устраните одно незначительное повторение, почистите сложную цепочку if.

# Содержательные имена

## Имена должны передавать намерения программиста

Имя переменной, функции или класса должно отвечать на все главные вопросы. Оно должно сообщить, почему эта переменная (и т.д.) существует, что она делает и как используется. Если имя требует дополнительных комментариев, значит, оно не передает намерений программиста.

Примеры:

int elapsedTimeInDays;

int daysSinceCreation;

int daysSinceModification;

int fileAgeInDays;

Содержательные имена существенно упрощают понимание и модификацию кода.

## Избегайте дезинформации

Программисты должны избегать ложных ассоциаций, затемняющих смысл кода. Не используйте слова со скрытыми значениями, отличными от предполагаемого. Например, ее обозначайте группу учетных записей именем accountList, если только она действительно не хранится в списке (List). Примеры дезинформирующих имен встречаются при использовании строчной «L» и прописной «O» в именах переменных (т.к. они почти не отличаются от констант «1» и «0» соответственно).

## Используйте осмысленные различия

Недостаточно добавить в имя серию цифр или неинформативные слова, даже если компилятору этого будет достаточно. Если имена различаются, то они должны обозначать разные понятия.

Неинформативные слова также применяются для создания бессодержательных различий. Допустим, у вас имеется класс Product. Создав другой класс с именем ProductInfo или ProductData, вы создаете разные имена, которые по сути обозначают одно и то же. Info и Data не несут полезной информации, как и артикли a, an и the.

## Используйте удобопроизносимые имена

Если имя невозможно нормально произнести, то при любом его упоминании в обсуждении вы выглядите полным идиотом. «Итак, за этим би-си-эр-три-си-эн-тэ у нас идет пи-эс-зэт-кью, видите?» А это важно, потому что программирование является социальной деятельностью.

## Выбирайте имена, удобные для поиска

У однобуквенных имен и числовых констант имеется один специфический недостаток: их трудно искать в большом объеме текста.

Строка MAX\_CLASSES\_PER\_STUDENT отыскивается легко, а с числом 7 могут возникнуть проблемы. Система поиска находит эту цифру в именах файлов, в определениях констант и в различных выражениях, где значение используется с совершенно другим смыслом. Еще хуже, если константа представляет собой длинное число, в котором были случайно переставлены цифры; в программе появляется ошибка, которая одновременно скрывается от поиска.

Лично я считаю, что однобуквенные имена могут использоваться ТОЛЬКО для локальных переменных в коротких методах. Длина имени должна соответствовать размеру его области видимости. Если переменная или константа может встречаться или использоваться в нескольких местах кодового блока, очень важно присвоить ей имя, удобное для поиска.

## Избегайте схем кодирования имен

У нас и так хватает хлопот с кодированием, чтобы искать новые сложности. Кодирование информации о типе или области видимости в именах только создает новые хлопоты по расшифровке.

Про венгерскую запись: Java-программисту кодировать типы в именах не нужно. Объекты обладают сильной типизацией, а рабочие среды развились до такой степени, что могут выявить ошибку типа еще до начала компиляции! Таким образом, в наши дни венгерская запись и другие формы кодирования типов в именах превратились в обычные пережитки прошлого. Они усложняют изменение имени или типа переменных, функций и классов. Они затрудняют чтение кода. Наконец, они повышают риск того, что система кодирования собьет с толку читателя кода.

Префиксы m\_, которыми когда-то снабжались переменные классов, тоже стали ненужными. Классы и функции должны быть достаточно компактными, чтобы вы могли обходиться без префиксов. Кроме того, люди быстро учатся игнорировать префиксы (и суффиксы), чтобы видеть содержательную часть имени. Чем больше мы читаем код, тем реже замечаем префиксы.

## Избегайте мысленных преобразований

Не заставляйте читателя мысленно преобразовывать ваши имена в другие, уже известные ему.

Такая проблема часто возникает при использовании однобуквенных имен переменных. Конечно, счетчик цикла можно назвать i, j или k (но только не l!), если его область видимости очень мала, и он не конфликтует с другими именами. Это связано с тем, что однобуквенные имена счетчиков циклов традиционны. Однако в большинстве других контекстов однобуквенные имена нежелательны; в сущности, вы создаете временный заменитель, который должен быть мысленно преобразован пользователем в реальную концепцию.

## Имена классов

Имена классов и объектов должны представлять собой существительные и их комбинации: Customer, WikiPage, Account и AddressParser. Старайтесь не использовать в именах классов такие слова, как Manager, Processor, Data или Info. Имя класса не должно быть глаголом.

## Имена методов

Имена методов представляют собой глаголы или глагольные словосочетания: postPayment, deletePage, save и т.д. Методы чтения/записи и предикаты образуются из значения и префикса get, set и is согласно стандарту javabean7.

string name = employee.getName();

customer.setName("mike");

if (paycheck.isPosted())...

При перегрузке конструкторов используйте статические методы-фабрики с именами, описывающими аргументы. Например, запись

Complex fulcrumPoint = Complex.FromRealNumber(23.0);

обычно лучше записи

Complex fulcrumPoint = new Complex(23.0);

## Избегайте остроумия

Если ваши имена будут излишне остроумными, то их смысл будет понятен только людям, разделяющим чувство юмора автора — и только если они помнят шутку.

## Выберите одно слово для каждой концепции

Выберите одно слово для представления одной абстрактной концепции и придерживайтесь его. Например, существование в разных классах эквивалентных методов с именами fetch, retrieve и get неизбежно создаст путаницу.

Имена функций должны быть законченными и логичными, чтобы программист мог сразу выбрать правильный метод без сбора дополнительной информации.

## Воздержитесь от каламбуров

Старайтесь не использовать одно слово в двух смыслах. В сущности, обозначение двух разных идей одним термином — это каламбур.

Задача автора — сделать свой код как можно более понятным. Код должен восприниматься с первого взгляда, не требуя тщательного изучения. Ориентируйтесь на модель популярной литературы, в которой сам автор должен доступно выразить свои мысли, а не на академическую модель, в которой ученик усердным трудом постигает скрытый смысл публикации.

## Используйте имена из пространства решения

Не забывайте: ваш код будут читать программисты. А раз так, не стесняйтесь использовать термины из области информатики, названия алгоритмов и паттернов, математические термины и т.д. Не ограничивайтесь именами исключительно из пространства задачи; не заставляйте своих коллег постоянно бегать к клиенту и спрашивать, что означает каждое имя, когда соответствующая концепция уже знакома им под другим названием.

## Используйте имена из пространства задачи

Если для того, что вы делаете, не существует подходящего «программизма», используйте имя из пространства задачи. По крайней мере программист, занимающийся сопровождением кода, сможет узнать у специалиста в предметной области, что означает это имя.

Разделение концепций из пространств задачи и решения — часть работы хорошего программиста и проектировщика. В коде, главным образом ориентированном на концепции из пространства задачи, следует использовать имена из пространства задачи.

## Добавьте содержательный контекст

Лишь немногие имена содержательны сами по себе. Все остальные имена следует помещать в определенный контекст для читателя кода, заключая их в классы, функции и пространства имен с правильно выбранными названиями. В крайнем случае контекст имени можно уточнить при помощи префикса.

## Не добавляйте избыточный контекст

Если вы работаете над вымышленным приложением «Gas Station Deluxe», не стоит снабжать имя каждого класса префиксом GSD. В сущности, вы работаете против собственного инструментария. Введите букву «G», нажмите клавишу завершения — и вы получите длинный-предлинный список всех классов в системе. Разумно ли это? IDE пытается помочь вам, так стоит ли ей мешать?

Короткие имена обычно лучше длинных, если только их смысл понятен читателю кода. Не включайте в имя больше контекста, чем необходимо.

Имена accountAddress и customerAddress хорошо подходят для экземпляров класса Address, но для классов такой выбор неудачен. Address — вот хорошее имя класса. Если потребуется подчеркнуть различия между MAC-адресами, адресами портов и веб-адресами, я подумаю об использовании имен PostalAddress, MAC и URI. Полученные имена становятся более точными, а это, собственно, и является главной целью всего присваивания имен.

# Функции

## Компактность

Функции должны быть как можно более компактными.

Желательно, чтобы длина функции не превышала 20 строк.

Блоки в командах if, else, while и т.д. должны состоять из одной строки, в которой обычно содержится вызов функции. Это не только делает вмещающую функцию более компактной, но и способствует документированию кода, поскольку вызываемой в блоке функции можно присвоить удобное содержательное имя.

Кроме того, функции не должны содержать вложенных структур, так как это ­приводит к их увеличению. Максимальный уровень отступов в функции не должен превышать одного-двух. Разумеется, это упрощает чтение и понимание функций.

## Правило одной операции

ФУНКЦИЯ ДОЛЖНА ВЫПОЛНЯТЬ ТОЛЬКО ОДНУ ОПЕРАЦИЮ. ОНА ДОЛЖНА ВЫПОЛНЯТЬ ЕЕ ХОРОШО. И НИЧЕГО ДРУГОГО ОНА ДЕЛАТЬ НЕ ДОЛЖНА.

Проблема в том, что иногда бывает трудно определить, что же считать «одной операцией». Если функция выполняет только те действия, которые находятся на одном уровне под объявленным именем функции, то эта функция выполняет одну операцию.

Чтобы определить, что функция выполняет более одной операции, попробуйте извлечь из нее другую функцию, которая бы не являлась простой переформулировкой реализации.

Функцию, выполняющую только одну операцию, невозможно осмысленно разделить на секции.

## Один уровень абстракции на функцию

Чтобы убедиться в том, что функция выполняет «только одну операцию», необходимо проверить, что все команды функции находятся на одном уровне абстракции.

Смешение уровней абстракции внутри функции всегда создает путаницу. Читатель не всегда понимает, является ли некоторое выражение важной концепцией или второстепенной подробностью. Что еще хуже, при их смешении функция постепенно начинает обрастать все большим количеством второстепенных подробностей.

Код должен читаться как рассказ — сверху вниз.

За каждой функцией должны следовать функции следующего уровня абстракции. Это позволяет читать код, последовательно спускаясь по уровням абстракции в ходе чтения списка функций. Я называю такой подход «правилом понижения».

Сказанное можно сформулировать и иначе: программа должна читаться так, словно она является набором TO-абзацев, каждый из которых описывает текущий уровень абстракции и ссылается на последующие TO-абзацы следующего нижнего уровня.

* Чтобы включить начальные и конечные блоки, мы сначала включаем начальные блоки, затем содержимое тестовой страницы, а затем включаем конечные блоки.
* Чтобы включить начальные блоки, мы сначала включаем пакетные начальные блоки, если имеем дело с пакетом тестов, а затем включаем обычные начальные блоки.
* Чтобы включить пакетные начальные блоки, мы ищем в родительской иерархии страницу SuiteSetUp и добавляем команду include с путем к этой странице.
* Чтобы найти в родительской иерархии…

## Используйте содержательные имена

Трудно пере­оценить пользу хороших имен. Вспомните принцип Уорда: «Вы работаете с чистым кодом, если каждая функция в основном делает то, что вы от нее ожидали». Половина усилий по реализации этого принципа сводится к выбору хороших имен для компактных функций, выполняющих одну операцию. Чем меньше и специализированнее функция, тем проще выбрать для нее содержательное имя.

Не бойтесь использовать длинные имена. Длинное содержательное имя лучше короткого невразумительного. Выберите схему, которая позволяет легко прочитать слова в имени функции, а затем составьте из этих слов имя, которое описывает назначение функции.

Не бойтесь расходовать время на выбор имени. Опробуйте несколько разных имен и посмотрите, как читается код с каждым из вариантов. В современных рабочих средах (таких, как Eclipse и IntelliJ) задача смены имени решается тривиально. Используйте одну из этих сред и поэкспериментируйте с разными именами, пока не найдете самое содержательное.

Выбор содержательных имен прояснит архитектуру модуля и поможет вам усовершенствовать ее. Нередко поиски хороших имен приводят к полезной реструктуризации кода.

Будьте последовательны в выборе имен. Используйте в именах функций те же словосочетания, глаголы и существительные, которые используются в ваших модулях. Для примера можно взять имена includeSetupAndTeardownPages, includeSetupPages, includeSuiteSetupPage и includeSetupPage. Благодаря единой фразеологии эти имена рассказывают связную историю.

## Аргументы функций

В идеальном случае количество аргументов функции равно нулю (нуль-арная функция). Далее следуют функции с одним аргументом (унарные) и с двумя аргументами (бинарные). Функций с тремя аргументами (тернарных) следует по возможности избегать. Необходимость функций с большим количеством аргументов (полиарных) должна быть подкреплена очень вескими доводами — и все равно такие функции лучше не использовать.

Аргументы усложняют функции и лишают их значительной части концептуальной мощи. Аргумент и имя функции находятся на разных уровнях абстракции, а читателю приходится помнить о подробностях, которые на данный момент не особенно важны.

Аргументы создают еще больше проблем с точки зрения тестирования. Только представьте, как трудно составить все тестовые сценарии, проверяющие правильность работы кода со всеми комбинациями аргументов. Если аргументов нет — задача тривиальна. При одном аргументе все обходится без особых сложностей. С двумя аргументами ситуация усложняется. Если же аргументов больше двух, задача тестирования всех возможных комбинаций выглядит все более устрашающе.

Выходные аргументы запутывают ситуацию еще быстрее, чем входные. Читая код функции, мы обычно предполагаем, что функция получает информацию в аргументах, и выдает ее в возвращаемом значении. Как правило, никто не ожидает, что функция будет возвращать информацию в аргументах. Таким образом, выходные аргументы часто заставляют нас браться за чтение функции заново.

Если уж обойтись без аргументов никак не удается, постарайтесь хотя бы ограничиться одним входным аргументом.

**Стандартные унарные формы**

Существует два очень распространенных случая вызова функции с одним аргументом. Первая — проверка некоторого условия, связанного с аргументом, как в вызове boolean fileExists("MyFile"). Вторая — обработка аргумента, его преобразование и возвращение. Например, вызов InputStream fileOpen("MyFile") преобразует имя файла в формате String в возвращаемое значение InputStream. Выбирайте имена, которые четко отражают различия, и всегда используйте две формы в логически непротиворечивом контексте.

Несколько менее распространенным, но все равно очень полезным частным случаем функции с одним аргументом является событие. В этой форме имеется входной аргумент, а выходного аргумента нет. Предполагается, что программа интерпретирует вызов функции как событие и использует аргумент для изменения состояния системы, например, void passwordAttemptFailedNtimes(int attempts). Будьте внимательны при использовании данной формы. Читателю должно быть предельно ясно, что перед ним именно событие. Тщательно выбирайте имена и контексты.

Старайтесь избегать унарных функций, не относящихся к этим формам, например void includeSetupPageInto(StringBuffer pageText). Преобразования, в которых вместо возвращаемого значения используется выходной аргумент, сбивают читателя с толку. Если функция преобразует свой входной аргумент, то результат должен передаваться в возвращаемом значении.

**Аргументы-флаги**

Аргументы-флаги уродливы. Передача логического значения функции — воистину ужасная привычка. Она немедленно усложняет сигнатуру ­метода, громко провозглашая, что функция выполняет более одной операции. При истинном значении флага выполняется одна операция, а при ложном — другая!

**Бинарные функции**

Функцию с двумя аргументами понять сложнее, чем унарную функцию. Бинарные функции не являются абсолютным злом, и вам почти наверняка придется писать их. Тем не менее следует помнить, что за их использование приходится расплачиваться, а вам стоит воспользоваться всеми доступными средствами для их преобразования в унарные.

**Тернарные функции**

Разобраться в функции с тремя аргументами значительно сложнее, чем в бинарной функции. Проблемы соблюдения порядка аргументов, приостановки чтения и игнорирования увеличиваются более чем вдвое. Я рекомендую очень хорошо подумать, прежде чем создавать тернарную функцию.

**Объекты как аргументы**

Если функция должна получать более двух или трех аргументов, весьма вероятно, что некоторые из этих аргументов стоит упаковать в отдельном классе. Сокращение количества аргументов посредством создания объектов может показаться жульничеством, но это не так. Если переменные передаются совместно как единое целое, то, скорее всего, вместе они образуют концепцию, заслуживающую собственного имени.

**Списки аргументов**

Иногда функция должна получать переменное количество аргументов. Для примера возьмем метод String.format:

String.format("%s worked %.2f hours.", name, hours);

Если все переменные аргументы считаются равноправными, как в этом примере, то их совокупность эквивалентна одному аргументу типа List. По этой причине функция String.format фактически является бинарной. Следовательно, в данном случае действуют уже знакомые правила. Функции с переменным списком аргументов могут быть унарными, бинарными и даже тернарными, но использовать большее количество аргументов было бы ошибкой.

**Глаголы и ключевые слова**

Выбор хорошего имени для функции способен в значительной мере объяснить смысл функции, а также порядок и смысл ее аргументов. В унарных функциях сама функция и ее аргумент должны образовывать естественную пару «глагол/существительное». Например, вызов вида write(name) смотрится весьма информативно. Читатель понимает, что чем бы ни было «имя» (name), оно куда-то «записывается» (write). Еще лучше запись writeField(name), которая сообщает, что «имя» записывается в «поле» какой-то структуры.

## Избавьтесь от побочных эффектов

Побочные эффекты суть ложь. Ваша функция обещает делать что-то одно, но делает что-то другое, скрытое от пользователя. Иногда она вносит неожиданные изменения в переменные своего класса — скажем, присваивает им значения параметров, переданных функции, или глобальных переменных системы. В любом случае такая функция является коварной и вредоносной ложью, которая часто приводит к созданию противоестественных временных привязок и других зависимостей.

**Выходные аргументы**

Аргументы естественным образом интерпретируются как входные данные функции. Каждый, кто занимался программированием более нескольких лет, наверняка сталкивался с необходимостью дополнительной проверки аргументов, которые на самом деле оказывались выходными, а не входными. Пример:

appendFooter(s);

Присоединяет ли эта функция s в качестве завершающего блока к чему-то другому?  Или она присоединяет какой-то завершающий блок к s? Является ли s входным или выходным аргументом? Конечно, можно посмотреть на сигнатуру функции и получить ответ:

public void appendFooter(StringBuffer report)

Вопрос снимается, но только после проверки объявления. Все, что заставляет обращаться к сигнатуре функции, нарушает естественный ритм чтения кода. Подобных «повторных заходов» следует избегать.

До наступления эпохи объектно-ориентированного программирования без выходных аргументов иногда действительно не удавалось обойтись. Но в ОО-языках эта проблема в целом исчезла, потому что сама функция может вызываться для выходного аргумента. Иначе говоря, функцию appendFooter лучше вызывать в виде

report.appendFooter();

В общем случае выходных аргументов следует избегать. Если ваша функция должна изменять чье-то состояние, пусть она изменяет состояние своего объекта-владельца.

## Разделение команд и запросов

Функция должна что-то делать или отвечать на какой-то вопрос, но не одновременно. Либо функция изменяет состояние объекта, либо возвращает информацию об этом объекте. Совмещение двух операций часто создает путаницу.

## Используйте исключения вместо возвращения кодов ошибок

Возвращение кодов ошибок функциями-командами является неочевидным нарушением принципа разделения команд и запросов. Оно поощряет использование команд в предикатных выражениях if:

if (deletePage(page) == E\_OK)

Такие конструкции не страдают от смешения глаголов с прилагательными, но они приводят к созданию структур слишком глубокой вложенности. При возвращении кода ошибки возникает проблема: вызывающая сторона должна немедленно отреагировать на ошибку.

С другой стороны, если вместо возвращения кодов ошибок используются исключения, то код обработки ошибок изолируется от ветви нормального выполнения и упрощается.

**Изолируйте блоки try/catch**

Блоки try/catch выглядят весьма уродливо. Они запутывают структуру кода и смешивают обработку ошибок с нормальной обработкой. По этой причине тела блоков try и catch рекомендуется выделять в отдельные функции.

**Обработка ошибок как одна операция**

Функции должны выполнять одну операцию. Обработка ошибок — это одна операция. Значит, функция, обрабатывающая ошибки, ничего другого делать не должна. Отсюда следует, что если в функции присутствует ключевое слово try, то оно должно быть первым словом в функции, а после блоков catch/finally ничего другого быть не должно.

**Магнит зависимостей Error.java**

Возвращение кода ошибки обычно подразумевает, что в программе имеется некий класс или перечисление, в котором определяются все коды ошибок.

public enum Error {

  OK,

  INVALID,

  NO\_SUCH,

  LOCKED,

  OUT\_OF\_RESOURCES,

  WAITING\_FOR\_EVENT;

}

Подобные классы называются магнитами зависимостей; они должны импортироваться и использоваться многими другими классами. При любых изменениях перечисления Error все эти классы приходится компилировать и развертывать заново. Это обстоятельство создает негативную нагрузку на класс Error. Программистам не хочется добавлять новые ошибки, чтобы не создавать себе проблем со сборкой и развертыванием. Соответственно, вместо добавления новых кодов ошибок они предпочитают использовать старые.

Если вместо кодов ошибок использовать исключения, то новые исключения определяются производными от класса исключения. Их включение в программу не требует перекомпиляции или повторного развертывания.

## Не повторяйтесь

Дублирование создает проблемы, потому что оно увеличивает объем кода, а при изменении алгоритма вам придется вносить изменения сразу в нескольких местах. Также в несколько раз возрастает вероятность ошибки.

Дублирование иногда считается корнем всего зла в программировании.

## Структурное программирование

Некоторые программисты следуют правилам структурного программирования, изложенным Эдгаром Дейкстрой. Дейкстра считает, что каждая функция и каждый блок внутри функции должны иметь одну точку входа и одну точку выхода. Выполнение этого правила означает, что функция должна содержать только одну команду return, в циклах не должны использоваться команды break или continue, а команды goto не должны использоваться никогда и ни при каких условиях.

Хотя мы с симпатией относимся к целям и методам структурного программирования, в очень компактных функциях эти правила не приносят особой пользы. Только при увеличении объема функций их соблюдение обеспечивает существенный эффект.

Итак, если ваши функции остаются очень компактными, редкие вкрапления множественных return, команд break и continue не принесут вреда, а иногда даже повышают выразительность по сравнению с классической реализацией с одной точкой входа и одной точкой выхода. С другой стороны, команда goto имеет смысл только в больших функциях, поэтому ее следует избегать.

## Как научиться писать такие функции?

Написание программ сродни любому другому виду письменной работы. Когда вы пишете статью или доклад, вы сначала излагаете свои мысли, а затем «причесываете» их до тех пор, пока они не будут хорошо читаться. Первый вариант ­может быть неуклюжим и нелогичным; вы переделываете, дополняете и уточняете его, пока он не будет читаться так, как вам хочется.

Когда я пишу свои функции, они получаются длинными и сложными. В них встречаются многоуровневые отступы и вложенные циклы. Они имеют длинные списки аргументов. Имена выбираются хаотично, а в коде присутствуют дубликаты. Но у меня также имеется пакет модульных тестов для всех этих неуклюжих строк до последней.

Итак, я начинаю «причесывать» и уточнять свой код, выделять новые функции, изменять имена и устранять дубликаты. Я сокращаю методы и переупорядочиваю их. Иногда приходится ломать целые классы, но при этом слежу за тем, чтобы все тесты выполнялись успешно.

В конечном итоге у меня остаются функции, построенные по правилам, изложенным в этой главе. Я не записываю их так с самого начала. И вообще не думаю, что кому-нибудь это под силу.

## Завершение

Каждая система строится в контексте языка, отражающего специфику предметной области и разработанного программистами для описания этой системы. В этом языке функции играют роль глаголов, а классы — существительных. Не стоит полагать, что мы возвращаемся к кошмарной древней практике, по которой существительные и глаголы в документе с требованиями становились первыми кандидатами для классов и функций системы. Скорее речь идет о гораздо более древней истине. Искусство программирования является (и всегда было) искусством языкового проектирования.

Опытные программисты рассматривают систему как историю, которую они должны рассказать, а не как программу, которую нужно написать. Они используют средства выбранного ими языка программирования для конструирования гораздо более богатого и выразительного языка, подходящего для этого повествования. Частью этого предметно-ориентированного языка является иерархия функций, которые описывают все действия, выполняемые в рамках системы. В результате искусной рекурсии эти действия формулируются на том самом предметно-ориентированном языке, который они определяют для изложения своей маленькой части истории.

# Комментарии

Грамотное применение комментариев должно компенсировать нашу неудачу в выражении своих мыслей в коде. Комментарий — всегда признак неудачи. Мы вынуждены использовать комментарии, потому что нам не всегда удается выразить свои мысли без них, однако гордиться здесь нечем.

Итак, вы оказались в ситуации, в которой необходимо написать комментарий? Хорошенько подумайте, нельзя ли пойти по другому пути и выразить свои намерения в коде.

Почему я так настроен против комментариев? Потому что они лгут. Не всегда и не преднамеренно, но это происходит слишком часто. Чем древнее комментарий, чем дальше он расположен от описываемого им кода, тем больше вероятность того, что он просто неверен. Причина проста: программисты не могут нормально сопровождать комментарии.

Неточные комментарии гораздо вреднее, чем полное отсутствие комментариев. Они обманывают и сбивают с толку. Они создают у программиста невыполнимые ожидания. Они устанавливают устаревшие правила, которые не могут (или не должны) соблюдаться в будущем.

## Комментарии не компенсируют плохого кода

Одной из распространенных причин для написания комментариев является низкое качество кода. Вы пишете модуль и видите, что код получился запутанным и беспорядочным. Вы знаете, что разобраться в нем невозможно. Поэтому вы говорите себе: «О, да это стоит прокомментировать!» Нет! Лучше исправьте свой код!

Ясный и выразительный код с минимумом комментариев гораздо лучше громоздкого, сложного кода с большим количеством комментариев. Не тратьте время на написание комментариев, объясняющих созданную вами путаницу, — лучше потратьте его на исправление.

## Хорошие комментарии

Впрочем, необходимые и полезные комментарии все же существуют.

**Представление намерений**

Иногда комментарий выходит за рамки полезной информации о реализации и описывает намерения, заложенные в решение. Возможно, вы не согласитесь с тем, как программист решает проблему, но по крайней мере вы знаете, что он пытается сделать.

**Прояснение**

Иногда смысл загадочного аргумента или возвращаемого значения бывает удобно преобразовать в удобочитаемую форму. В общем случае лучше подумать, как сделать так, чтобы этот аргумент или возвращаемое значение говорили сами за себя; но если они являются частью стандартной библиотеки или используются в коде, который вы не можете изменить, то пояснительный комментарий может быть весьма полезным.

**Предупреждения о последствиях**

Иногда бывает полезно предупредить ­других программистов о нежелательных последствиях от каких-либо действий.

**Комментарии TODO**

Иногда бывает полезно оставить заметки «на будущее» в форме комментариев //TODO. Комментарии TODO напоминают о том, что, по мнению программиста, сделать необходимо, но по какой-то причине нельзя сделать прямо сейчас.

**Усиление**

Комментарий может подчеркивать важность обстоятельства, которое на первый взгляд кажется несущественным.

## Плохие комментарии

Обычно такие комментарии представляют собой «подпорки» для некачественного кода или оправдания сомнительных решений, а их текст напоминает рассуждения вслух самого программиста.

**Бормотание**

Не стоит лепить комментарии «на скорую руку» только потому, что вам кажется, что это уместно или этого требует процесс. Любой комментарий, смысл которого приходится искать в других модулях, не несет полезной информации и не стоит битов, затраченных на его написание.

**Избыточные комментарии**

Такой комментарий не объясняет код, не предоставляет обоснований и не раскрывает намерений. Он читается не проще, чем сам код. Более того, комментарий уступает коду в точности и навязывает читателю эту неточность взамен истинного понимания.

**Недостоверные комментарии**

Иногда с самыми лучшими намерениями программист делает в комментариях заявления, неточные и не соответствующие истине.

**Не используйте комментарии там, где можно использовать функцию или переменную**

**Позиционные маркеры**

Некоторые программисты любят отмечать определенные позиции в исходных файлах. Например:

// Действия //////////////////////////////////

В отдельных случаях объединение функций под такими заголовками имеет смысл. Взгляните на дело под таким углом: заголовки привлекают внимание только в том случае, если они встречаются не слишком часто. Используйте их умеренно и только тогда, когда они приносят ощутимую пользу. При слишком частом употреблении заголовков читатель воспринимает их как фоновый шум и перестает обращать на них внимание.

**Комментарии за закрывающей фигурной скобкой**

Применение таких комментариев оправдано в длинных функциях с многоуровневой вложенностью, но они только загромождают компактные специализированные функции, которым мы отдает предпочтение. Итак, если у вас возникает желание прокомментировать закрывающие фигурные скобки, лучше постарайтесь укоротить свои функции.

**Закомментированный код**

В программировании редко встречаются привычки более отвратительные, чем закрытие комментариями неиспользуемого кода. Никогда не делайте этого! У других программистов, видящих закомментированный код, не хватает храбрости удалить его. Они полагают, что код оставлен не зря и слишком важен для удаления. В итоге закомментированный код скапливается, словно осадок на дне бутылки плохого вина.

**Нелокальная информация**

Если вы должны написать комментарий, проследите за тем, чтобы он описывал находящийся поблизости код. Не излагайте информацию системного уровня в контексте локального комментария.

**Слишком много информации**

Не включайте в комментарии интересные исторические дискуссии или опи­сания подробностей, не относящиеся к делу.

**Неочевидные комментарии**

Связь между комментарием и кодом, который он описывает, должна быть очевидной. Цель комментария — объяснить код, который не объясняет сам себя. Плохо, когда сам комментарий нуждается в объяснениях.

**Заголовки функций**

Короткие функции не нуждаются в долгих описаниях. Хорошо выбранное имя компактной функции, которая выполняет одну операцию, обычно лучше заголовка с комментарием.

# Форматирование

## Вертикальное форматирование

Достаточно серьезную систему можно построить из файлов, типичная длина которых составляет 200 строк, с верхним пределом в 500 строк. Хотя это не должно считаться раз и навсегда установленным правилом, такие показатели весьма желательны. Маленькие файлы обычно более понятны, чем большие.

**Газетная метафора**

Представьте себе хорошо написанную газетную статью. Естественно, статья читается по вертикали. В самом начале обычно располагается заголовок с общей темой статьи; он помогает вам решить, представляет ли статья интерес для вас. В первом абзаце приводится краткое изложение сюжета на уровне общих концепций, без приведения каких-либо подробностей. По мере продвижения к концу статьи объем детализации непрерывно растет, пока вы не узнаете все даты, имена, цитаты и т.д.

Исходный файл должен выглядеть как газетная статья. Имя файла должно быть простым, но содержательным. Одного имени должно быть достаточно для того, чтобы читатель понял, открыл ли он нужный модуль или нет. Начальные блоки исходного файла описывают высокоуровневые концепции и алгоритмы. Степень детализации увеличивается при перемещении к концу файла, а в самом конце собираются все функции и подробности низшего уровня в исходном файле.

**Вертикальное разделение концепций**

Практически весь код читается слева направо и сверху вниз. Каждая строка представляет выражение или условие, а каждая группа строк представляет законченную мысль. Эти мысли следует отделять друг от друга пустыми строками. Каждая пустая строка становится зрительной подсказкой, указывающей на начало новой самостоятельной концепции. В ходе просмотра листинга ваш взгляд привлекает первая строка, следующая за пустой строкой.

**Вертикальное сжатие**

Если вертикальные пропуски разделяют концепции, то вертикальное сжатие подчеркивает тесные связи. Таким образом, строки кода, между которыми существует тесная связь, должны быть «сжаты» по вертикали.

**Вертикальные расстояния**

Вам когда-нибудь доводилось метаться по классу, прыгая от одной функции к другой, прокручивая исходный файл вверх-вниз, пытаясь разобраться, как функции связаны друг с другом и как они работают, — только для того, чтобы окончательно заблудиться в его запутанных нагромождениях? Когда-нибудь искали определение функции или переменной по цепочкам наследования? Все это крайне неприятно, потому что вы стараетесь понять, как работает система, а вместо этого вам приходится тратить время и интеллектуальные усилия на поиски и запоминание местонахождения отдельных фрагментов.

Концепции, тесно связанные друг с другом, должны находиться поблизости друг от друга по вертикали. Разумеется, это правило не работает для концепций, находящихся в разных файлах. Но тесно связанные концепции и не должны находиться в разных файлах, если только это не объясняется очень вескими доводами. Кстати, это одна из причин, по которой следует избегать защищенных переменных.

Если концепции связаны настолько тесно, что они находятся в одном исходном файле, их вертикальное разделение должно показывать, насколько они важны для понимания друг друга. Не заставляйте читателя прыгать туда-сюда по исходным файлам и классам.

Объявления переменных. Переменные следует объявлять как можно ближе к месту использования. Управляющие переменные циклов обычно объявляются внутри конструкции цикла. В отдельных случаях переменная может объявляться в начале блока или непосредственно перед циклом в длинной функции. Переменные экземпляров, напротив, должны объявляться в начале класса. Это не увеличивает вертикальное расстояние между переменными, потому что в хорошо спроектированном классе они используются многими, если не всеми, методами класса.

В Java переменные обычно размещаются в начале класса.

Зависимые функции. Если одна функция вызывает другую, то эти функции должны располагаться вблизи друг от друга по вертикали, а вызывающая функция должна находиться над вызываемой (если это возможно). Тем самым формируется естественная структура программного кода. Если это правило будет последовательно соблюдаться, читатели кода будут уверены в том, что определения функций следуют неподалеку от их вызовов.

Концептуальное родство. Некоторые фрагменты кода требуют, чтобы их разместили вблизи от других фрагментов. Такие фрагменты обладают определенным концептуальным родством. Чем сильнее родство, тем меньше должно быть вертикальное расстояние между ними.

**Вертикальное упорядочение**

Как правило, взаимозависимые функции должны размещаться в нисходящем порядке. Иначе говоря, вызываемая функция должна располагаться ниже вызывающей функции. Так формируется логичная структура модуля исходного кода – от высокого уровня к более низкому.

Как и в газетных статьях, читатель ожидает, что самые важные концепции будут изложены сначала, причем с минимальным количеством второстепенных де­талей. Низкоуровневые подробности естественно приводить в последнюю ­очередь. Это позволяет нам бегло просматривать исходные файлы, извлекая суть из нескольких начальных функций, без погружения в подробности.

## Горизонтальное форматирование

Насколько широкой должна быть строка? Строки лучше делать по возможности короткими. Установленное Холлеритом старое ограничение в 80 символов выглядит излишне жестким; я ничего не имеют против строк длиной в 100 и даже 120 символов. Но более длинные строки, вероятно, вызваны небрежностью программиста. Лично я установил себе «верхнюю планку» в 120 символов.

**Горизонтальное разделение и сжатие**

Горизонтальные пропуски используются для группировки взаимосвязанных ­элементов и разделения разнородных элементов.

Знаки присваивания окружены пробелами, обеспечивающими их визуальное выделение. Операторы присваивания состоят из двух основных элементов: левой и правой частей. Пробелы наглядно подчеркивают это разделение.

С другой стороны, я не стал отделять имена функций от открывающих скобок. Это обусловлено тем, что имя функции тесно связано с ее аргументами. Пробелы изолируют их вместо того, чтобы объединять. Я также разделил аргументы в скобках пробелами, чтобы выделить запятые и подчеркнуть, что аргументы не зависят друг от друга.

Пробелы также применяются для визуального обозначения приоритета операторов. Между множителями нет пробелов, потому что они обладают высоким приоритетом. Слагаемые разделяются пробелами, так как сложение и вычитание имеют более низкий приоритет.

**Отступы**

Исходный файл имеет иерархическую структуру. В нем присутствует информация, относящаяся к файлу в целом; к отдельным классам в файле; к методам ­внутри классов; к блокам внутри методов и рекурсивно – к блокам внутри блоков. Каждый уровень этой иерархии образует область видимости, в которой могут объявляться имена и в которой интерпретируются исполняемые команды.

Чтобы создать наглядное представление этой иерархии, мы снабжаем строки исходного кода отступами, размер которых соответствует их позиции в иерархии. Команды уровня файла (такие, как большинство объявлений классов) отступов не имеют. Методы в классах сдвигаются на один уровень вправо от уровня класса. Реализации этих методов сдвигаются на один уровень вправо от объявления класса. Реализации блоков сдвигаются на один уровень вправо от своих внешних блоков и т.д.

Программисты широко используют эту схему расстановки отступов в своей работе. Чтобы определить, к какой области видимости принадлежат строки кода, они визуально группируют строки по левому краю. Это позволяет им быстро пропускать области видимости, не относящиеся к текущей ситуации (например, реализации команд if и while). У левого края ищутся объявления новых методов, новые переменные и даже новые классы. Без отступов программа становится практически нечитаемой для людей.

Наше зрение быстро охватывает структуру файла с отступами. Мы почти мгновенно находим переменные, конструкторы и методы. Всего за несколько секунд можно понять, что класс предоставляет простой интерфейс для работы с сокетом, с поддержкой тайм-аута. С другой стороны, разобраться в версии без отступов без тщательного анализа практически невозможно.

Нарушения отступов. Иногда возникает соблазн нарушить правила расстановки отступов в коротких командах if, коротких циклах while или коротких функциях. Но каждый раз, когда я поддавался этому искушению, я почти всегда возвращался и расставлял отступы, как положено.

**Вырожденные области видимости**

Иногда тело цикла while  или команды for не содержит команд, то есть является вырожденным. Я не люблю такие структуры и стараюсь избегать их. А когда это невозможно, я по крайней мере слежу за тем, чтобы пустое тело имело правильные отступы и было заключено в фигурные скобки. Вы не представляете, как часто меня обманывала точка с запятой, молчаливо прячущаяся в конце цикла while в той же строке. Если не сделать эту точку хорошо заметной, разместив ее в отдельной строке, ее попросту слишком сложно разглядеть:

while (dis.read(buf, 0, readBufferSize) != -1)

  ;

# Объекты и структуры данных

Существует веская причина для ограничения доступа к переменным в программах: мы не хотим, чтобы другие программисты зависели от них. Мы хотим иметь возможность свободно менять тип или реализацию этих переменных так, как считаем нужным.

## Абстракция данных

Скрытие реализации не сводится к созданию прослойки функций между переменными. Скрытие реализации направлено на формирование абстракций! Класс не просто ограничивает доступ к переменным через методы чтения/записи. Вместо этого он предоставляет абстрактные интерфейсы, посредством которых пользователь оперирует с сущностью данных. Знать, как эти данные реализованы, ему при этом не обязательно.

## Антисимметрия данных/объектов

Чем объекты отличаются от структур данных? Объекты скрывают свои данные за абстракциями и предоставляют функции, работающие с этими данными. Структуры данных раскрывают свои данные и не имеют осмысленных функций.

Обратите внимание на то, как они дополняют друг друга, фактически являясь противоположностями. В этом проявляется основополагающая дихотомия между объектами и структурами данных.

Процедурный код (код, использующий структуры данных) позволяет легко добавлять новые функции без изменения существующих структур данных. Объектно-ориентированный код, напротив, упрощает добавление новых классов без изменения существующих функций.

Обратные утверждения также истинны:

Процедурный код усложняет добавление новых структур данных, потому что оно требует изменения всех функций. Объектно-ориентированный код усложняет добавление новых функций, потому что для этого должны измениться все классы.

В любой сложной системе возникают ситуации, когда вместо новых функций в систему требуется включить новые типы данных. Для таких ситуаций объекты и объектно-ориентированное программирование особенно уместны. Впрочем, бывает и обратное — вместо новых типов данных требуется добавить новые функции. Тогда лучше подходит процедурный код и структуры данных.

## Закон Деметры

Хорошо известное эвристическое правило, называемое законом Деметры, гласит, что модуль не должен знать внутреннее устройство тех объектов, с которыми он работает. Объекты скрывают свои данные и предоставляют операции для работы с ними. Это означает, что объект не должен раскрывать свою внутреннюю структуру через методы доступа, потому что внутреннюю структуру следует скрывать.

В более точной формулировке закон Деметры гласит, что метод f класса C должен ограничиваться вызовом методов следующих объектов:

* C;
* объекты, созданные f;
* объекты, переданные f в качестве аргумента;
* объекты, хранящиеся в переменной экземпляра C.

Метод не должен вызывать методы объектов, возвращаемых любыми из разрешенных функций. Другими словами, разговаривать можно с друзьями, но не с чужаками.

## Гибриды

Гибридные структуры — наполовину объекты, наполовину структуры данных. Гибриды содержат как функции для выполнения важных операций, так и открытые переменные или открытые методы чтения/записи, которые во всех отношениях делают приватные переменные открытыми. Другим внешним функциям предлагается использовать эти переменные так, как в процедурных программах используются структуры данных.

Подобные гибриды усложняют как добавление новых функций, так и новых структур данных. Они объединяют все худшее из обеих категорий. Не используйте гибриды. Они являются признаком сумбурного проектирования, авторы которого не уверены (или еще хуже, не знают), что они собираются защищать: функции или типы.

## Объекты передачи данных

Квинтэссенцией структуры данных является класс с открытыми переменными и без функций. Иногда такие структуры называются объектами передачи данных, или DTO (Data Transfer Object). Структуры DTO чрезвычайно полезны, особенно при работе с базами данных, разборе сообщений из сокетов и т.д. С них часто начинается серия фаз преобразования низкоуровневых данных, полученных из базы, в объекты кода приложения.

**Активные записи**

Активные записи (Active Records) составляют особую разновидность DTO. Они тоже представляют собой структуры данных с открытыми переменными (или переменными с bean-доступом), но обычно в них присутствуют навигационные методы — такие, как save или find. Активные записи чаще всего являются результатами прямого преобразования таблиц баз данных или других источников данных.

К сожалению, разработчики часто пытаются интерпретировать такие структуры данных, как объекты, и включают в них методы, реализующие бизнес-логику. Однако такой подход нежелателен, так как он создает гибрид между структурой данных и объектом.

Конечно, проблема решается иначе: активные записи интерпретируются как структуры данных, а в программе создаются отдельные объекты, которые содержат бизнес-логику и скрывают свои внутренние данные (которые, возможно, представляют собой обычные экземпляры класса активной записи).

## Заключение

Объекты предоставляют поведение и скрывают данные. Это позволяет программисту легко добавлять новые виды объектов, не изменяя существующего поведения. С другой стороны, объекты усложняют добавление нового поведения к существующим объектам. Структуры данных предоставляют данные, но не обладают сколько-нибудь значительным поведением. Они упрощают добавление нового поведения в существующие структуры данных, но затрудняют добавление новых структур данных в существующие функции.

Если в некоторой системе нас прежде всего интересует гибкость в добавлении новых типов данных, то в этой части системы предпочтение отдается объектной реализации. В других случаях нам нужна гибкость расширения поведения, и тогда в этой части используются типы данных и процедуры. Хороший программист относится к этой проблеме без предубеждения и выбирает то решение, которое лучше всего подходит для конкретной ситуации.

# Обработка ошибок

## Используйте исключения  вместо кодов ошибок

В далеком прошлом многие языки программирования не поддерживали механизма обработки исключений. В таких языках возможности обработки и получения информации об ошибках были ограничены. Программа либо устанавливала флаг ошибки, либо возвращала код, который проверялся вызывающей стороной.

У обоих решений имеется общий недостаток: они загромождают код на стороне вызова. Вызывающая сторона должна проверять ошибки немедленно после вызова. По этой причине при обнаружении ошибки лучше инициировать исключение. Код вызова становится более понятным, а его логика не скрывается за кодом обработки ошибок.

## Начните с написания команды  try-catch-finally

У исключений есть одна интересная особенность: они определяют область видимости в вашей программе. Размещая код в секции try команды try-catch-finally, вы утверждаете, что выполнение программы может прерваться в любой точке, а затем продолжиться в секции catch.

Блоки try в каком-то отношении напоминают транзакции. Секция catch должна оставить программу в целостном состоянии, что бы и произошло в секции try. По этой причине написание кода, который может инициировать исключения, рекомендуется начинать с конструкции try-catch-finally. Это поможет вам определить, чего должен ожидать пользователь кода, что бы ни произошло в коде try.

Определив область видимости при помощи структуры try-catch, мы можем использовать методологию TDD для построения остальной необходимой логики.

Попробуйте написать тесты, принудительно инициирующие исключения, а затем включите в обработчик поведение, обеспечивающее прохождение тестов. Это заставит вас построить транзакционную область видимости блока try и поможет сохранить ее транзакционную природу.

## Используйте непроверяемые исключения

Java-программисты годами обсуждали преимущества и недостатки проверяемых исключений (checked exceptions). Когда проверяемые исключения появились в первой версии Java, всем казалось, что это отличная идея. В сигнатуре каждого метода должны быть перечислены все исключения, которые могут передаваться вызывающей стороне. Фактически исключения становились частью типа метода. Если сигнатура не соответствовала тому, что происходит в коде, то программа просто не компилировалась.

Но сейчас стало ясно, что проверяемые исключения не являются необходимыми для создания надежных программ. Цена проверяемых исключений — нарушение принципа открытости/закрытости [Martin]. Если вы инициируете проверяемое исключение из метода своего кода, а catch находится тремя уровнями выше, то это исключение должно быть объявлено в сигнатурах всех методов между вашим методом и catch. Следовательно, изменение на низком уровне программного продукта приводит к изменениям сигнатур на многих более высоких уровнях. Измененные модули приходится строить и развертывать заново, притом, что в программе не изменилось ничего, что было бы существенно для них.

Представьте иерархию вызовов большой системы. Функции верхнего уровня вызывают функции нижележащего уровня, которые, в свою очередь, вызывают функции низких уровней и т.д. Теперь допустим, что одна из низкоуровневых функций изменилась таким образом, что она должна инициировать исключение. Если это исключение является проверяемым, то в сигнатуру функции должна быть добавлена секция throws. Но тогда каждая функция, вызывающая нашу измененную функцию, тоже должна быть изменена с перехватом нового исключения или присоединением соответствующей секции throws к ее сигнатуре. И так до бесконечности. В итоге мы имеем каскад изменений, пробивающихся с нижних уровней программного продукта на верхние уровни! При этом нарушается инкапсуляция, потому что все функции на пути инициирования должны располагать подробной информацией об этом низкоуровневом исключении. Учитывая, что главной целью исключений является возможность обработки ошибок «на расстоянии», такое нарушение инкапсуляции проверяемыми исключениями выглядит особенно постыдно.

Проверяемые исключения иногда могут пригодиться при написании особо важных библиотек: программист обязан перехватить их. Но в общем случае разработки приложений проблемы, создаваемые зависимостями, перевешивают преимущества.

## Передавайте контекст с исключениями

Каждое исключение, инициируемое в программе, должно содержать достаточно контекстной информации для определения источника и местонахождения ошибки. В Java из любого исключения можно получить данные трассировки стека; однако по трассировке невозможно узнать, с какой целью выполнялась операция, завершившаяся неудачей.

Создавайте содержательные сообщения об ошибках и передавайте их со своими исключениями. Включайте в них сведения о сбойной операции и типе сбоя. Если в приложении ведется журнал, передайте информацию, достаточную для регистрации ошибки из секции catch.

## Определяйте классы исключений в контексте потребностей вызывающей стороны

При определении классов исключений в приложениях думать необходимо прежде всего о том, как они будут перехватываться.

Часто в определенной области кода бывает достаточно одного класса исключения. Информация, передаваемая с исключением, позволяет различать разные виды ошибок. Используйте разные классы исключений только в том случае, если вы намерены перехватывать одни исключения, разрешая прохождение других типов.

## Не возвращайте null

На мой взгляд, при любых обсуждениях обработки ошибок необходимо упомянуть о неправильных действиях программистов, провоцирующих ошибки. На первом месте в этом списке стоит возвращение null. Я видел бесчисленное множество приложений, в которых едва ли не каждая строка начиналась с проверки null.

Возвращая null, мы фактически создаем для себя лишнюю работу, а для вызывающей стороны — лишние проблемы. Стоит пропустить всего одну проверку null, и приложение «уходит в штопор».

Легко сказать, что проблемы в приведенном коде возникли из-за пропущенной проверки null. В действительности причина в другом: этих проверок слишком много. Если у вас возникает желание вернуть null из метода, рассмотрите возможность выдачи исключения или возвращения объекта «особого случая». Если ваш код вызывает метод стороннего API, способный вернуть null, создайте для него обертку в виде метода, который инициирует исключение или возвращает объект особого случая.

## Не передавайте null

Возвращать null из методов плохо, но передавать null при вызове еще хуже. По возможности избегайте передачи null в своем коде (исключение составляют разве что методы сторонних API, при вызове которых без нее не обойтись).

В большинстве языков программирования не существует хорошего способа справиться со случайной передачей null с вызывающей стороны. А раз так, разумно запретить передачу null по умолчанию. В этом случае вы будете знать, что присутствие null в списке аргументов свидетельствует о возникшей проблеме; это будет способствовать уменьшению количества ошибок, сделанных по неосторожности.

# Модульные тесты

## Три закона TTD

Рассмотрим следующие три закона:

* Первый закон. Не пишите код продукта, пока не напишете отказной модульный тест.
* Второй закон. Не пишите модульный тест в объеме большем, чем необходимо для отказа. Невозможность компиляции является отказом.
* Третий закон. Не пишите код продукта в объем большем, чем необходимо для прохождения текущего отказного теста.

Эти три закона устанавливают рамки рабочего цикла, длительность которого составляет, вероятно, около 30 секунд. Тесты и код продукта пишутся вместе, а тесты на несколько секунд опережают код продукта.

## О чистоте тестов

Тесты «на скорую руку» равносильны полному отсутствию тестов, если не хуже. Дело в том, что тесты должны изменяться по мере развития кода продукта. Чем примитивнее тесты, тем труднее их изменять. Если тестовый код сильно запутан, то может оказаться, что написание нового кода продукта займет меньше времени, чем попытки втиснуть новые тесты в обновленный пакет. При изменении кода продукта старые тесты перестают проходить, а неразбериха в тестовом коде не позволяет быстро разобраться с возникшими проблемами. Таким образом, тесты начинают рассматриваться как постоянно растущий балласт.

Мораль проста: тестовый код не менее важен, чем код продукта. Не считайте его «кодом второго сорта». К написанию тестового кода следует относиться вдумчиво, внимательно и ответственно. Тестовый код должен быть таким же чистым, как и код продукта.

Если не поддерживать чистоту своих тестов, то вы их лишитесь. А без тестов утрачивается все то, что обеспечивает гибкость кода продукта. Именно модульные тесты обеспечивают гибкость, удобство сопровождения и возможность повторного использования нашего кода. Это объясняется просто: если у вас есть тесты, вы не боитесь вносить изменения в код! Без тестов любое изменение становится потенциальной ошибкой.

Итак, наличие автоматизированного пакета модульных тестов, охватывающих код продукта, имеет важнейшее значение для чистоты и ясности архитектуры. А причина заключается в том, что тесты обеспечивают возможность внесения изменения.

Таким образом, если ваши тесты недостаточно чисты и проработаны, ваши возможности по изменению кода сокращаются и вы лишаетесь возможности улучшения структуры кода. Некачественные тесты приводит к некачественному коду продукта. В конечном итоге тестирование вообще становятся невозможным, и код продукта начинает загнивать.

## Чистые тесты

Какими отличительными признаками характеризуется чистый тест? Тремя: удобочитаемостью, удобочитаемостью и удобочитаемостью. Вероятно, удобочитаемость в модульных тестах играет еще более важную роль, чем в коде продукта. Что делает тестовый код удобочитаемым? То же, что делает удобочитаемым любой другой код: ясность, простота и выразительность. В тестовом коде необходимо передать максимум информации минимумом выразительных средств.

В структуре тестов может быть воплощен паттерн ПОСТРОЕНИЕ-ОПЕРАЦИИ-ПРОВЕРКА. Каждый тест четко делится на три части. Первая часть строит тестовые данные, вторая часть выполняет операции с тестовыми данными, а третья часть проверяет, что операция привела к ожидаемым результатам.

**Предметно-ориентированный  язык тестирования**

Методика построения предметно-ориентированного языка для программирования тестов. Вместо вызова функций API, используемых программистами для манипуляций с системой, мы строим набор функций и служебных программ, использующих API; это упрощает написание и чтение тестов. Наши функции и служебные программы образуют специализированный API, то есть по сути — язык тестирования, который программисты используют для упрощения работы над тестами, а также чтобы помочь другим программистам, которые будут читать эти тесты позднее.

Тестовый API не проектируется заранее; он развивается на базе многократной переработки тестового кода, перегруженного ненужными подробностями.

## Одна проверка на тест

Существует точка зрения, согласно которой каждая тестовая функция в тесте JUnit должна содержать одну — и только одну — директиву assert. Тесты приводят к одному выводу, который можно быстро и легко понять при чтении.

Я думаю, что правило «одного assert» является хорошей рекомендацией. Обычно я стараюсь создать предметно-ориентированный язык тестирования, который это правило поддерживает. Но при этом я не боюсь включать в свои тесты более одной директивы assert. Вероятно, лучше всего сказать, что количество директив assert в тесте должно быть сведено к минимуму.

**Одна концепция на тест**

Пожалуй, более полезное правило гласит, что в каждой тестовой функции должна тестироваться одна концепция. Так что, вероятно, лучше всего сформулировать это правило так: количество директив assert на концепцию должно быть минимальным, и в тестовой функции должна проверяться только одна концепция.

## F.I.R.S.T.

Чистые тесты должны обладать еще пятью характеристиками, названия которых образуют приведенное сокращение.

**Быстрота (Fast).** Тесты должны выполняться быстро. Если тесты выполняются медленно, вам не захочется часто запускать их. Без частого запуска тестов проблемы не будут выявляться на достаточно ранней стадии, когда они особенно легко исправляются. В итоге вы уже не так спокойно относитесь к чистке своего кода, и со временем код начинает загнивать.

**Независимость (Independent).** Тесты не должны зависеть друг от друга. Один тест не должен создавать условия для выполнения следующего теста. Все тесты должны выполняться независимо и в любом порядке на ваше усмотрение. Если тесты зависят друг от друга, то при первом отказе возникает целый каскад сбоев, который усложняет диагностику и скрывает дефекты в зависимых тестах.

**Повторяемость (Repeatable).** Тесты должны давать повторяемые результаты в любой среде. Вы должны иметь возможность выполнить тесты в среде реальной эксплуатации, в среде тестирования или на вашем ноутбуке во время возвращения домой с работы. Если ваши тесты не будут давать однозначных результатов в любых условиях, вы всегда сможете найти отговорку для объяснения неудач. Также вы лишитесь возможности проводить тестирование, если нужная среда недоступна.

**Очевидность (Self-Validating).** Результатом выполнения теста должен быть логический признак. Тест либо прошел, либо не прошел. Чтобы узнать результат, пользователь не должен читать журнальный файл. Не заставляйте его вручную сравнивать два разных текстовых файла. Если результат теста не очевиден, то отказы приобретают субъективный характер, а выполнение тестов может потребовать долгой ручной обработки данных.

**Своевременность (Timely).** Тесты должны создаваться своевременно. Модульные тесты пишутся непосредственно перед кодом продукта, обеспечивающим их прохождение. Если вы пишете тесты после кода продукта, вы можете решить, что тестирование кода продукта создает слишком много трудностей, а все из-за того, что удобство тестирования не учитывалось при проектировании кода продукта.

# Классы

## Строение класса

По стандартным правилам Java класс должен начинаться со списка переменных. Сначала перечисляются открытые статические константы. Далее следуют приватные статические переменные, а за ними идут приватные переменные экземпляров. Открытых переменных обычно нет, трудно найти веские причины для их использования.

За списком переменных обычно следуют открытые функции. Мы предпочитаем размещать приватные вспомогательные функции, вызываемые открытыми функциями, непосредственно за самой открытой функцией. Такое размещение соответствует правилу понижения, в результате чего программа читается как газетная статья.

### Инкапсуляция

Мы предпочитаем объявлять переменные и вспомогательные функции приватными, но относимся к ним без фанатизма. Иногда переменную или вспомогательную функцию приходится объявлять защищенной, чтобы иметь возможность обратиться к ней из теста. С нашей точки зрения тесты исключительно важны. Если тест из того же пакета должен вызвать функцию или обратиться к переменной, мы используем защищенный или пакетный уровень доступа. Тем не менее начинать следует с поиска способа, сохраняющего приватность. Ослабление инкапсуляции всегда должно быть последней мерой.

### Классы должны быть компактными!

Как и в случае с функциями, компактность должна стать основным правилом проектирования классов. И для классов начинать следует с вопроса: «А насколько компактными?»

Размер функций определяется количеством физических строк. В классах используется другая метрика; мы подсчитываем ответственности.

Имя класса должно описывать его ответственности. В сущности, имя должно стать первым фактором, способствующим определению размера класса. Если для класса не удается подобрать четкое, короткое имя, вероятно, он слишком велик. Чем туманнее имя класса, тем больше вероятность, что он имеет слишком много ответственностей. В частности, присутствие в именах классов слов-проныр «Processor», «Manager» и «Super» часто свидетельствует о нежелательном объединении ответственностей.

Краткое описание класса должно укладываться примерно в 25 слов, без выражений «если», «и», «или» и «но». Как бы вы описали класс SuperDashboard? «Класс предоставляет доступ к компоненту, который последним имел фокус ввода, и позволяет отслеживать номера версии и сборки». Первое «и» указывает на то, что SuperDashboard имеет слишком много ответственностей.

### Принцип единой ответственности (SRP)

Принцип единой ответственности утверждает, что класс или модуль должен иметь одну — и только одну — причину для изменения. Этот принцип дает нам как определение ответственности, так и критерий для оценки размера класса. Классы должны иметь одну ответственность, то есть одну причину для изменений.

Попытки идентификации ответственностей (причин для изменения) часто помогают выявить и создать более качественные абстракции для нашего кода.

Еще раз выделю основные моменты: система должна состоять из множества мелких классов, а не из небольшого числа больших. Каждый класс инкапсулирует одну ответственность, имеет одну причину для изменения и взаимодействует с другими классами для реализации желаемого поведения системы.

### Связность

Классы должны иметь небольшое количество переменных экземпляров. Каждый метод класса должен оперировать с одной или несколькими из этих переменных. В общем случае, чем с большим количеством переменных работает метод, тем выше связность этого метода со своим классом. Класс, в котором каждая переменная используется каждым методом, обладает максимальной связностью.

Связность класса должна быть высокой. Высокая связность означает, что методы и переменные класса взаимозависимы и существуют как единое целое.

Стратегия компактных функций и коротких списков параметров иногда приводит к росту переменных экземпляров, используемых подмножеством методов. Это почти всегда свидетельствует о том, что по крайней мере один класс пытается выделиться из более крупного класса. Постарайтесь разделить переменные и методы на два и более класса, чтобы новые классы обладали более высокой связностью.

### Поддержание связности приводит к уменьшению классов

Сам акт разбиения больших функций на меньшие приводит к росту количества классов. Если группа функций должна работать с некоторыми переменными, не образуют ли они класс сами по себе? Конечно, образуют. Если классы утрачивают связность, разбейте их!

Таким образом, разбиение большой функции на много мелких функций также часто открывает возможность для выделения нескольких меньших классов. В результате строение программы улучшается, а ее структура становится более прозрачной.

## Структурирование с учетом изменений

Большинство систем находится в процессе непрерывных изменений. Каждое изменение создает риск того, что остальные части системы будут работать не так, как мы ожидаем. В чистой системе классы организованы таким образом, чтобы риск от изменений был сведен к минимуму.

Структура системы должна быть такой, чтобы обновление системы (с добавлением новых или изменением существующих аспектов) создавало как можно меньше проблем. В идеале новая функциональность должна реализовываться расширением системы, а не внесением изменений в существующий код.

**Изоляция изменений**

Потребности меняются со временем; следовательно, меняется и код. В начальном курсе объектно-ориентированного программирования мы узнали, что классы делятся на конкретные, содержащие подробности реализации (код), и абстрактные, представляющие только концепции. Если клиентский класс зависит от конкретных подробностей, то изменение этих подробностей может нарушить его работоспособность. Чтобы изолировать воздействие этих подробностей на класс, в систему вводятся интерфейсы и абстрактные классы.

Если система обладает достаточной логической изоляцией для подобного тестирования, она также становится более гибкой и более подходящей для повторного использования. Отсутствие жестких привязок означает, что элементы системы лучше изолируются друг от друга и от изменений. Изоляция упрощает понимание каждого элемента системы.

Сведение к минимуму логических привязок соответствует другому принципу проектирования классов, известному как принцип обращения зависимостей (DIP, Dependency Inversion Principle). По сути DIP гласит, что классы системы должны зависеть от абстракций, а не от конкретных подробностей.

# Формирование архитектуры

## Четыре правила

Четыре простых правила, выполнение которых помогало бы повысить качество проектирования. Многие полагают, что четыре правила простой архитектуры [XPE] Кента Бека оказывают значительную помощь в проектировании программных продуктов. Согласно Кенту, архитектура может считаться «простой», если она:

* обеспечивает прохождение всех тестов,
* не содержит дублирующегося кода,
* выражает намерения программиста,
* использует минимальное количество классов и методов.

**Правило № 1: выполнение всех тестов**

Прежде всего система должна делать то, что задумано ее проектировщиком. Система может быть отлично спланирована «на бумаге», но если не существует простого способа убедиться в том, что она действительно решает свои задачи, то результат выглядит сомнительно.

Система, тщательно протестированная и прошедшая все тесты, контролируема. На первый взгляд утверждение кажется очевидным, но это весьма важно. Невозможно проверить работу системы, которая не является контролируемой, а непроверенные системы не должны запускаться в эксплуатацию.

К счастью, стремление к контролируемости системы ведет к архитектуре с компактными узкоспециализированными классами. Все просто: классы, соответствующие принципу SRP, проще тестировать. Чем больше тестов мы напишем, тем дальше продвинемся к простоте тестирования. Таким образом, обеспечение полной контролируемости системы помогает повысить качество проектирования.

Жесткая привязка усложняет написание тестов. Таким образом, чем больше тестов мы пишем, тем интенсивнее используем такие принципы, как DIP, и такие инструменты, как внедрение зависимостей, интерфейсы и абстракции, для минимизации привязок.

Как ни удивительно, выполнение простого и очевидного правила, гласящего, что для системы необходимо написать тесты и постоянно выполнять их, влияет на соответствие системы важнейшим критериям объектно-ориентированного программирования: устранению жестких привязок и повышению связности. Написание тестов улучшает архитектуру системы.

**Правила № 2–4: переработка кода**

Когда у вас появился полный набор тестов, можно заняться чисткой кода и классов. Для этого код подвергается последовательной переработке (рефакторингу). Мы добавляем несколько строк кода, делаем паузу и анализируем новую архитектуру. Не ухудшилась ли она по сравнению с предыдущим вариантом? Если ухудшилась, то мы чистим код и тестируем его, чтобы убедиться, что в нем ничего не испорчено. Наличие тестов избавляет от опасений, что чистка кода нарушит его работу!

В фазе переработки применяется абсолютно все, что вы знаете о качественном проектировании программных продуктов. В ход идут любые приемы: повышение связности, устранение жестких привязок, разделение ответственности, изоляция системных областей ответственности, сокращение объема функций и классов, выбор более содержательных имен и т.д. Также применяются три критерия простой архитектуры: устранение дубликатов, обеспечение выразительности и минимизация количества классов и методов.

**Отсутствие дублирования**

Дублирование — главный враг хорошо спроектированной системы. Его последствия — лишняя работа, лишний риск и лишняя избыточная сложность. Дублирование проявляется во многих формах. Конечно, точное совпадение строк кода свидетельствует о дублировании. Похожие строки часто удается «причесать» так, чтобы сходство стало еще более очевидным; это упростит рефакторинг. Кроме того, дублирование может существовать и в других формах — таких, как дублирование реализации.

Чтобы создать чистую систему, необходимо сознательно стремиться к устранению дубликатов, пусть даже всего в нескольких строках кода.

Паттерн ШАБЛОННЫЙ МЕТОД относится к числу стандартных приемов устранения высокоуровневого дублирования.

## Выразительность

Большинству читателей доводилось работать с запутанным кодом. Легко написать код, понятный для нас самих, потому что в момент его написания мы глубоко понимаем решаемую проблему. У других программистов, которые будут заниматься сопровождением этого кода, такого понимания не будет.

Основные затраты программного проекта связаны с его долгосрочным сопровождением. Чтобы свести к минимуму риск появления дефектов в ходе внесения изменений, очень важно понимать, как работает система. С ростом сложности системы разработчику приходится разбираться все дольше и дольше, а вероятность того, что он поймет что-то неправильно, только возрастает. Следовательно, код должен четко выражать намерения своего автора. Чем понятнее будет код, тем меньше времени понадобится другим программистам, чтобы разобраться в нем.

Хороший выбор имен помогает выразить ваши намерения. Имя класса или функции должно восприниматься «на слух», а когда читатель разбирается в том, что делает класс, это не должно вызывать у него удивления.

Относительно небольшой размер функций и классов также помогает выразить ваши намерения. Компактным классам и функциям проще присваивать имена; они легко пишутся и в них легко разобраться.

Стандартная номенклатура также способствует выражению намерений автора. В частности, передача информация и выразительность являются важнейшими целями для применения паттернов проектирования. Включение стандартных названий паттернов (например, КОМАНДА или ПОСЕТИТЕЛЬ) в имена классов, реализующих эти паттерны, помогает кратко описать вашу архитектуру для других разработчиков.

Хорошо написанные модульные тесты тоже выразительны. Они могут рассматриваться как разновидность документации, построенная на конкретных примерах. Читая код тестов, разработчик должен составить хотя бы общее представление о том, что делает класс.

## Минимум классов и методов

Это правило рекомендует ограничиться небольшим количеством функций и классов.

Многочисленность классов и методов иногда является результатом бессмысленного догматизма. В качестве примера можно привести стандарт кодирования, который требует создания интерфейса для каждого без исключения класса. Или разработчиков, настаивающих, что поля данных и поведение всегда должны быть разделены на классы данных и классы поведения. Избегайте подобных догм, а в своей работе руководствуйтесь более прагматичным подходом.

Наша цель — сделать так, чтобы система была компактной, но при этом одновременно сохранить компактность функций и классов. Однако следует помнить, что из четырех правил простой архитектуры это правило обладает наименьшим приоритетом. Свести к минимуму количество функций и классов важно, однако прохождение тестов, устранение дубликатов и выразительность кода все же важнее.

# Многопоточность

## Зачем нужна многопоточность?

Многопоточное программирование может рассматриваться как стратегия устранения привязок. Оно помогает отделить выполняемую операцию от момента ее выполнения. В однопоточных приложениях «что» и «когда» связаны так сильно, что просмотр содержимого стека часто позволяет определить состояние всего приложения. Программист, отлаживающий такую систему, устанавливает точку прерывания (или серию точек прерывания) и узнает состояние системы на момент остановки.

Отделение «что» от «когда» способно кардинально улучшить как производительность, так и структуру приложения. Со структурной точки зрения многопоточное приложение выглядит как взаимодействие нескольких компьютеров, а не как один большой управляющий цикл. Такая архитектура упрощает понимание системы и предоставляет мощные средства для разделения ответственности.

Но структура — не единственный аргумент для многопоточного программирования. В некоторых системах действуют ограничения по времени отклика и пропускной способности, требующие ручного кодирования многопоточных решений. Пример: допустим, система в любой момент времени работает только с одним пользователем, обслуживание которого у нее занимает всего одну секунду. При малом количестве пользователей система оперативно реагирует на все запросы, но с увеличением количества пользователей растет и время отклика. Никто не захочет стоять в очереди после 150 других пользователей! Время отклика такой системы можно было бы улучшить за счет параллельного обслуживания многих пользователей.

Или возьмем систему, которая анализирует большие объемы данных, но выдает окончательный результат только после их полной обработки. Наборы данных могут обрабатываться параллельно на разных компьютерах.

**Мифы и неверные представления**

С многопоточностью связан целый ряд распространенных мифов и неверных представлений.

* Многопоточность всегда повышает быстродействие. Действительно, многопоточность иногда повышает быстродействие, но только при относительно большом времени ожидания, которое могло бы эффективно использоваться другими потоками или процессорами.
* Написание многопоточного кода не изменяет архитектуру программы. На самом деле архитектура многопоточного алгоритма может заметно отличаться от архитектуры однопоточной системы. Отделение «что» от «когда» обычно оказывает огромное влияние на структуру системы.
* При работе с контейнером (например, веб-контейнером или EJB-контейнером) разбираться в проблемах многопоточного программирования не обязательно. В действительности желательно знать, как работает контейнер и как защититься от проблем одновременного обновления и взаимных блокировок, описанных позднее в этой главе.

Несколько более объективных утверждений, относящихся к написанию многопоточного кода:

* Многопоточность сопряжена с определенными дополнительными затратами — в отношении как производительности, так и написания дополнительного кода.
* Правильная реализация многопоточности сложна даже для простых задач.
* Ошибки в многопоточном коде обычно не воспроизводятся, поэтому они часто игнорируются как случайные отклонения (а не как систематические дефекты, которыми они на самом деле являются).
* Многопоточность часто требует фундаментальных изменений в стратегии проектирования.

## Защита от ошибок многопоточности

Далее перечислены некоторые принципы и приемы, которые помогают защитить вашу систему от проблем многопоточности.

### Принцип единой ответственности

Принцип единой ответственности (SRP) [PPP] гласит, что метод/класс/компонент должен иметь только одну причину для изменения. Многопоточные архитектуры достаточно сложны, чтобы их можно было рассматривать как причину изменения сами по себе, а следовательно, они должны отделяться от основного кода. К сожалению, подробности многопоточной реализации нередко встраиваются в другой код. Однако разработчик должен учитывать ряд факторов:

* Код реализации многопоточности имеет собственный цикл разработки, модификации и настройки.
* При написании кода реализации многопоточности возникают специфические сложности, принципиально отличающиеся от сложностей однопоточного кода (и часто превосходящие их).
* Количество потенциальных сбоев в неверно написанном многопоточном коде достаточно велико и без дополнительного бремени в виде окружающего кода приложения.

Рекомендация: отделяйте код, относящийся к реализации многопоточности, от остального кода.

### Следствие: ограничивайте область видимости данных

Два программных потока, изменяющих одно поле общего объекта, могут мешать друг другу, что приводит к непредвиденному поведению. Одно из возможных решений — защита критической секции кода, в которой происходят обращения к общему объекту, ключевым словом synchronized. Количество критических секций в коде должно быть сведено к минимуму. Чем больше в программе мест, в которых обновляются общие данные, тем с большей вероятностью:

* вы забудете защитить одно или несколько из этих мест, что приведет к нарушению работы всего кода, изменяющего общие данные.
* попытки уследить за тем, чтобы все было надежно защищено, приведут к дублированию усилий (нарушение принципа DRY).

Вам будет труднее определить источник многопоточных сбоев, который и так достаточно сложно найти.

Рекомендация: серьезно относитесь к инкапсуляции данных; жестко ограничьте доступ ко всем общим данным.

### Следствие: используйте копии данных

Как избежать нежелательных последствий одновременного доступа к данным? Например, просто не использовать его. Существуют разные стратегии: например, в одних ситуациях можно скопировать общий объект и ограничить доступ к копии (доступ только для чтения). В других ситуациях объекты копируются, результаты работы нескольких программных потоков накапливаются в копиях, а затем объединяются в одном потоке.

Если существует простой способ избежать одновременного доступа к объектам, то вероятность возникновения проблем в полученном коде значительно снижается. Вас беспокоят затраты на создание лишних объектов? Поэкспериментируйте и выясните, действительно ли она так высока. Как правило, если копирование объектов позволяет избежать синхронизации в коде, экономия на защитных блокировках быстро окупит дополнительные затраты на создание объектов и уборку мусора.

### Следствие: потоки должны быть как можно более независимы

Постарайтесь писать многопоточный код так, чтобы каждый поток существовал в собственном замкнутом пространстве и не использовал данные совместно с другими процессами. Каждый поток обрабатывает один клиентский запрос, все его данные берутся из отдельного источника и хранятся в локальных переменных. В этом случае каждый поток работает так, словно других потоков не существует, а следовательно, нет и требований к синхронизации.

Рекомендация: постарайтесь разбить данные не независимые подмножества, с которыми могут работать независимые потоки (возможно, на разных процессорах).

## Знайте свою библиотеку

Рекомендация: изучайте доступные классы. Если вы работаете на Java, уделите особое внимание пакетам java.util.concurrent, java.util.concurrent.atomic и java.util.concurrent.locks.

## Знайте модели выполнения

**Базовые определения**:

Связанные ресурсы - Ресурсы с фиксированным размером или количеством, существующие в многопоточной среде, например подключения к базе данных или буферы чтения/записи

Взаимное исключение  - В любой момент времени с общими данными или с общим ресурсом может работать только один поток

Зависание - Работа одного или нескольких потоков приостанавливается на слишком долгое время (или навсегда). Например, если высокоприоритетным потокам всегда предоставляется возможность отработать первыми, то низкоприоритетные потоки зависнут (при условии, что в системе постоянно появляются новые высокоприоритетные потоки)

Взаимная блокировка (deadlock) - Два и более потока бесконечно ожидают завершения друг друга. Каждый поток захватил ресурс, необходимый для продолжения работы другого потока, и ни один поток не может завершиться без получения захваченного другим потоком ресурса

Обратимая  блокировка (livelock) - Потоки не могут «разойтись» — каждый из потоков пытается выполнять свою работу, но обнаруживает, что другой поток стоит у него на пути. Потоки постоянно пытаются продолжить выполнение, но им это не удается в течение слишком долгого времени (или вообще не удается)

**Модель «производители-потребители»**

Один или несколько потоков-производителей создают задания и помещают их в буфер или очередь. Один или несколько потоков-потребителей извлекают ­задания из очереди и выполняют их.  Очередь между производителями и потребителями является связанным ресурсом. Это означает, что производители перед записью должны дожидаться появления свободного места в очереди, а потребители должны дожидаться появления заданий в очереди для обработки. Координация производителей и потребителей основана на передаче сигналов. Производитель записывает задание и сигнализирует о том, что очередь не пуста. Потребитель читает задание и сигнализирует о том, что очередь не заполнена. Обе стороны должны быть готовы ожидать оповещения о возможности продолжения работы.

**Модель «читатели-писатели»**

Если в системе имеется общий ресурс, который в основном служит источником информации для потоков-«читателей», но время от времени обновляется потоками-«писателями», на первый план выходит проблема оперативности обновления. Если обновление будет происходить недостаточно часто, это может привести к зависанию и накоплению устаревших данных. С другой стороны, слишком частые обновления влияют на производительность. Координация работы читателей так, чтобы они не пытались читать данные, обновляемые писателями, и наоборот, — весьма непростая задача. Писатели обычно блокируют работу многих читателей в течение долгого периода времени, а это отражается на производительности.

Проектировщик должен найти баланс между потребностями читателей и писателей, чтобы обеспечить правильный режим работы, нормальную производительность системы и избежать зависания. В одной из простых стратегий писатели дожидаются, пока в системе не будет ни одного читателя, и только после этого выполняют обновление. Однако при постоянном потоке читателей такая стратегия приведет к зависанию писателей. С другой стороны, при большом количестве высокоприоритетных писателей пострадает производительность. Поиск баланса и предотвращение ошибок многопоточного обновления — основные проблемы этой модели выполнения.

**Модель «обедающих философов»**

Представьте нескольких философов, сидящих за круглым столом. Слева у каждого философа лежит вилка, а в центре стола стоит большая тарелка спагетти. Философы проводят время в размышлениях, пока не проголодаются. Проголодавшись, философ берет вилки, лежащие по обе стороны, и приступает к еде. Для еды необходимы две вилки. Если сосед справа или слева уже использует одну из необходимых вилок, философу приходится ждать, пока сосед закончит есть и положит вилки на стол. Когда философ поест, он кладет свои вилки на стол и снова погружается в размышления.

Заменив философов программными потоками, а вилки — ресурсами, мы получаем задачу, типичную для многих корпоративных систем, в которых приложения конкурируют за ресурсы из ограниченного набора. Если небрежно отнестись к проектированию такой системы, то конкуренция между потоками может привести к возникновению взаимных блокировок, обратимых блокировок, падению производительности и эффективности работы.

## Остерегайтесь зависимостей между синхронизированными методами

Зависимости между синхронизированными методами приводят к появлению коварных ошибок в многопоточном коде. В языке Java существует ключевое слово synchronized для защиты отдельных методов. Но если общий класс содержит более одного синхронизированного метода, возможно, ваша система спроектирована неверно.

Рекомендация: избегайте использования нескольких методов одного совместно используемого объекта.

Впрочем, иногда без использования разных методов одного общего объекта обойтись все же не удается. Для обеспечения правильности работы кода в подобных ситуациях существуют три стандартных решения:

* Блокировка на стороне клиента — клиент устанавливает блокировку для сервера перед вызовом первого метода и следит за тем, чтобы блокировка распространялась на код, вызывающий последний метод.
* Блокировка на стороне сервера — на стороне сервера создается метод, который блокирует сервер, вызывает все методы, после чего снимает блокировку. Этот новый метод вызывается клиентом.
* Адаптирующий сервер — в системе создается посредник, который реализует блокировку. Ситуация может рассматриваться как пример блокировки на стороне сервера, в которой исходный сервер не может быть изменен.

## Синхронизированные секции должны иметь минимальный размер

Ключевое слово synchronized устанавливает блокировку. Все секции кода, защищенные одной блокировкой, в любой момент времени гарантированно выполняются только в одном программном потоке. Блокировки обходятся дорого, так как они создают задержки и увеличивают затраты ресурсов. Следовательно, код не должен перегружаться лишними конструкциями synchronized. С другой стороны, все критические секции должны быть защищены. Следовательно, код должен содержать как можно меньше критических секций.

Для достижения этой цели некоторые наивные программисты делают свои критические секции очень большими. Однако синхронизация за пределами минимальных критических секций увеличивает конкуренцию между потоками и снижает производительность.

Рекомендация: синхронизированные секции в ваших программах должны иметь минимальные размеры.

## О трудности корректного завершения

Написание системы, которая должна работать бесконечно, заметно отличается от написания системы, которая работает в течение некоторого времени, а затем корректно завершается.

Реализовать корректное завершение порой бывает весьма непросто. Одна из типичных проблем — взаимная блокировка программных потоков, бесконечно долго ожидающих сигнала на продолжение работы.

Представьте систему с родительским потоком, который порождает несколько дочерних потоков, а затем дожидается их завершения, чтобы освободить свои ресурсы и завершиться. Что произойдет, если один из дочерних потоков попадет во взаимную блокировку? Родитель будет ожидать вечно, и система не сможет корректно завершиться.

Рекомендация: начинайте думать о корректном завершении на ранней стадии разработки. На это может уйти больше времени, чем вы предполагаете. Проанализируйте существующие алгоритмы, потому что эта задача сложнее, чем кажется.

## Тестирование многопоточного кода

Тестирование не гарантирует правильности работы кода. Тем не менее качественное тестирование сводит риск к минимуму. Для однопоточных решений эти утверждения безусловно верны. Но как только в системе появляются два и более потока, использующие общий код и работающих с общими данными, ситуация значительно усложняется.

Рекомендация: пишите тесты, направленные на выявление существующих проблем. Часто выполняйте их для разных вариантов программных/системных конфигураций и уровней нагрузки. Если при выполнении теста происходит ошибка, обязательно найдите причину. Не игнорируйте ошибку только потому, что при следующем запуске тест был выполнен успешно.

Несколько более конкретных рекомендаций:

* Рассматривайте непериодические сбои как признаки возможных проблем многопоточности.
* Начните с отладки основного кода, не связанного с многопоточностью.
* Реализуйте логическую изоляцию конфигураций многопоточного кода.
* Обеспечьте возможность настройки многопоточного кода.
* Протестируйте программу с количеством потоков, превышающим количество процессоров.
* Протестируйте программу на разных платформах.
* Применяйте инструментовку кода для повышения вероятности сбоев.

**Рассматривайте непериодические сбои как признаки возможных проблем многопоточности**

В многопоточном коде сбои происходят даже там, где их вроде бы и быть не может. Многие разработчики (в том числе и автор) не обладают интуитивным представлением о том, как многопоточный код взаимодействует с другим кодом. Ошибки в многопоточном коде могут проявляться один раз за тысячу или даже миллион запусков. Воспроизвести такие ошибки в системе бывает очень трудно, поэтому разработчики часто склонны объяснять их «фазами Луны», случайными сбоями оборудования или другими несистематическими причинами. Однако игнорируя существование этих «разовых» сбоев, вы строите свой код на потенциально ненадежном фундаменте.

Рекомендация: не игнорируйте системные ошибки, считая их случайными, разовыми сбоями.

**Начните с отладки основного кода, не связанного с многопоточностью**

На первый взгляд совет выглядит тривиально, но еще раз подчеркнуть его значимость не лишне. Убедитесь в том, что сам код работает вне многопоточного контекста. В общем случае это означает создание POJO-объектов, вызываемых из потоков. POJO-объекты не обладают поддержкой многопоточности, а следовательно, могут тестироваться вне многопоточной среды. Чем больше системного кода можно разместить в таких POJO-объектах, тем лучше.

Рекомендация: не пытайтесь одновременно отлавливать ошибки в обычном и многопоточном коде. Убедитесь в том, что ваш код работает за пределами многопоточной среды выполнения.

**Реализуйте переключение конфигураций многопоточного кода**

Напишите вспомогательный код поддержки многопоточности, который может работать в разных конфигурациях.

* Один поток; несколько потоков; количество потоков изменяется по ходу выполнения.
* Многопоточный код взаимодействует с реальным кодом или тестовыми заменителями.
* Код выполняется с тестовыми заменителями, которые работают быстро; медленно; с переменной скоростью.
* Настройте тесты таким образом, чтобы они могли выполняться заданное количество раз.

Рекомендация: реализуйте свой многопоточный код так, чтобы он мог выполняться в различных конфигурациях.

**Применяйте инструментовку кода для повышения вероятности сбоев**

Ошибки в многопоточном коде обычно хорошо скрыты от наших глаз. Простыми тестами они не выявляются. Такие ошибки могут проявляться с периодичностью в несколько часов, дней или недель!

Почему же многопоточные ошибки возникают так редко и непредсказуемо, почему их так трудно воспроизвести? Потому что лишь несколько из тысяч возможных путей выполнения кода плохо написанной секции приводят к фактическому отказу. Таким образом, вероятность выбора сбойного пути ничтожно мала. Это обстоятельство серьезно усложняет выявление ошибок и отладку.

Как повысить вероятность выявления таких редких ошибок? Внесите в свой год соответствующие изменения и заставьте его выполняться по разным путям — включите в него вызовы таких методов, как Object.wait(), Object.sleep(), Object.yield() и Object.priority().

Каждый из этих методов влияет на порядок выполнения программы, повышая шансы на выявление сбоя.

**Ручная инструментовка**

Разработчик вставляет вызовы wait(), sleep(), yield() и priority() в свой код вручную. Такой вариант отлично подходит для тестирования особенно коварных фрагментов кода.

Ручная инструментовка имеет много недостатков:

* Разработчик должен каким-то образом найти подходящие места для вставки вызовов.
* Как узнать, где и какой именно вызов следует вставить?
* Если вставленные вызовы останутся в окончательной версии кода, это приведет к замедлению его работы.
* Вам приходится действовать «наобум»: вы либо находите скрытые дефекты, либо не находите их. Вообще говоря, шансы не в вашу пользу.

**Автоматизированная инструментовка**

Также возможна программная инструментовка кода с применением таких инструментов, как Aspect-Oriented Framework, CGLIB или ASM.

Рекомендация: используйте стратегию случайного выбора пути выполнения для выявления ошибок.

# Запахи и эвристические правила

Чтобы построить этот список, я просмотрел и переработал несколько разных программ. При внесении каждого изменения я спрашивал себя, почему я это делаю, и записывал результат. Так появился довольно длинный список того, что, на мой взгляд, «дурно пахнет» при чтении кода. Предполагается, что вы будете читать список от начала к концу, а также использовать его как краткий справочник.

## Комментарии

**C1: Неуместная информация**

В комментариях неуместно размещать информацию, которую удобнее хранить в других источниках: в системах управления исходным кодом, в системах контроля версий и в других системах протоколирования. Например, история изменений только загромождает исходные файлы длинным историческим и малоинтересным текстом. Метаданные (авторы, дата последней модификации и т.д.) в общем случае также неуместны в комментариях. Комментарии должны быть зарезервированы для технической информации о коде и его архитектуре.

**C2: Устаревший комментарий**

Комментарий, содержимое которого потеряло актуальность, считается устаревшим. Комментарии стареют довольно быстро. Не пишите комментарии, которые с течением времени устареют. Обнаружив устаревший комментарий, обновите его или избавьтесь от него как можно быстрее. Устаревшие комментарии часто «отрываются» от кода, который они когда-то описывали. Так в вашем коде появляются плавучие островки недостоверности и бесполезности.

**C3: Избыточный комментарий**

Избыточным считается комментарий, описывающий то, что и так очевидно. Например:

i++; // Увеличение переменной i

Комментарии должны говорить то, что не может сказать сам код.

**C4: Плохо написанный комментарий**

Если уж вы беретесь за написание комментария, напишите его хорошо. Не жалейте времени и позаботьтесь о том, чтобы это был лучший комментарий, который вы способны создать. Тщательно выбирайте слова. Следите за правильностью орфографии и пунктуации. Не пишите сумбурно. Не объясняйте очевидное. Будьте лаконичны.

**C5: Закомментированный код**

Этот код только попусту занимает место, «загнивая» и утрачивая актуальность с каждым днем. В нем вызываются несуществующие функции. В нем используются переменные, имена которых давно изменились. В нем соблюдаются устаревшие конвенции. Он загрязняет модуль, в котором он содержится, и отвлекает людей, которые пытаются его читать.

Увидев закомментированный код, удалите его! Не беспокойтесь, система управления исходным кодом его не забудет. Если кому-то этот код действительно понадобится, то он сможет вернуться к предыдущей версии.

## Рабочая среда

**E1: Построение состоит из нескольких этапов**

Построение проекта должно быть одной тривиальной операцией. Без выборки многочисленных фрагментов из системы управления исходным кодом. Без длинных серий невразумительных команд или контекстно-зависимых сценариев для построения отдельных элементов. Без поиска дополнительных файлов в формате JAR, XML и других артефактов, необходимых для вашей системы. Сначала вы проверяете систему одной простой командой, а потом вводите другую простую команду для ее построения.

**E2: Тестирование состоит из нескольких этапов**

Все модульные тесты должны выполняться всего одной командой. В лучшем случае все тесты запускаются одной кнопкой в IDE. В худшем случае одна простая команда вводится в командной строке. Запуск всех тестов — настолько важная и фундаментальная операция, что она должна быть быстрой, простой и очевидной.

## Функции

**F1: Слишком много аргументов**

Функции должны иметь небольшое количество аргументов. Лучше всего, когда аргументов вообще нет; далее следуют функции с одним, двумя и тремя аргументами. Функции с четырьмя и более аргументами весьма сомнительны; старайтесь не использовать их в своих программах.

**F2: Выходные аргументы**

Выходные аргументы противоестественны. Читатель кода ожидает, что аргументы используются для передачи входной, а не выходной информации. Если ваша функция должна изменять чье-либо состояние, пусть она изменяет состояние объекта, для которого она вызывалась.

**F3: Флаги в аргументах**

Логические аргументы явно указывают на то, что функция выполняет более одной операции. Они сильно запутывают код. Исключите их из своих программ.

**F4: Мертвые функции**

Если метод ни разу не вызывается в программе, то его следует удалить. Хранить «мертвый код» расточительно. Не бойтесь удалять мертвые функции. Не забудьте, что система управления исходным кодом позволит восстановить их в случае необходимости.

## Разное

**G1: Несколько языков в одном исходном файле**

В идеале исходный файл должен содержать код на одном — и только одном! — языке. На практике без смешения языков обойтись, скорее всего, не удастся. Но по крайней мере следует свести к минимуму как количество, так и объем кода на дополнительных языках в исходных файлах.

**G2: Очевидное поведение не реализовано**

Согласно «принципу наименьшего удивления», любая функция или класс должны реализовать то поведение, которого от них вправе ожидать программист. Если очевидное поведение не реализовано, читатели и пользователи кода перестают полагаться на свою интуицию в отношении имен функций. Они теряют доверие к автору кода и им приходится разбираться во всех подробностях реализации.

**G3: Некорректное граничное поведение**

Код должен работать правильно — вроде бы очевидное утверждение. Беда в том, что мы редко понимаем, насколько сложным бывает правильное поведение. Разработчики часто пишут функции, которые в их представлении работают, а затем доверяются своей интуиции вместо того, чтобы тщательно проверить работоспособность своего кода во всех граничных и особых ситуациях.

Усердие и терпение ничем не заменить. Каждая граничная ситуация, каждый необычный и особый случай способны нарушить работу элегантного и интуитивного алгоритма. Не полагайтесь на свою интуицию. Найдите каждое граничное условие и напишите для него тест.

**G4: Отключенные средства безопасности**

Отключать средства безопасности рискованно. Не отключайте сбойные тесты, обещая себе, что вы заставите их проходить позднее, — это так же неразумно, как считать кредитную карту источником бесплатных денег.

**G5: Дублирование**

Это одно из самых важных правил в книге и к нему следует относиться очень серьезно. Практически каждый автор, пишущий о проектировании программного обеспечения, упоминает это правило. Дэйв Томас (Dave Thomas) и Энди Хант (Andy Hunt) назвали его принципом DRY («Don’t Repeat Yourself», то есть «не повторяйтесь») [PRAG]. Кент Бек сделал его одним из основных принципов экстремального программирования в формулировке «Один, и только один раз». Рон Джеффрис (Ron Jeffries) ставит это правило на второе место, после требования о прохождении всех тестов.

Каждый раз, когда в программе встречается повторяющийся код, он указывает на упущенную возможность для абстракции. Возможно, дубликат мог бы стать функцией или даже отдельным классом. «Сворачивая» дублирование в подобные абстракции, вы расширяете лексикон языка программирования. Другие программисты могут воспользоваться созданными вами абстрактными концепциями. Повышение уровня абстракции ускоряет программирование и снижает вероятность ошибок.

Простейшая форма дублирования — куски одинакового кода. Такие дубликаты заменяются простыми методами.

Менее тривиальная форма дублирования — цепочки switch/case или if/else, снова и снова встречающиеся в разных модулях и всегда проверяющие одинаковые наборы условий. Вместо них надлежит применять полиморфизм.

Еще сложнее модули со сходными алгоритмами, но содержащие похожих строк кода. Однако дублирование присутствует и в этом случае. Проблема решается применением паттернов ШАБЛОННЫЙ МЕТОД или СТРАТЕГИЯ.

В сущности, большинство паттернов проектирования, появившихся за последние 15 лет, представляет собой хорошо известные способы борьбы с дублированием.

**G6: Код на неверном уровне абстракции**

В программировании важную роль играют абстракции, отделяющие высокоуровневые общие концепции от низкоуровневых подробностей. Иногда эта задача решается созданием абстрактных классов, содержащих высокоуровневые концепции, и производных классов, в которых хранятся низко­уровневые концепции. Действуя подобным образом, необходимо позаботиться о том, чтобы разделение было полным. Все низкоуровневые концепции должны быть сосредоточены в производных классах, а все высокоуровневые концепции объединяются в базовом классе.

Например, константы, переменные и вспомогательные функции, относящиеся только к конкретной реализации, исключаются из базового класса. Базовый класс не должен ничего знать о них.

Правило также относится к исходным файлам, компонентам и модулям. Качественное проектирование требует, чтобы концепции разделялись на разных уровнях и размещались в разных контейнерах. Иногда такими контейнерами являются базовые и производные классы; в других случаях это могут быть исходные файлы, модули или компоненты. Но какое бы решение ни было выбрано в конкретном случае, разделение должно быть полным. Высокоуровневые и низкоуровневые концепции не должны смешиваться.

**G7: Базовые классы, зависящие от производных**

Самая распространенная причина для разбиения концепций на базовые и производные классы состоит в том, чтобы концепции базового класса, относящиеся к более высокому уровню, были независимы от низкоуровневых концепций производных классов. Следовательно, когда в базовом классе встречаются упоминания имен производных классов, значит, в проектировании что-то сделано не так. В общем случае базовые классы не должны ничего знать о своих производных классах.

Конечно, у этого правила имеются свои исключения. Иногда количество производных классов жестко фиксировано, а в базовом классе присутствует код для выбора между производными классами. Подобная ситуация часто встречается в реализациях конечных автоматов. Однако в этом случае между базовым и производными классами существует жесткая привязка, и они всегда размещаются вместе в одном файле jar. В общем случае нам хотелось бы иметь возможность размещения производных и базовых классов в разных файлах jar.

Размещение производных и базовых классов в разных файлах jar, при котором базовые файлы jar ничего не знают о содержимом производных файлов jar, позволяет организовать развертывание систем в формате дискретных, независимых компонентов. Если в такие компоненты будут внесены изменения, то они развертываются заново без необходимости повторного развертывания базовых компонентов. Такая архитектура значительно сокращает последствия от вносимых изменений и упрощает сопровождение систем в условиях реальной эксплуатации.

**G8: Слишком много информации**

Хорошо определенные модули обладают компактными интерфейсами, позволяющими сделать много минимальными средствами. Для плохо определенных модулей характерны широкие, глубокие интерфейсы, которые заставляют пользователя выполнять много разных операций для решения простых задач. Хорошо определенный интерфейс предоставляет относительно небольшое количество функций, поэтому степень логической привязки при его использовании относительно невелика. Плохо определенный интерфейс предоставляет множество функций, которые необходимо вызывать, поэтому его использование сопряжено с высокой степенью логической привязки.

Хорошие разработчики умеют ограничивать интерфейсы своих классов и модулей. Чем меньше методов содержит класс, тем лучше. Чем меньше переменных известно функции, тем лучше. Чем меньше переменных экземпляров содержит класс, тем лучше.

Скрывайте свои данные. Скрывайте вспомогательные функции. Скрывайте константы и временные переменные. Не создавайте классы с большим количеством методов или переменных экземпляров. Не создавайте большого количества защищенных переменных и функций в субклассах. Сосредоточьтесь на создании очень компактных, концентрированных интерфейсов. Сокращайте логические привязки за счет ограничения информации.

**G9: Мертвый код**

Мертвым кодом называется код, не выполняемый в ходе работы программы. Он содержится в теле команды if, проверяющей невозможное условие. Он содержится в секции catch для блока try, никогда не инициирующего исключения. Он содержится в маленьких вспомогательных методах, которые никогда не вызываются, или в никогда не встречающихся условиях switch/case.

Мертвый код плох тем, что спустя некоторое время он начинает «плохо пахнуть». Чем древнее код, тем сильнее и резче запах. Дело в том, что мертвый код не обновляется при изменении архитектуры. Он компилируется, но не соответствует более новым конвенциям и правилам. Он был написан в то время, когда система была другой. Обнаружив мертвый код, удалите его из системы.

**G10: Вертикальное разделение**

Переменные и функции должны определяться вблизи от места их использования. Локальные переменные должны объявляться непосредственно перед первым использованием и должны обладать небольшой вертикальной областью видимости. Объявление локальной переменной не должно отдаляться от места ее использования на сотню строк.

Приватные функции должны определяться сразу же после первого использования. Приватные функции принадлежат области видимости всего класса, но вертикальное расстояние между вызовами и определениями все равно должно быть минимальным. Приватная функция должна обнаруживаться простым просмотром кода от места первого использования.

**G11: Непоследовательность**

Если некая операция выполняется определенным образом, то и все похожие операции должны выполняться так же. Это правило возвращает нас к «принципу наименьшего удивления». Ответственно подходите к выбору новых схем и обозначений, а если уж выбрали — продолжайте следовать им.

Если в функцию включена переменная response для хранения данных HttpServletResponse, будьте последовательны и используйте такое же имя переменной в других функциях, работающих с объектами HttpServletResponse. Если метод называется processVerificationRequest, присваивайте похожие имена (например, processDeletionRequest) методам, обрабатывающим другие запросы.

**G12: Балласт**

Неиспользуемые переменные, невызываемые функции, бессодержательные комментарии — все это бесполезный балласт, который следует удалить. Поддерживайте чистоту в своих исходных файлах, следите за их структурой и не допускайте появления балласта.

**G13: Искусственные привязки**

То, что не зависит друг от друга, не должно объединяться искусственными привязками. Например, обобщенные перечисления не должны содержаться в более конкретных классах, потому что в этом случае информация о конкретном классе должна быть доступна в любой точке приложения, в которой используется перечисление. То же относится и к статическим функциям общего назначения, объявляемым в конкретных классах.

В общем случае искусственной считается привязка между двумя модулями, не имеющая явной, непосредственной цели. Искусственная привязка возникает в результате размещения переменной, константы или функции во временно удобном, но неподходящем месте. Главные причины для появления искусственных привязок — лень и небрежность.

**G14: Функциональная зависть**

Для методов класса должны быть важны переменные и функции того класса, которому они принадлежат, а не переменные и функции других классов. Когда метод использует методы доступа другого объекта для манипуляций с его данными, то он завидует области видимости класса этого объекта. Он словно мечтает находиться в другом классе, чтобы иметь прямой доступ к переменным, с которыми он работает.

В общем случае от функциональной зависти следует избавиться, потому что она предоставляет доступ к «внутренностям» класса другому классу. Впрочем, иногда функциональная зависть оказывается неизбежным злом.

**G15: Аргументы-селекторы**

Ничто так не раздражает, как висящий в конце вызова функции аргумент false. Зачем он здесь? Что изменится, если этот аргумент будет равен true? Смысл селектора трудно запомнить, но дело не только в этом — селектор указывает на объединение нескольких функций в одну. Аргументы-селекторы помогают ленивому программисту избежать разбиения большой функции на несколько меньших.

Конечно, селекторы не обязаны быть логическими величинами. Это могут быть элементы перечислений, целые числа или любые другие типы аргументов, в зависимости от которых выбирается поведение функции. В общем случае лучше иметь несколько функций, чем передавать функции признак для выбора поведения.

**G16: Непонятные намерения**

Код должен быть как можно более выразительным. Слишком длинные выражения, венгерская запись, «волшебные числа» — все это скрывает намерения автора.

**G17: Неверное размещение**

Одно из самых важных решений, принимаемых разработчиком, — выбор места для размещения кода. Например, где следует объявить константу PI? В классе Math? А может, ей место в классе Trigonometry? Или в классе Circle?

В игру вступает принцип наименьшего удивления. Код следует размещать там, где читатель ожидает его увидеть. Константа PI должна находиться там, где объявляются тригонометрические функции. Константа OVERTIME\_RATE объявляется в классе HourlyPayCalculator.

Иногда мы пытаемся «творчески» подойти к размещению функциональности. Мы размещаем ее в месте, удобном для нас, но это не всегда выглядит естественно для читателя кода. Предположим, потребовалось напечатать отчет с общим количеством отработанных часов. Мы можем просуммировать часы в коде, печатающем отчет, или же накапливать сумму в коде обработки учетных карточек рабочего времени.

Чтобы принять решение, можно посмотреть на имена функций. Допустим, в модуле отчетов присутствует функция с именем getTotalHours, а в модуле обработки учетных карточек присутствует функция saveTimeCard. Какая из этих двух функций, если судить по имени, наводит на мысль о вычислении суммы? Ответ очевиден.

Очевидно, по соображениям производительности сумму правильнее вычислять при обработке карточек, а не при печати отчета. Все верно, но этот факт должен быть отражен в именах функций. Например, в модуле обработки учетных карточек должна присутствовать функция computeRunningTotalOfHours.

**G18: Неуместные статические методы**

Иногда мы пишем статические функции, которые статическими быть не должны.

В общем случае отдавайте предпочтение нестатическим методам перед статическими. Если сомневаетесь, сделайте функцию нестатической. Если вы твердо уверены, что функция должна быть статической, удостоверьтесь в том, что от нее не потребуется полиморфное поведение.

**G19: Используйте пояснительные переменные**

Один из самых эффективных способов улучшения удобочитаемости программы заключается в том, чтобы разбить обработку данных на промежуточные значения, хранящиеся в переменных с содержательными именами.

Простое использование пояснительных переменных четко объясняет, что первое совпадение содержит ключ (key), а второе — значение (value).

**G20: Имена функций должны описывать выполняемую операцию**

Если вам приходится обращаться к реализации (или документации), чтобы ­понять, что делает та или иная функция, постарайтесь найти более удачное имя или разбейте функциональность на меньшие функции с более понятными именами.

**G21: Понимание алгоритма**

Очень много странного кода пишется из-за того, что люди не утруждают себя пониманием алгоритмов. Они заставляют программу работать «грубой силой», набивая ее командами if и флагами, вместо того чтобы остановиться и подумать, что же в действительности происходит.

Прежде чем откладывать в сторону готовую функцию, убедитесь в том, что вы понимаете, как она работает. Прохождения всех тестов недостаточно. Вы должны знать, что ваше решение правильно.

Один из лучших способов достичь этого знания и понимания — разбить функцию на фрагменты настолько чистые и выразительные, что вам станет совершенно очевидно, как работает данная функция.

**G22: Преобразование логических зависимостей в физические**

Если один модуль зависит от другого, зависимость должна быть не только логической, но и физической. Зависимый модуль не должен делать никаких предположений (иначе говоря, создавать логические зависимости) относительно того модуля, от которого он зависит. Вместо этого он должен явно запросить у этого модуля всю необходимую информацию.

**G23: Используйте полиморфизм вместо if/Else или switch/Case**

Я использую правило «ОДНОЙ КОМАНДЫ SWITCH»: для каждого типа выбора программа не должна содержать более одной команды switch. Множественные конструкции switch следует заменять полиморфными объектами.

**G24: Соблюдайте стандартные конвенции**

Все рабочие группы должны соблюдать единые стандарты кодирования, основанные на отраслевых нормах. Стандарт кодирования определяет, где объявляются переменные экземпляров; как присваиваются имена классов, методов и переменных; где размещаются фигурные скобки и т.д. Документ с явным описанием этих правил не нужен — сам код служит примером оформления.

Правила должны соблюдаться всеми участниками группы. Это означает, что каждый участник группы должен быть достаточно разумным, чтобы понимать: неважно, как именно размещаются фигурные скобки, если только все согласились размещать их одинаковым образом.

**G25: Заменяйте «волшебные числа» именованными константами**

В общем случае присутствие «сырых» чисел в коде нежелательно. Числа следует скрыть в константах с содержательными именами.

Например, число 86,400 следует скрыть в константе SECONDS\_PER\_DAY. Если в странице отчета выводится 55 строк, число 55 следует скрыть в константе LINES\_PER\_PAGE.

**G26: Будьте точны**

Наивно ожидать, что первая запись, возвращаемая по запросу, является единственной. Использовать числа c плавающей точкой для представления де­нежных сумм — почти преступление. Отсутствие блокировок и/или управления транз­акциями только потому, что вы думаете, что одновременное обновление маловероятно — в лучшем случае халатность. Объявление переменной с типом ArrayList там, где более уместен тип List — чрезмерное ограничение. Объявление всех переменных защищенными по умолчанию — недостаточное ограничение.

Принимая решение в своем коде, убедитесь в том, что вы действуете предельно точно и аккуратно. Знайте, почему принимается решение, и как вы собираетесь поступать с исключениями из правила. Не ленитесь обеспечивать точность своих решений. Если вы решили вызвать функцию, которая может вернуть null — проверьте возвращаемое значение. Если вы запрашиваете из базы данных запись, которая, по вашему мнению, является единственной — проверьте, не вернул ли запрос дополнительные записи. Если вам нужно работать с денежными суммами, используйте целые числа и округляйте результат по действующим правилам. Если в программе существует возможность одновременного объявления, реализуйте ту или иную разновидность блокировки. Неоднозначности и неточности в коде объясняются либо недопониманием, либо ленью. В любом случае от них следует избавиться.

**G27: Структура важнее конвенций**

Воплощайте архитектурные решения на уровне структуры кода; она важнее стандартов и конвенций. Содержательные имена полезны, но структура, заставляющая пользователя соблюдать установленные правила, важнее. Например, конструкции switch/case с хорошо выбранными именами элементов перечисления уступают базовым классам с абстрактными методами. Ничто не вынуждает пользователя применять одинаковую реализацию switch/case во всех случаях; с другой стороны, базовые классы заставляют его реализовать все абстрактные методы в конкретных классах.

**G28: Инкапсулируйте условные конструкции**

В булевской логике достаточно трудно разобраться и вне контекста команд if или while. Выделите в программе функции, объясняющие намерения условной конструкции. Например, команда

if (shouldBeDeleted(timer))

выразительнее команды

if (timer.hasExpired() && !timer.isRecurrent())

**G29: Избегайте отрицательных условий**

Отрицательные условия немного сложнее для понимания, чем положительные. Таким образом, по возможности старайтесь формулировать положительные условия. Например, запись

if (buffer.shouldCompact())

предпочтительнее записи

if (!buffer.shouldNotCompact())

**G30: Функции должны выполнять одну операцию**

Часто возникает искушение разделить свою функцию на несколько секций для выполнения разных операций. Такие функции выполняют несколько операций; их следует преобразовать в группу меньших функций, каждая из которых выполняет только одну операцию.

**G31: Скрытые временные привязки**

Временные привязки часто необходимы, но они не должны скрываться. Структура аргументов функций должна быть такой, чтобы последовательность вызова была абсолютно очевидной.

Например, временную привязку можно реализовать посредством создания «эстафеты». Каждая функция выдает результат, необходимый для работы следующей функции, и вызвать эти функции с нарушением порядка сколько-нибудь разумным способом уже не удастся.

**G32: Структура кода должна быть обоснована**

Структура кода должна выбираться не произвольно, а по строго определенным причинам. Позаботьтесь о том, чтобы эти причины были выражены в структуре кода. Если при чтении кода создается впечатление, что его структура выбрана произвольно, другим пользователям может показаться, что ее можно изменить. Если во всей системе последовательно используется единая структура кода, другие пользователи примут ее и сохранят действующие правила.

**G33: Инкапсулируйте граничные условия**

Отслеживать граничные условия нелегко. Разместите их обработку в одном месте. Не позволяйте им «растекаться» по всему коду. Не допускайте, чтобы в вашей программе кишели многочисленные +1 и –1.

**G34: Функции должны быть написаны на одном уровне абстракции**

Все команды функции должны быть сформулированы на одном уровне абстракции, который расположен одним уровнем ниже операции, описываемой именем функции. Возможно, это эвристическое правило сложнее всего правильно интерпретировать и соблюдать.

**G35: Храните конфигурационные данные на высоких уровнях**

Если в программе имеется константа, определяющая значение по умолчанию или параметр конфигурации, и эта константа известна на высоких уровнях абстракции, — не прячьте ее в низкоуровневой функции. Передайте ее в аргументе низкоуровневой функции, вызываемой из функции высокого уровня.

Конфигурационные константы находятся на очень высоком уровне. Если потребуется, их можно легко изменить. Их значения передаются на более низкие уровни иерархии другим компонентам приложения. Значения этих констант не принадлежат нижним уровням приложения.

**G36: Избегайте транзитивных обращений**

В общем случае модуль не должен обладать слишком полной информацией о тех компонентах, с которыми он взаимодействует. Точнее, если A взаимодействует с B, а B взаимодействует с C, то модули, использующие A, не должны знать о C (то есть нежелательны конструкции вида a.getB().getC().doSomething();). Иногда это называется «законом Деметры». Прагматичные программисты используют термин «умеренный код».

В любом случае все сводится к тому, что модули должны обладать информацией только о тех модулях, с которыми они непосредственно взаимодействуют, а не располагать навигационной картой всей системы.

Если в нескольких модулях используется та или иная форма команды a.getB().getC(), то в дальнейшем вам будет трудно изменить архитектуру системы, вставив между B и C промежуточный компонент Q. Придется найти каждое вхождение a.getB().getC() и преобразовать его в a.getB().getQ().getC(). Так образуются жесткие, закостеневшие архитектуры. Слишком многие модули располагают слишком подробной информацией о системе.

Весь необходимый сервис должен предоставляться компонентами, с которыми напрямую взаимодействует модуль. Не заставляйте пользователя странствовать по графу объектов системы в поисках нужного метода. Проблема должна решаться простыми вызовами вида

myCollaborator.doSomething()

## Java

**J1: Используйте обобщенные директивы импорта**

Если вы используете два и более класса из пакета, импортируйте весь пакет командой

import package.\*;

Длинные списки импорта пугают читателя кода. Начало модуля не должно загромождаться 80-строчным списком директив импорта. Список импорта должен быть точной и лаконичной конструкцией, показывающей, с какими пакетами мы собираемся работать.

Конкретные директивы импорта определяют жесткие зависимости, обобщенные директивы импорта — нет. Если вы импортируете конкретный класс, то этот класс обязательно должен существовать. Но пакет, импортируемый обобщенной директивой, может не содержать ни одного класса. Директива импорта просто добавляет пакет в путь поиска имен. Таким образом, обобщенные директивы импорта не создают реальных зависимостей, а следовательно, способствуют смягчению логических привязок между модулями.

Обобщенные директивы импорта иногда становятся причиной конфликтов имен и неоднозначностей. Два класса с одинаковыми именами, находящиеся в разных пакетах, должны импортироваться конкретными директивами (или по крайней мере их имена должны уточняться при использовании). Это создает определенные неудобства, однако ситуация встречается достаточно редко, так что в общем случае обобщенные директивы импорта все равно лучше конкретных.

**J2: Не наследуйте от констант**

Я уже неоднократно встречался с этим явлением, и каждый раз оно заставляло меня недовольно поморщиться. Программист размещает константы в интерфейсе, а затем наследует от этого интерфейса для получения доступа к константам.

Совершенно отвратительная привычка! Константы скрыты на верхнем уровне ­иерархии наследования. Брр! Наследование не должно применяться для того, чтобы обойти языковые правила видимости. Используйте статическое импортирование.

**J3: Константы против перечислений**

В языке появились перечисления ( Java 5) — пользуйтесь ими! Не используйте старый трюк с public static final int. Смысл int может потеряться; смысл перечислений потеряться не может, потому что они принадлежат указанному перечислению.

Тщательно изучите синтаксис перечислений. Не забудьте, что перечисления могут содержать методы и поля. Это очень мощные синтаксические инструменты, значительно превосходящие int по гибкости и выразительности.

## Имена

**N1: Используйте содержательные имена**

Не торопитесь с выбором имен. Позаботьтесь о том, чтобы имена были содержательными. Помните, что смысл может изменяться в ходе развития программного продукта; почаще переосмысливайте уместность выбранных вами имен.

Имена в программных продуктах на 90% определяют удобочитаемость кода. Не жалейте времени на то, чтобы выбрать их осмысленно, и поддерживайте их актуальность. Имена слишком важны, чтобы относиться к ним легкомысленно.

Сила хорошо выбранных имен заключается в том, что они дополняют структуру кода описаниями. На основании этих описаний у читателя формируются определенные предположения по поводу того, что делают другие функции модуля.

**N2: Выбирайте имена на подходящем уровне абстракции**

Не используйте имена, передающие информацию о реализации. Имена должны отражать уровень абстракции, на котором работает класс или функция.

**N3: По возможности используйте стандартную номенклатуру**

Имена проще понять, если они основаны на существующих конвенциях или стандартных обозначениях. Например, при использовании паттерна ДЕКОРАТОР можно включить в имена декорирующих классов слово Decorator. Например, имя AutoHangupModemDecorator может быть присвоено классу, который дополняет класс Modem возможностью автоматического разрыва связи в конце сеанса.

Паттерны составляют лишь одну разновидность стандартов. Например, в языке Java функции, преобразующие объекты в строковые представления, часто называются toString. Лучше следовать подобным стандартным конвенциям, чем изобретать их заново.

**N4: Недвусмысленные имена**

Выбирайте имена, которые максимально недвусмысленно передают назначение функции или переменной.

**N5: Используйте длинные имена для длинных областей видимости**

Длина имени должна соответствовать длине его области видимости. Переменным с крошечной областью видимости можно присваивать очень короткие имена, но у переменных с большей областью видимости имена должны быть длинными.

Если область видимости переменной составляет всего пять строк, то переменной можно присвоить имя i или j (например, в счетчиках).

С другой стороны, смысл коротких имен переменных и функций рассеивается на длинных дистанциях. Таким образом, чем длиннее область видимости имени, тем более длинным и точным должно быть ее имя.

**N6: Избегайте кодирования**

Информация о типе или области видимости не должна кодироваться в именах. Префиксы вида m\_ или f бессмысленны в современных средах. Кроме того, информация о проекте и/или подсистеме (например, префикс vis\_ для подсистемы визуализации) также отвлекает читателя и является избыточной. Современные среды разработки позволяют получить всю необходимую информацию без уродования имен. Поддерживайте чистоту в своих именах, не загрязняйте их венгерской записью.

**N7: Имена должны описывать побочные эффекты**

Имена должны описывать все, что делает функция, переменная или класс. Не скрывайте побочные эффекты за именами. Не используйте простые глаголы для описания функции, которая делает что-то помимо этой простой операции.

## Тесты

**T1: Нехватка тестов**

Сколько тестов должен включать тестовый пакет? Тестовый пакет должен тестировать все, что может сломаться. Если в системе остались условия, не проверенные тестами, или вычисления, правильность которых не подтверждена, значит, количество тестов недостаточно.

**T2: Используйте средства анализа покрытия кода**

Средства анализа покрытия сообщают о пробелах в вашей стратегии тестирования. Они упрощают поиск модулей, классов и функций с недостаточно полным тестированием. Во многих IDE используются визуальные обозначения: строки, покрытые тестами, выделяются зеленым цветом, а непокрытые — красным. Это позволяет легко и быстро обнаружить команды if или catch, тело которых не проверяется тестами.

**T3: Не пропускайте тривиальные тесты**

Тривиальные тесты пишутся легко, а их информативная ценность превышает затраты времени на их создание.

**T4: Отключенный тест как вопрос**

Иногда мы не уверены в подробностях поведения системы, потому что неясны сами требования к программе. Вопрос о требованиях можно выразить в виде ­теста — закомментированного или помеченного аннотацией @Ignore. Выбор за­висит от того, компилируется или нет код, к которому относится неопределенность.

**T5: Тестируйте граничные условия**

Особенно тщательно тестируйте граничные условия. Программисты часто ­правильно реализуют основную часть алгоритма, забывая о граничных ситуациях.

**T6: Тщательно тестируйте код рядом с ошибками**

Ошибки часто собираются группами. Если вы обнаружили ошибку в функции, особенно тщательно протестируйте эту функцию. Может оказаться, что ошибка была не одна.

**T7: Закономерности сбоев часто несут полезную информацию**

Иногда анализ закономерностей в сбоях тестовых сценариев помогает выявить причины возникших проблем. Это еще один аргумент в пользу максимальной полноты тестовых сценариев. Всесторонние наборы тестовых сценариев, упорядоченные логичным образом, выявляют закономерности.

Простой пример: вы заметили, что все тесты с входными данными, длина которых превышает пять символов, завершаются неудачей? Или что любой тест, который передает во втором аргументе функции отрицательное число, не проходит? Иногда простая закономерность чередования красного и зеленого в тестовом отчете заставляет нас воскликнуть «Ага!» на пути к правильному решению.

**T8: Закономерности покрытия кода часто несут полезную информацию**

Анализ того, какой код выполняется или не выполняется в ходе тестирования, иногда подсказывает причины возможных сбоев в ходе тестирования.

**T9: Тесты должны работать быстро**

Медленный тест не выполняется за разумное время. Если время поджимает, из тестового пакета первыми будут удалены медленные тесты. Сделайте все необходимое для того, чтобы ваши тесты работали быстро.