ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ЛЕКЦИЯ № 6

ПРЕПОДАВАТЕЛЬ: ХУСТОЧКА А.В.

КНИГА «ЧИСТЫЙ КОД»

- Данная лекция представляет собой выжимку из книги «Чистый код» Роберта Мартина.
- Бесплатная электронная .pdf-версия: https://sd.blackball.lv/library/chistyj_kod_-_sozdanie_analiz_i_refaktoring_(2013).pdf
- Не стоит все советы из книги принимать за «чистую моменту» и безукоризненно им следовать: всё-таки прошло достаточно много времени с момента написания книги, технологии шагнули вперёд и кое-что могло потерять актуальность.

ПЛОХОЙ КОД

- Плохой код когда-нибудь мешал вашей работе? Любой сколько-нибудь опытный программист неоднократно попадал в подобную ситуацию. Мы продираемся через плохой код. Мы вязнем в хитросплетении ветвей, попадаем в скрытые ловушки. Мы с трудом прокладываем путь, надеясь получить хоть какую-нибудь подсказку, что же происходит в коде; но не видим вокруг себя ничего, кроме новых залежей невразумительного кода.
- Конечно, плохой код мешал вашей работе. Почему же вы писали его? Пытались поскорее решить задачу? Торопились? Возможно. А может быть, вам казалось, что у вас нет времени качественно выполнить свою работу; что ваше начальство будет недовольно, если вы потратите время на чистку своего кода. А может, вы устали работать над программой и вам хотелось поскорее избавиться от нее. А может, вы посмотрели на список запланированных изменений и поняли, что вам необходимо поскорее «прикрутить» этот модуль, чтобы перейти к следующему.
- Такое бывало с каждым. Каждый из нас смотрел на тот хаос, который он только что сотворил, и решал оставить его на завтра. Каждый с облегчением видел, что бестолковая программа работает, и решал, что рабочая мешанина — лучше, чем ничего. Каждый обещал себе вернуться и почистить код… потом. Конечно, в те дни мы еще не знали закон Леблана: потом равносильно никогда.

ИСКУССТВО ЧИСТОГО КОДА

- Допустим, вы согласились с тем, что беспорядок в коде замедляет вашу работу.
- Допустим, вы согласились, что для быстрой работы необходимо соблюдать чистоту. Тогда вы должны спросить себя: «А как мне написать чистый код?» Бесполезно пытаться написать чистый код, если вы не знаете, что это такое.
- Чтобы написать чистый код, необходимо сознательно применять множество приемов, руководствуясь приобретенным усердным трудом чувством «чистоты». Ключевую роль здесь играет «чувство кода». Одни с этим чувством рождаются. Другие работают, чтобы развить его. Это чувство не только позволяет отличить хороший код от плохого, но и демонстрирует стратегию применения наших навыков для преобразования плохого кода в чистый код

ЧТО ТАКОЕ «ЧИСТЫЙ КОД»?

- Вероятно, сколько существует программистов, столько найдется и определений. Поэтому я спросил у некоторых известных, чрезвычайно опытных программистов, что они думают по этому поводу.

- Я люблю, чтобы мой код 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. (c) Бьёрн Страуструп, создатель C++ и автор книги «The C++ Programming Language»

ПРАВИЛО БОЙСКАУТА

- Хорошо написать код недостаточно. Необходимо поддерживать чистоту кода с течением времени. Все мы видели, как код загнивает и деградирует с течением времени. Значит, мы должны активно поработать над тем, чтобы этого не произошло.
- У бойскаутов существует простое правило, которое применимо и к нашей профессии:

  Оставь место стоянки чище, чем оно было до твоего прихода.

- 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
- Избегайте дезинформации
  - Не обозначайте группу учетных записей именем accountList, если только она действительно не хранится в списке (List). Слово «список» имеет для программиста вполне конкретный смысл. Если записи хранятся не в List, а в другом контейнере, это может привести к ложным выводам. В этом примере лучше подойдет имя accountGroup, bunchOfAccounts и даже просто accounts.
- Используйте осмысленные различия
  - 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
- Венгерская запись
  - Венгерская запись играла важную роль во времена Windows C API, когда программы работали с целочисленными дескрипторами (handle), длинными указателями, указателями на void или различными реализациями «строк» (с разным применением и атрибутами). Компиляторы в те дни не поддерживали проверку типов, поэтому программистам были нужны «подсказки» для запоминания типов. В современных языках существует куда более развитая система типов, а компиляторы запоминают типы и обеспечивают их соблюдение. Более того, появилась тенденция к использованию меньших классов и более коротких функций, чтобы программисты видели точку объявления каждой используемой переменной. (Пример венгерской нотации: int aSizes; a - array)
- Имена классов и объектов должны представлять собой существительные и их комбинации
  - Например: 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 представляют собой глаголы или глагольные словосочетания
  - Например: postPayment, deletePage, save и т. д. Методы чтения/записи и предикаты образуются из значения и префикса get, set и is
- Избегайте остроумия
  - Остроумие часто воплощается в форме просторечий или сленга. Например, не используйте имя whack() вместо kill()

ФУНКЦИИ

- Компактность!
  - Первое правило: функции должны быть компактными. Второе правило: функции должны быть еще компактнее.
- Блоки и отступы
- Правило одной операции
  - Функция должна выполнять только одну операцию. Она должна выполнять её хорошо. И ничего другого она делать не должна.
- Секции в функциях
  - Выделение пустыми строками секций инициализации переменных, обработки ошибок итд
- Используйте содержательные имена функций
- Аргументы функций
  - Недопустимо использовать простыню из переменных

- 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!
- 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. Для примера возьмем следующую функцию:

```
public boolean set(String attribute, String value);
```

- Функция присваивает значение атрибуту с указанным именем и возвращает true, если присваивание прошло успешно, или false, если такой атрибут не существует. Это приводит к появлению странных конструкций вида

```
if (set("username", "unclebob"))...
```

- Представьте происходящее с точки зрения читателя кода. Что проверяет это условие? Что атрибут "username" содержит ранее присвоенное значение "unclebob"? Или что проверяет атрибуту "username" успешно присвоено значение "unclebob"? Смысл невозможно вывести из самого вызова, потому что мы не знаем, чем в данном случае является слово set — глаголом или прилагательным.
- Автор предполагал, что set является глаголом, но в контексте команды if это имя скорее воспринимается как прилагательное. Таким образом, команда читается в виде «Если атрибуту username ранее было присвоено значение unclebob», а не «присвоить атрибуту username значение unclebob, и если все прошло успешно, то...»

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.

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.

КОММЕНТАРИИ

- Ничто не помогает так, как уместный комментарий. Ничто не загромождает модуль так, как бессодержательные и безапелляционные комментарии. Ничто не приносит столько вреда, как старый, утративший актуальность комментарий, распространяющий ложь и дезинформацию.
- Комментарии — не список Шиндлера. Не стоит относиться к ним как к «абсолютному добру». На самом деле комментарии в лучшем случае являются неизбежным злом.

- 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!

ХОРОШИЕ КОММЕНТАРИИ

- Юридические комментарии

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
// Публикуется на условиях лицензии GNU General Public License версии 2 и выше.
```

- Информативные комментарии

Уместный пример:

```
// Поиск по формату: kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile( "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

ПЛОХИЕ КОММЕНТАРИИ

- Недостоверные комментарии
- Закомментированный код
- 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. Если уж вы берете на себя хлопоты, связанные с написанием комментария, то по крайней мере читатель должен посмотреть на комментарий и на код и понять, о чем говорится в комментарии. Для примера возьмем следующий комментарий из общих модулей Apache:

```
/*
* Начать с массива, размер которого достаточен для хранения
* всех пикселов (плюс байты фильтра), плюс еще 200 байт
* для данных заголовка
*/
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

Что такое «байты фильтра»? Они как-то связаны с +1? Или с *3? И с тем и с другим? Один пиксел соответствует одному байту? И почему 200? Цель комментария — объяснить код, который не объясняет сам себя. Плохо, когда сам комментарий нуждается в объяснениях.

ФОРМАТИРОВАНИЕ

- Прежде всего твердо заявляю: форматирование кода важно. Оно слишком важно, чтобы не обращать на него внимания, и слишком важно, чтобы относиться к нему с религиозным пылом.
- Форматирование кода направлено на передачу информации, а передача информации является первоочередной задачей профессионального разработчика.

Вертикальное форматирование (длина файла)

- 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.
- 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.
- 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

Горизонтальное форматирование (ширина строки в файла)

- Насколько широкой должна быть строка? Чтобы ответить на этот вопрос, мы проанализируем ширину строк в типичных программах.
- Горизонтальное разделение и сжатие
  - Горизонтальные пропуски используются для группировки взаимосвязанных элементов и разделения разнородных элементов. Рассмотрим следующую функцию:

```
double determinant(double a, double b, double c) { return b*b - 4*a*c; }
```

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.

МОДУЛЬНЫЕ ТЕСТЫ

- За последние десять лет наша профессия прошла долгий путь. В 1997 году никто не слыхал о методологии TDD (Test Driven Development, то есть «разработка через тестирование»). Для подавляющего большинства разработчиков модульные тесты представляли собой короткие фрагменты временного кода, при помощи которого мы убеждались в том, что наши программы «работают». Мы тщательно выписывали свои классы и методы, а потом подмешивали специализированный код для их тестирования. Как правило, при этом использовалась какая-нибудь несложная управляющая программа, которая позволяла вручную взаимодействовать с тестируемым кодом.

ТРИ ЗАКОНА TTD

- В наши дни каждому известно, что по требованиям методологии TDD модульные тесты должны писаться заранее, еще до написания кода продукта. Но это
- правило — всего лишь верхушка айсберга.
- 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. При такой организации работы мы пишем десятки тестов ежедневно, сотни тестов ежемесячно, тысячи тестов ежегодно. При такой организации работы тесты охватывают практически все аспекты кода продукта. Громадный объем тестов, сравнимый с объемом самого кода продукта, может создать немало организационных проблем.

О ЧИСТОТЕ ТЕСТОВ

- 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.
- Тесты как средство обеспечения изменений
  - Если не поддерживать чистоту своих тестов, то вы их лишитесь. А без тестов утрачивается все то, что обеспечивает гибкость кода продукта. Да, вы не ошиблись. 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. Какой бы гибкой ни была ваша архитектура, каким бы качественным ни было логическое деление вашей архитектуры, без тестов вы будете сопротивляться изменениям из опасений, что они приведут к появлению скрытых ошибок.
  - С тестами эти опасения практически полностью исчезают. Чем шире охват тестирования, тем меньше вам приходится опасаться. Вы можете практически свободно вносить изменения даже в имеющий далеко не идеальную архитектуру, запутанный и малопонятный код. Таким образом, вы можете спокойно улучшать архитектуру и строение кода!

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.

Одна проверка на тест

- Существует точка зрения, согласно которой каждая тестовая функция в тесте должна содержать одну — и только одну — директиву assert. Такое правило может показаться излишне жестким, но его преимущества наглядно видны. Тесты приводят к одному выводу, который можно быстро и легко понять при чтении.

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.

РАБОЧАЯ СРЕДА

- 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.
- 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.

РАЗНОЕ

- Несколько языков в одном исходном файле
  - Современные среды программирования позволяют объединять в одном исходном файле код, написанный на разных языках. Например, исходный файл на языке Java может содержать вставки XML, HTML, YAML, JavaDoc, English, JavaScript и т. д. Или, скажем, наряду с кодом HTML в файле JSP может присутствовать код Java, синтаксис библиотеки тегов, комментарии на английском языке, комментарии Javadoc, XML, JavaScript и т. д. В лучшем случае результат получается запутанным, а в худшем — неаккуратным и ненадежным.
  - 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.

- Очевидное поведение не реализовано
  - Любая функция или класс должны реализовать то поведение, которого от них вправе ожидать программист. Допустим, имеется функция, которая преобразует название дня недели в элемент перечисления, представляющий этот день. Day day = DayDate.StringToDay(String dayName); Логично ожидать, что строка "Monday" будет преобразована в Day.MONDAY.
  - Также можно ожидать, что будут поддерживаться стандартные сокращения дней недели, а регистр символов будет игнорироваться. 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.

- 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.

- Отключенные средства безопасности
  - Авария на Чернобыльской станции произошла из-за того, что директор завода отключил все механизмы безопасности, один за другим. Они усложняли проведение эксперимента. Результат — эксперимент так и не состоялся, а мир столкнулся с первой серьезной катастрофой в гражданской атомной энергетике.
  - Отключать средства безопасности рискованно. Ручное управление бывает необходимо, но оно всегда сопряжено с риском. Иногда отключение некоторых (или всех!) предупреждений компилятора позволяет успешно построить программу, но при этом вы рискуете бесконечными отладочными сеансами. Не отключайте сбойные тесты, обещая себе, что вы заставите их проходить позднее, — это так же неразумно, как считать кредитную карту источником бесплатных денег.

- Дублирование
  - 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. Программа выглядит так, словно у программиста дрожат руки, и он снова и снова вставляет один и тот же фрагмент. 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 [GOF].

- Слишком много информации
  - 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.

- 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, сделайте то, что положено делать в таких случаях: достойно похороните его. Удалите его из системы.

- Непонятные намерения
  - Код должен быть как можно более выразительным. Слишком длинные выражения, венгерская запись, «волшебные числа» — все это скрывает намерения автора. Например, функция overTimePay могла бы выглядеть и так:

```
public int m_otCalc() {
        return iThsWkd * iThsRte +
                (int) Math.round(0.5 * iThsRte *
                        Math.max(0, iThsWkd - 400) );
}
```

  - Такая запись выглядит компактной и плотной, но разбираться в ней — сущее мучение. Не жалейте времени на то, чтобы сделать намерения своего кода максимально прозрачными для читателей.

- Имена функций должны описывать выполняемую операцию
  - Взгляните на следующий код:

```
Date newDate = date.add(5);
```

  - Как вы думаете, что он делает — прибавляет пять дней к date? А может, пять недель или часов? Изменяется ли экземпляр date, или функция возвращает новое значение Date без изменения старого? По вызову невозможно понять, что делает эта функция. Если функция прибавляет пять дней с изменением date, то она должна называться addDaysTo или increaseByDays. С другой стороны, если функция возвращает новую дату, смещенную на пять дней, но не изменяет исходного экземпляра date, то она должна называться daysLater или daysSince.
  - 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.