Антипаттерны проектирования. Чистый код.

Анти паттерны

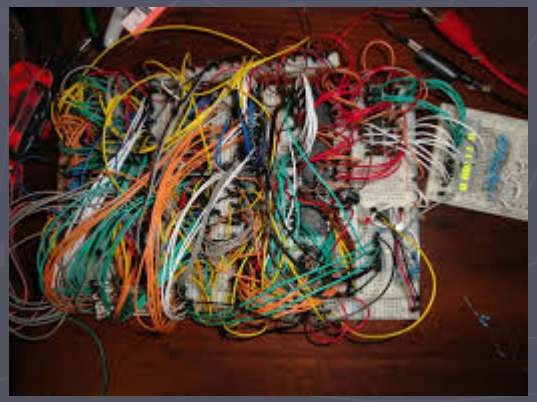
- Spaggeticode

Спагетти-код — слабо структурированная и плохо спроектированная система, запутанная и очень сложная для понимания.

извилистый и очень запутанныйзапутанный код

```
C     A weird program for calculating Pi written in Fortran.
C     From: Fink, D.G., Computers and the Human Mind, Anchor Books, 1966.

      PROGRAM PI
      DIMENSION TERM(100)
      N=1
    3 TERM(N)=((-1)**(N+1))*(4./(2.*N-1.))
      N=N+1
      IF (N-101) 3,6,6
    6 N=1
    7 SUM98 = SUM98+TERM(N)
      WRITE(*,28) N, TERM(N)
      N=N+1
      IF (N-99) 7, 11, 11
   11 SUM99=SUM98+TERM(N)
      SUM100=SUM99+TERM(N+1)
      IF (SUM98-3.141592) 14,23,23
   14 IF (SUM99-3.141592) 23,23,15
   15 IF (SUM100-3.141592) 16,23,23
   16 AV89=(SUM98+SUM99)/2.
      AV90=(SUM99+SUM100)/2.
      COMANS=(AV89+AV90)/2.
      IF (COMANS-3.1415920) 21,19,19
   19 IF (COMANS-3.1415930) 20,21,21
   20 WRITE(*,26)
      GO TO 22
   21 WRITE(*,27) COMANS
   22 STOP
   23 WRITE(*,25)
      GO TO 22
   25 FORMAT('ERROR IN MAGNITUDE OF SUM')
   26 FORMAT('PROBLEM SOLVED')
   27 FORMAT('PROBLEM UNSOLVED', F14.6)
   28 FORMAT(I3, F14.6)
      END
```



Когда проявляется?

Когда алгоритм реализуется в рамках одного метода (функции) и содержит очень большой длинный и запутанный код.

Какие проблемы несёт

- Подобный код в будущем не может разобрать даже автор.
- Очень часто содержит в себе множество других анти-паттернов программирования, включая Copy and Paste Programming.
- Малоэффективный Code Review.
- Использовать спагетти-код повторно невозможно.

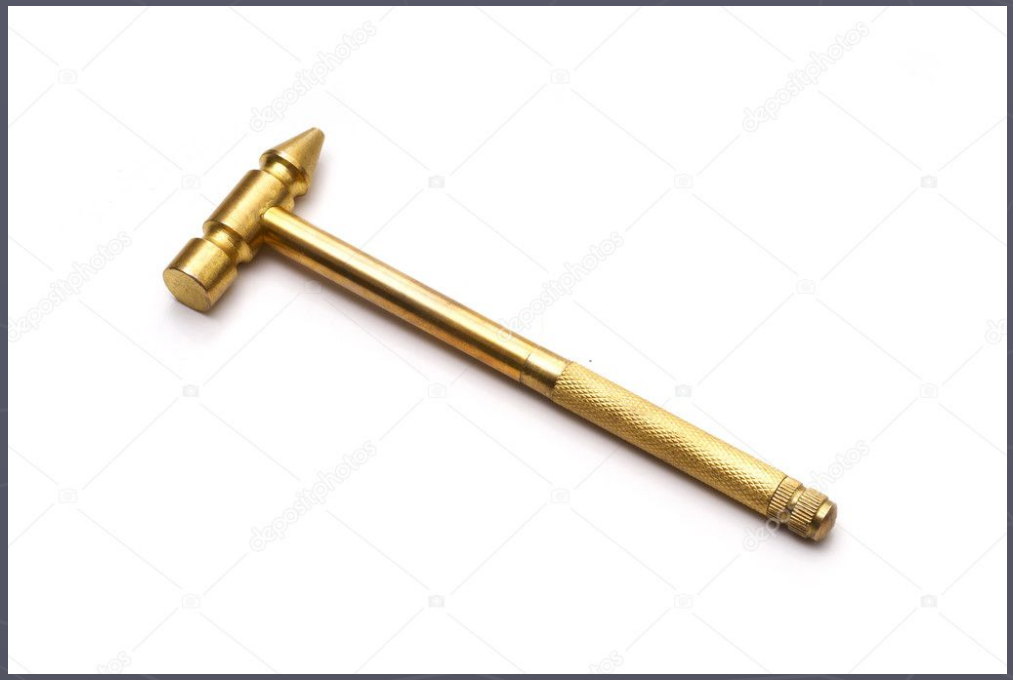
Причины возникновения:

В основном такой код проявляется из-за недостатка опыта в разработке и непонимания принципов программирования. "работает - не трогай" во время рефакторинга так же приводит к спагетти-коду.

Решение

Рефакторинг или полное выкашивание таких участков кода и переписывание их заново.

- Golden hammer (золотой молоток) – одно решение для всех задач.
- уверенность в полной универсальности кода



Какие проблемы несёт

Многие программисты используют данный анти-паттерн не подозревая о собственной некомпетентности, что приводит к остановке саморазвития.

Причинами возникновения

- У новичков — лень к изучению нового и увеличению знаний в шаблонах проектирования; как следствие, новичок пытается решить все задачи единственным известным способом, который он освоил.
- У профессионалов — профессиональная деформация, что приводит к выработке предпочтений в шаблонах проектирования, а не использования того шаблона, который нужен для решения конкретной задачи.

Как решить проблему

При решении задачи продумывать не одно решение, а несколько, определить достоинства и недостатки, и делать взвешенный выбор в пользу самого удачного решения — именно к поиску таких решений и сводится эффективная разработка.

►Code duplication (copy paste)

```
public void SetAudiType(Car car)
{
    car.Type = CarType.Audi;
}
public void SetBMwType(Car car)
{
    car.Type = CarType.Audi;
}
```

разработчик не создал обобщенный метод SetType который бы устанавливаем нужный тип, вместо этого он скопировал код и создал идентичный метод.

Когда он проявляется

Когда разработчику требуется реализовать две схожих задачи, самым простым решением чаще всего он видит следующее: написать одну задачу, скопировать и внести необходимые изменения, чтобы решить вторую задачу.

Какие проблемы несёт

- Ухудшается повторное использование кода — если потребуется подобная функциональность в новом проекте, то нужно будет вычленять код и переносить его.
- Понижается качество кода — часто найденные недочёты в коде правятся только в одном методе, в остальных недостатки остаются.
- Усложняется поддержка кода — в случае, если в изначальном коде была ошибка, которую в будущем нужно исправить, то эта означает, что ошибка попала во все проекты, куда копировался код. Это приводит необходимости выполнять множественные исправления в разных проектах.
- Code Review значительно усложняется, так как приходится делать обзор фактически одного и того же кода в разных проектах без видимой значительной выгоды и роста производительности труда.

Причины возникновения

- Безразличное отношение к будущим участникам разработки проекта – «другие поправят».
- Недостаток опыта в разработке.
- Ограничение времени на разработку.

Как решить проблему

- Создание приватного репозитория решений и использования их в качестве библиотек, модулей, зависимостей. Об этом следует думать до старта разработки.
- Чаще задавайте себе вопрос: возможно, что понадобится решить подобную задачу где-нибудь ещё?

Magic numbers

if (database (ID) == 5)

константы, используемые в коде, но которые не несут никакого смысла без соответствующего комментария.

```
public void setPassword(String password) {
     // don't do this
     if (password.length() > 7) {
          throw new InvalidArgumentException("password");
     }
}
public static final int MAX_PASSWORD_SIZE = 7;

public void setPassword(String password) {
     if (password.length() > MAX_PASSWORD_SIZE) {
          throw new InvalidArgumentException("password");
     }
}
```

Какие проблемы несёт

- Разработчик, который не является автором кода, с трудностями сможет объяснить как это работает.
- Со временем, автор кода тоже не сможет объяснить что-либо.
- Числа затрудняют понимание кода и его рефакторинг.

Причины возникновения

- Спешка при разработке.
- Отсутствие практики групповой разработки или сопровождения проекта.

Как решить проблему

Проводить Code Review, силами разработчиков, которые не задействованы в проекте.

- Hard Code (жесткий код) - фиксация в коде различных данных об окружении.

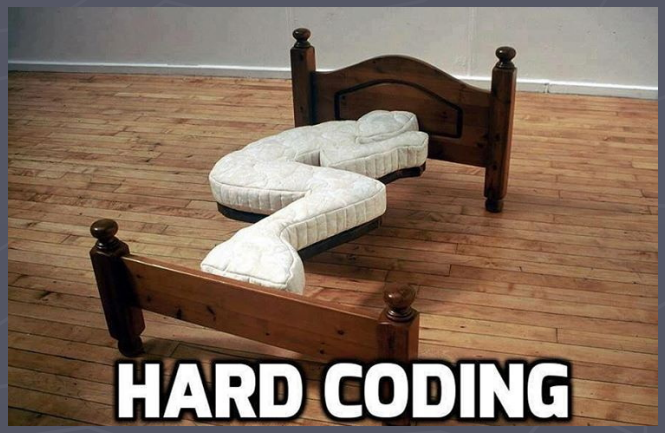
Конфиг, база, константа

```
float areaOfCircle(int radius)
{
    float area = 0;
    area = 3.14*radius*radius;
    return area;
}
```

Например: пути к файлам, имена процессов, устройств и так далее.

```
int main()
{
    const char *filename = "C:\\myfile.txt";

    printf("Filename is: %s\n", filename);
}
```



Какие проблемы несёт

- Код будет корректно работать только в том окружении, под который сделан хардкод.
- Может проявляется непредсказуемые дефекты во время переноса, переименования файлов, и их поведение может меняться при изменении конфигурации устройств.
- Невозможность гибкой настройки под нужный нам environment.
- Усложняет Unit и Integration testing.

Причины возникновения

- Разработчик во время написания или отладки алгоритма пишет хард-код и, по завершению, забывает удалить или модифицировать его.
- Малый опыт разработки под несколько платформ.

Как решить проблему

Оговаривать запрет на хард-код перед началом разработки проекта и проводить тщательные Code Review.

Выносить данные в отдельные конфигурационные файлы.

- Soft code

много абстракции

Много настроек (сложно и непрозрачно)

параноидальная боязнь хард-кода. Этот анти-паттерн является вторым концом палки о хард-коде и поэтому тоже является опасным.

Когда он проявляется

В проекте настраивается абсолютно всё, что делает конфигурацию невероятно сложной и непрозрачной.

Какие проблемы несёт

- При разработке много ресурсов уходит на реализацию возможности настроек абсолютно всего.
- Развёртывание такой системы влечет также дополнительные затраты

Причины возникновения

- Низкая квалификация разработчика — страх допустить анти-паттерн Hard Code.
- Небольшой опыт работы с разными окружениями.

Как решить проблему

Перед началом разработки проект следует определить, что должно быть настраиваемым, а что является постоянным независимо от окружения или может быть настроено автоматически.

Также использование принципов KISS, YAGNI поможет решить проблему.

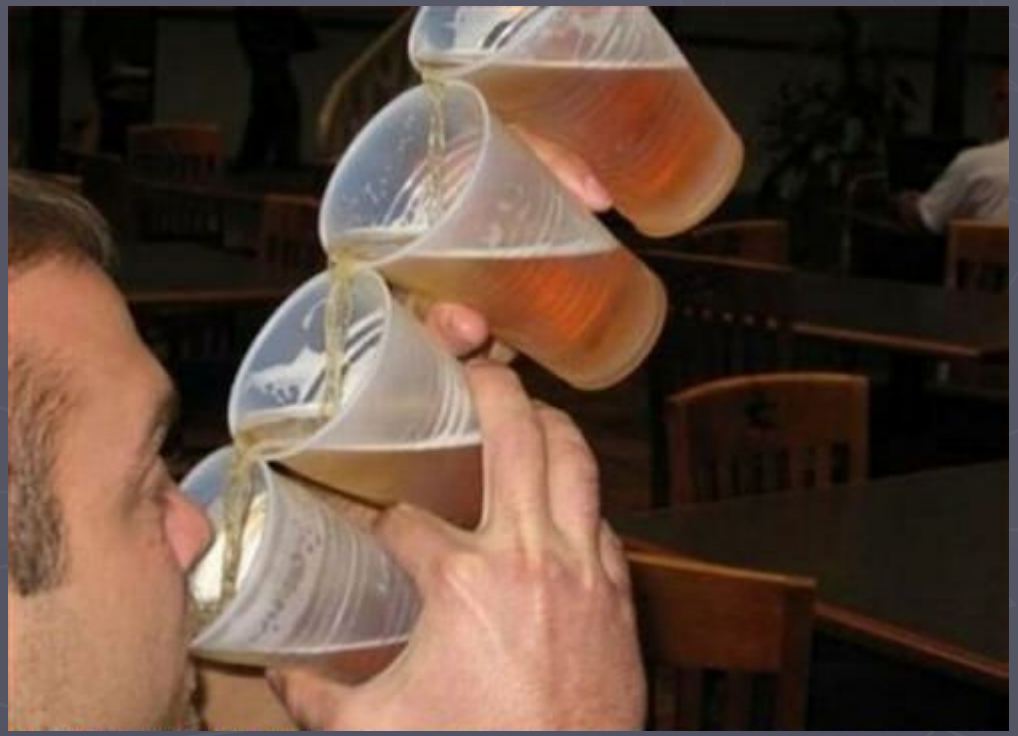
- Academicals complexity

(заумность решения)

Ненужная сложность может быть внесена в решение любой задачи.

Усложнение понимания кода

Снижение скорости работы



Когда проявляется?

- В коде есть избыточные проверки
- часть кода, реализовано с использованием анти-паттерна Soft Code, что позволяет конфигурировать поведение нашего кода.
- Также проявляется, когда технический долг не рефакторится.

Какие проблемы несёт

- Усложнение понимания кода.
- Снижение скорости работы

Причины возникновения

- Отсутствие или низкое качество рефакторинга.
- Некомпетентность программиста.
- Желание показать всем, какой ты молодец, выучил новые фичи.

Как решить проблему

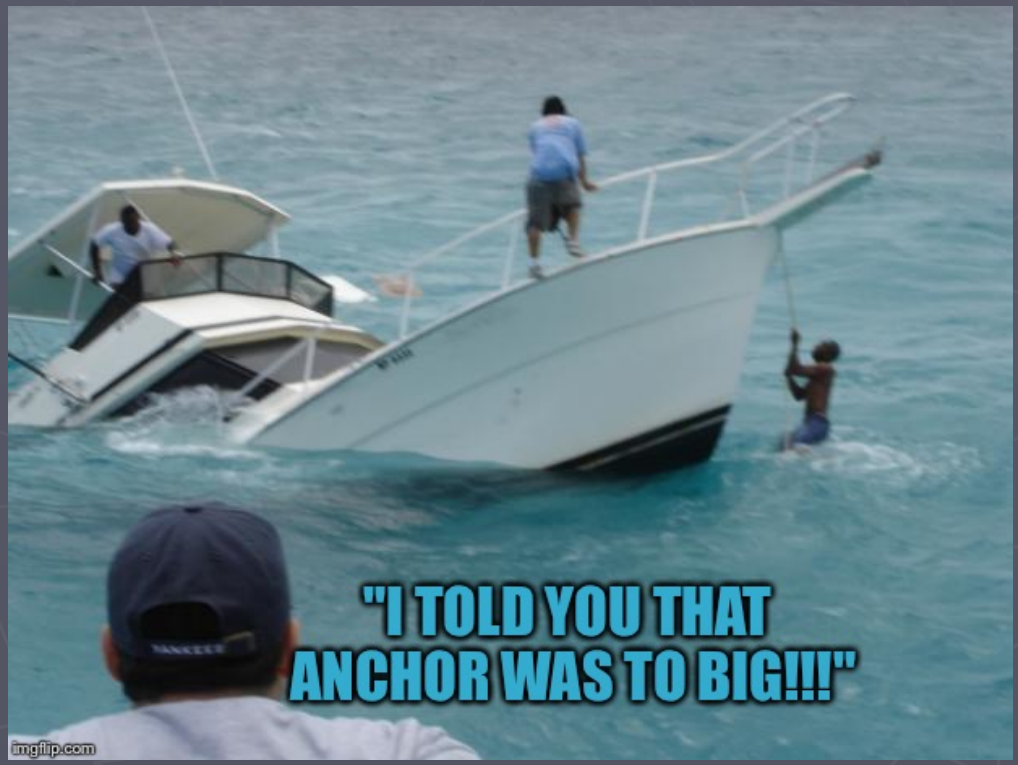
Следует проводить Code Review и выполнять рефакторинг.

Также использование принципов KISS поможет решить проблему.

Boad anchor

(лодочный якорь)

сохранение неиспользуемых частей системы, которые остались после оптимизации или рефакторинга.



Когда он проявляется

- После рефакторинга, некоторые части кода остаются в системе, хотя они уже больше не будут использоваться.
- При сохранении части кода «на будущее», на случай, если придётся ещё раз использовать.

Какие проблемы несёт

Значительно усложняет чтение проекта, не неся абсолютно никакой практической ценности.

Причины возникновения

Неумение использовать такие инструменты как “Система управления версиями” (git, mercurial).

Как решить проблему

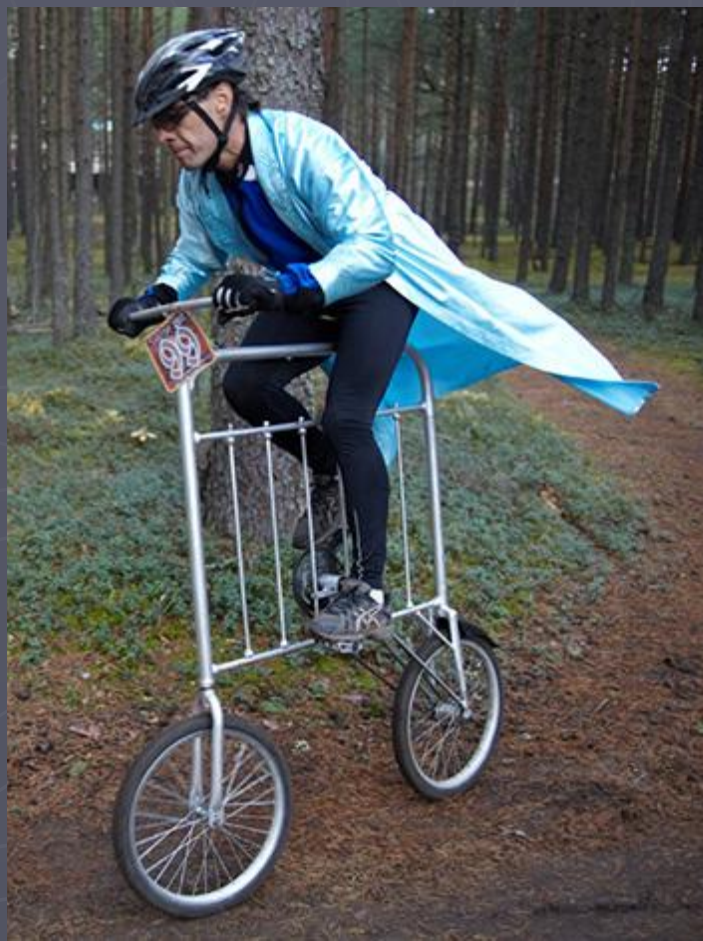
планирование при разработке, написание продуманного кода.

При рефакторинге и оптимизации кода принудительно удалять код, который более использоваться не будет или создавать отдельную ветку в системе управления версиями, на случай, если есть вероятность возврата к архивному решению.

- Reinvention the wheal

(изобретение колеса/велосипеда)

разработчик реализует собственное решение для задачи, для которой уже существуют решения, которое может быть лучшие, чем придуманное



Когда проявляется?

Когда разработчик считает свои знания уникальными, поэтому для каждой задачи пытается придумать собственное решение, не смотря на опыт его предшественников.

Какие проблемы несёт

- Потеря времени и понижение эффективности работы программиста.
- Снижает эффективность или оптимальность конечного продукта

►Причины возникновения:

- Повышенная самооценка или пониженная самокритичность.
- Нехватка времени на изучение готовых решений в интернете.

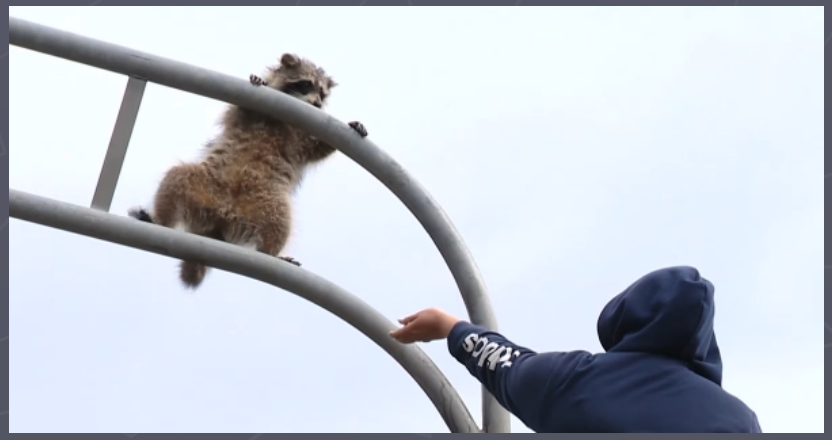
Как решить проблему

Разработчик должен ориентироваться в задачах, которые могут предстать перед ним, чтобы грамотно их решать — использовать готовые решение или изобретать собственные.

Полностью же отбрасывать возможность самостоятельного решения нельзя, так как это прямая дорога к программированию копи-пастом.

- Blind faith (слепая вера)

(строка вместо числа, проверки внешних параметров) : недостаточная проверка корректности входных данных, отсутствие тестирования при разработки кода и исправлении ошибок.



Когда он проявляется

Когда программист думает, что его код всегда будет работать в идеальных условиях, поэтому никогда не выдаст ошибок, или никогда не получит неверные входные данные или данные неверного типа.

Какие проблемы несёт

- Код делает неожидаемые действия.
- Приводит к брешам в безопасности.
- Приводит к каскаду ошибок, что значительно усложняет процесс исправления и возобновления работоспособности.

Причины возникновения

Избыточная доверие к потребителю кода.

Как решить проблему

Ввести правило в разработку, что все лгут, поэтому нельзя доверять никакому коду, даже собственному.

Тут важно не перейти грань и приводить код к анти-паттерну Accidental complexity. Следует помнить про проверку входных данных и возможные проблемы у чужого кода, который используете в проекте.

- God Object (божественный объект) (один класс)

Когда он проявляется

Когда уровень проекта превышает уровень компетенций разработчика

Какие проблемы несёт

- Объект берет на себя слишком много возможностей и/или хранит в себе практически все данные.
- Непереносимость кода.
- Сложно поддерживаемый код

Причинами возникновения

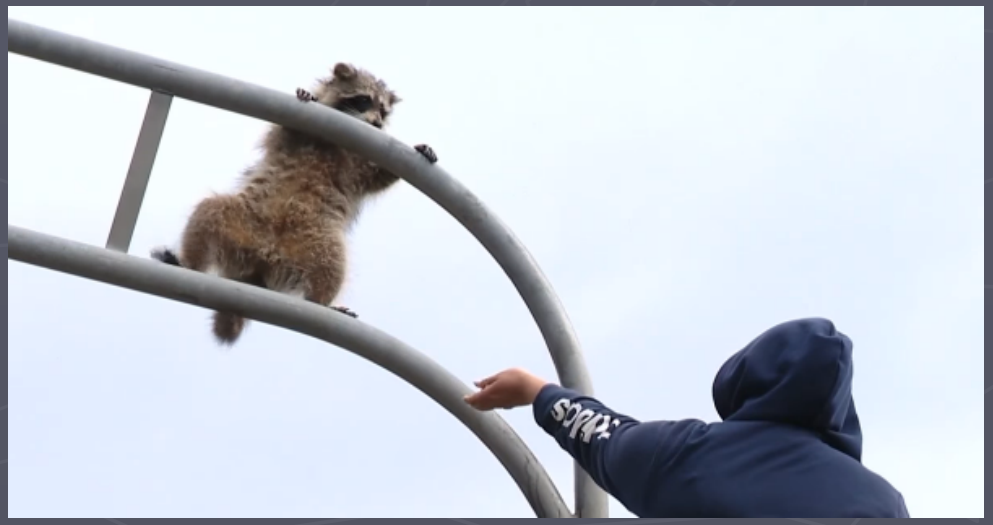
- Плохие знания шаблонов проектирования.
- Низкая компетенция у разработчика.

Как решить проблему

Использовать принципы разработки: DDD, TDD, DRY, KISS, SOLID

Blind faith. Слепая вера

недостаточная проверка корректности входных данных, отсутствие тестирования при разработки кода и исправлении ошибок.



Какие проблемы несёт

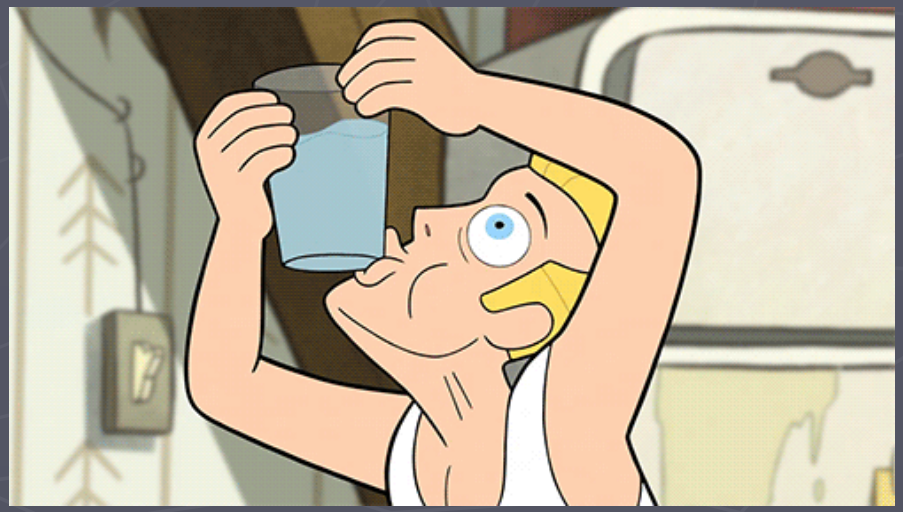
- Код делает неожидаемые действия.
- Приводит к брешам в безопасности.
- Приводит к каскаду ошибок, что значительно усложняет процесс стабилизирования.

Причины возникновения

- Избыточное доверие к юзеру.
- Недостаточное количество Unit и Integration тестов
- Отсутствие валидации на стороне бек-енда (валидация на фронт-енде ничего не стоит)

Programming by permutation (Программирование методом подбора)

Многие неопытные разработчики пытаются решать некоторые задачи методом перебора, подбором параметров, порядка вызова функций, вызов всех методов подряд у third-party библиотек и т.д.



Когда проявляется?

Если программист не понимает происходящего, не разбирается с библиотекой или тем алгоритмом который ему нужно реализовать, как следствие не сможет предусмотреть все варианты развития событий.

Какие проблемы несёт

- Будет потрачено время на решение задачи перебором, а после повторно потратится время на переделку решения.
- Приучает разработчика к тому, что написание кода — это магия, а не инженерная работа.

Причины возникновения

Всё сводится к низкой компетенции разработчика: если программист не может решить задачу несколькими путями — это скорее всего приведёт к появлению этого анти-паттерна.

- \+ новую функцию, которая не добавляется в принятое архитектурное решение;
- есть ошибка, а причины не ясны;
- сложная логика программы

Признаки

- дублирование кода;
- длинный метод;
- большой класс;
- длинный список параметров;
- зависимые функции;
- избыточные временные переменные;
- не сгруппированные данные.

- Введение параметра
- Подъём метода
- Спуск метода
- Перемещение метода
- Замена условного оператора полиморфизмом
- Замена наследования делегированием
- и т .д.

Решение

- Не браться за разработку задачи, в которой не хватает понимания и компетенции до тех пор, пока пробелы не будут закрыты.
- Пообщатся с архитекторами или более опытными разработчиками для прояснения ситуации или совместного поиска решений

Рефакторинг

- (англ. refactoring) или реорганизация кода — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы

Не оптимизация производительности

Не инженеринг

Технический долг

Все люди изначально стараются писать чистый код. Вряд ли найдётся программист, который намеренно плодит грязный код во вред проекту.

Но тогда почему чистый код становится грязным?

Причины появления технического долга

- Давление со стороны бизнеса
- Отсутствие понимания последствий технического долга
- Отсутствие борьбы с жёсткой связанностью компонентов
- Отсутствие авто-тестов
- Отсутствие документации
- Отсутствие взаимодействия между членами команды
- Отложенный рефакторинг
- Отсутствие контроля за соблюдением стандартов
- Отсутствие компетенции

Методы рефакторинга

- Изменение сигнатуры метода
- Инкапсуляция поля
- Выделение класса
- Выделение интерфейса
- Выделение метода
- Встраивание (Inline)
- Введение фабрики

«Чистый код»

- логика прямолинейная
- зависимости — минимальные
- стратегия обработки ошибок
- производительность — близка к оптимальной
- читабельный
- компактный
- решает одну задачу
- невозможно улучшить

Имена в программе

- Содержательные, передавать намерения программиста,

```
int timeInDays;
int creationTime;
int countOfOdd;
```

- не дезинформировать

```
int unix;
int list;
int result;
int XYZControllerForHandString;
int XYZControllerForHandStr;
string l, o;
if (0 == O) (1 == l)
```

- Не удлинять и не умничать

NameString
Name
the … a…. an….
variable
PersonObject
Person

- Между именами должны быть различия

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

- Удобопроизносимые
  - cohint
  - modfgthrerw

- Длина имени должна соответствовать размеру его области видимости.
  - e … - при поиске –большое количество
- Устарели – венгерская запись, handle, m_, CShape…, Ifactory
- Имена методов - глаголы
- Имена классов – существительные
- Содержательный контекст, но не избыточный
  - street, state, city addrState, addrCity
- Одно слово для каждой концепции
  - get , retrieve; insert , add; controller manager

- Избегать цифр

```
List<MilitaryPlane> militaryPlanes2 = new List<MilitaryPlane>();
```

File1, sum3

```
private void _1_OpenHomePageAndSelectOneWayTicket()
```

- Не использовать аббревиатуры

FS, ZG

- Не использовать общие понятия

result, check, sum, count

- Длинные имена

```
int f1(int a1, Collection<Integer> a2)
{
    int a5 = 0;
    for (int a3 = 0; a3 < a2.size() && a3 < a1; a3++)
    {
        int a6 = a2.get(a3);
        if (a6 >= 0)
        {
           Console.WriteLine(a6 + " ");
        }
        else
        {
            a5++;
        }
    }
    Console.WriteLine("");
    return a5;
}
```

```
int doSomethingWithCollectionElements(int numberOfResults,
                                      Collection<Integer> integerCollection)
{
    int resultToReturn = 0;
    for (int variableThatCountsUp = 0;
         variableThatCountsUp < integerCollection.size()
           && variableThatCountsUp < numberOfResults;
         variableThatCountsUp++)
    {
        int integerFromCollection = integerCollection.get(count);
        if (integerFromCollection >= 0)
        {
            Console.WriteLine(integerFromCollection + " ");
        }
        else
        {
            resultToReturn++;
        }
    }
    Console.WriteLine("\n");
    return resultToReturn;
}
```

```
int printFirstNPositive(int n, Collection<Integer> c)
{
    int skipped = 0;
    for (int i = 0; i < c.size() && i < n; i++)
    {
        int maybePositive = c.get(i);
        if (maybePositive >= 0)
        {
            Console.WriteLine(maybePositive + " ");
        }
        else
        {
            skipped++;
        }
    }
    Console.WriteLine("\n");
    return skipped;
}
```

- Используйте рефакторинг

Функции

- Компактность
- Удобна при чтении
- Блоки и отступы
- ФУНКЦИЯ ДОЛЖНА ВЫПОЛНЯТЬ ТОЛЬКО ОДНУ ОПЕРАЦИЮ – Определяется уровнем абстракции

```
public InboxPage readFirstMsgSubjectAndBody()
```

```
public static String testableHtml(
 PageData pageData,
 bool includeSuiteSetup
) throw Exception
{
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
 if (pageData.hasAttribute("Test")) {
 if (includeSuiteSetup) {
 WikiPage suiteSetup =
 PageCrawlerImpl.getInheritedPage(
 SuiteResponder.SUITE_SETUP_NAME, wikiPage
 );
 if (suiteSetup != null) {
 WikiPagePath pagePath =
 suiteSetup.getPageCrawler().getFullPath(suiteSetup);
String pagePathName = PathParser.render(pagePath);
buffer.append("!include -setup .")
 .append(pagePathName)
 .append("\n");
 }
 }
 WikiPage setup =
 PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
 if (setup != null) {
 WikiPagePath setupPath =
 wikiPage.getPageCrawler().getFullPath(setup);
String setupPathName = PathParser.render(setupPath);"
 buffer.append("!include -setup .")
 .append(setupPathName)
 .append("\n");
 }
 }
 buffer.append(pageData.getContent());
 if (pageData.hasAttribute("Test")) {
 WikiPage teardown =
 PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
 if (teardown != null) {
 WikiPagePath tearDownPath =
 wikiPage.getPageCrawler().getFullPath(teardown);
String tearDownPathName = PathParser.render(tearDownPath);
buffer.append("\n")
 .append("!include -teardown .")
 .append(tearDownPathName)
 .append("\n");
 }
 if (includeSuiteSetup) {
 WikiPage suiteTeardown =
 PageCrawlerImpl.getInheritedPage(
 SuiteResponder.SUITE_TEARDOWN_NAME,
 wikiPage
 );
 if (suiteTeardown != null) {
 WikiPagePath pagePath =
 suiteTeardown.getPageCrawler().getFullPath(suiteTeardown);
String pagePathName = PathParser.render(pagePath);
buffer.append("!include -teardown .")
 .append(pagePathName)
 .append("\n");
 return pageData.getHtml();
```

```
public bool setGet(String attribute, String value);
```

- Чем меньше аргументов у функции, тем лучше (0, 1, 2, >3 – плохо, можно заменять объектами)

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

- Возвращать значения через аргументы- плохо
- Аргументы –флаги плохо (сл. Функция выполняет более одной операции)

```
this.render(true);
```

Отделение команд от запроса

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

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

    if (attributeExists("username")) {
 setAttribute("username", "unclebob");
 ...
}
```

Побочные эффекты функций

- неожиданные изменения в переменных класса
- Вредные привязки и зависимости

Используйте исключения вместо возвращения кодов ошибок

```
if (deletePage(page) == E_OK) {
 if (registry.deleteReference(page.name) == E_OK) {
   if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
             logger.log("page deleted");
       } else {
```

```
try {
 deletePage(page);
 registry.deleteReference(page.name);
 configKeys.deleteKey(page.name.makeKey());
 }
catch (Exception e) {
```

- Исключайте дублирование
  - Увеличение кода
  - Больше вероятность ошибки
  - Сложность модификации
- Имя должно соответствовать действию и возвращаемому значению

```
void get(), String print()
```

- Используйте конструкцию глагол-объект для именования методов

```
GetUserId()
```

Комментарии

- Устаревают
  - программисты не могут нормально сопровождать комментарии.
- Вводят в заблуждение и т.п.
- Пишут для запутанного кода
  - Лучше исправить код

```
// Проверить, положена ли работнику полная премия
if ((employee.flags & HOURLY_FLAG) &&
 (employee.age > 65))
//----------------------------------------
if (employee.isEligibleForFullBenefits())
```

Избыточные комментарии

```
/**
* Реализация менеджера, связанная с контейнером.
*/
protected Manager manager = null;
/**
* Кластер, связанный с контейнером.
*/
protected Cluster cluster = null;
/**
* Удобочитаемое имя контейнера.
*/
protected String name = null;
```

- Могут предупреждать и быть информативными
- Удаляйте закомментированный код

- Не использовать /* */
- Не писать //for //if

Форматирование

- 1) понятное название файла
- 2) предпочтительней небольшой размер файла
- 3) вертикальное сжатие

```
foreach (Point a in demo.GetHashSet())
            {

            //Для всех
            //элементов в коллекции выполнит вывод.

                Console.WriteLine(a);
            }
```

```
static void Main(string[] args)
        {
            SuperHashSet<Point> demo = new SuperHashSet<Point>();
            demo.GetHashSet().Add(new Point(3, 5));
            demo.GetHashSet().Add(new Point(2, 9));
            demo.GetHashSet().Add(new Point(2, 9));

            foreach (Point a in demo.GetHashSet())
            {
                Console.WriteLine(a);
            }
```

4) вертикальное разделение концепций

```
static void Main(string[] args)
        {    SuperHashSet<Point> demo = new SuperHashSet<Point>();
            demo.GetHashSet().Add(new Point(3, 5));
            demo.GetHashSet().Add(new Point(2, 9));
            demo.GetHashSet().Add(new Point(2, 9));
            foreach (Point a in demo.GetHashSet())
            { Console.WriteLine(a);    }
```

- 5) тесно связанные друг с другом концепции , должны находиться поблизости друг от друга по вертикали
  - Переменные следует объявлять как можно ближе к месту использования. +зависимые функции
  - Переменные экземпляров должны объявляться в начале класса.

```
public static Test warning(String message){
 ...
 }
private static String exceptionToString(Throwable t){
 ...
 }

private String fName;
private Vector<Test> fTests = new Vector<Test>(10);

public TestSuite() {
}
```

- вызываемая функция должна располагаться ниже вызывающей функции (сверху-вниз).

- Горизонтальное форматирование
  - Длина строки <120 (80) символов (экран)
- Горизонтальное разделение и сжатие
  - Знаки разделяют пробелами
  - Имена функции не отделяются от аргументов
  - И т.д.
- Отступы
  - размер отступа соответствует позиции в иерархии

```
public class FitNesseServer  { private FitNesseContext context;
public FitNesseServer(FitNesseContext context)
{  this.context =
context;
} public void serve(Socket s) { serve(s, 10000); }
public void serve(Socket s, long requestTimeout)
{  try  {          FitNesseExpediter sender = new
FitNesseExpediter(s, context);
        sender.setRequestParsingTimeLimit(requestTimeout);
sender.start();
    }     catch (Exception e) { e.printStackTrace(); }
} }
```

Избыточный код

- Не писать в консоль

```
Console.WriteLine(passengerAirport
                                    .SortByMaxSpeed()
                                    .ToString());
```

- Не создавать лишних переменных

```
String title = driver.Title;
            getUriParser(title);
```

```
boolean res = triangle.check(a,b,c);
if(res) {
```

```
boolean flag = true;
return flag;
```

Инверсия (меньше кода)

```
if ((A < 0 || B < 0 || C < 0) && (B + C > A) || (A + C > B) || (A + B > C))
                return false;
else return true;
```

```
return !((A < 0 || B < 0 || C < 0) && (B + C > A) || (A + C > B) || (A + B > C));
```

```
public boolean isLogOut()
{
    if (!ForgotPassword.isDisplayed())
    {
        return false;
    } //if
    return true;
}
```

```
public boolean isLogOut()
{
    return !ForgotPassword.isDisplayed())

}
```

Исключение

- Сохранять (протоколировать)

```
public boolean isLetterPresentInList(String textToFind) {
    try {
        mailPage.getFirstSubjectThatContains(textToFind);
    } catch (NoSuchElementException e) {

        return false;
    }
    return true;
}
```

- Использовать специфичные исключения

NullSalaryException, NotFoundEmailException

(Не Exception)

Структуры и Объекты

```
 class Point
{
    public double x;
    public double y;
}
//Абстрактная реализация
public interface Point
{
    double X { get; set; }
    double Y { get; set; }
    void setCartesian(double x, double y);
    double getR();
    double getTheta();
  }
```

Объекты - предоставляют поведение и скрывают данные
Структуры данных предоставляют данные, но не обладают значительным поведением

Объекты

- Закон Деметры
  - модуль не должен знать внутреннее устройство тех объектов, с которыми он работает
  - 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

```
String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

- Объекты передачи данных DTO (Data Transfer Object)
  - при работе с базами данных
  - разборе сообщений и др.
- Активные записи (Active Records)
  - разновидность DTO
  - +навигационные методы (save change)

Класс

- Последовательность описания
- 1)открытые статические константы
- 2) приватные статические переменные
- 3) приватные переменные
- 4) защищённые
- 5) открытых обычно нет
- 6) открытые функции
- 7) приватные функции (м.б. около открытых)

Класс

- классы должны быть компактными и еще компактнее
  - Метрика – ответственность - Принцип единой ответственности (SRP1)

```
public class Version
{
 public int getMajorVersionNumber()
 public int getMinorVersionNumber()
 public int getBuildNumber()
}
```

система с множеством малых классов имеет больше «подвижных частей», чем система с несколькими большими классами

```
public class SuperDashboard : JFrame
 public String getCustomizerLanguagePath()
 public void setSystemConfigPath(String systemConfigPath)
 public String getSystemConfigDocument()
 public void setSystemConfigDocument(String
systemConfigDocument)
 public bool getGuruState()
 public bool getNoviceState()
 public bool getOpenSourceState()
 public void showObject(MetaObject object)
 public void showProgress(String s)
 public bool isMetadataDirty()
 public void setIsMetadataDirty(bool isMetadataDirty)
 public Component getLastFocusedComponent()
 public void setLastFocused(Component lastFocused)
 public void setMouseSelectState(boolean isMouseSelected)
 public bool isMouseSelected()
 public LanguageManager getLanguageManager()
 public Project getProject()
 public Project getFirstProject()
 public Project getLastProject()
 public String getNewProjectName()
 public void setComponentSizes(Dimension dim)
 public String getCurrentDir()
 public void setCurrentDir(String newDir)
 public void updateStatus(int dotPos, int markPos)
 public Class[] getDataBaseClasses()
 public MetadataFeeder getMetadataFeeder()
 public void addProject(Project project)
```

- Имя файла должно соответствовать имени классу

```
Project/Folder/Class:
Program/Unit_1/Class.cs
```

- Имя – существительное
- Не называть аббревиатурами
- Не давать не понятных названий

Entity, Instance

- Не использовать цифры

MyClass1

- Связность
  - чем с большим количеством переменных работает метод, тем выше связность этого метода со своим классом
  - разбиение большой функции на много мелких функций - открывает возможность для выделения нескольких меньших классов

- Структура
  - Структура системы должна быть такой, чтобы обновление системы (с добавлением новых или изменением существующих аспектов) создавало как можно меньше проблем

Изоляция

- Чтобы изолировать воздействие подробностей на класс, в систему вводятся интерфейсы и абстрактные классы

```
public interface StockExchange
{
    Money currentPrice(String symbol);
}

public Portfolio {
       private StockExchange exchange;

       public Portfolio(StockExchange exchange){
              this.exchange = exchange;
}
 // ...
```