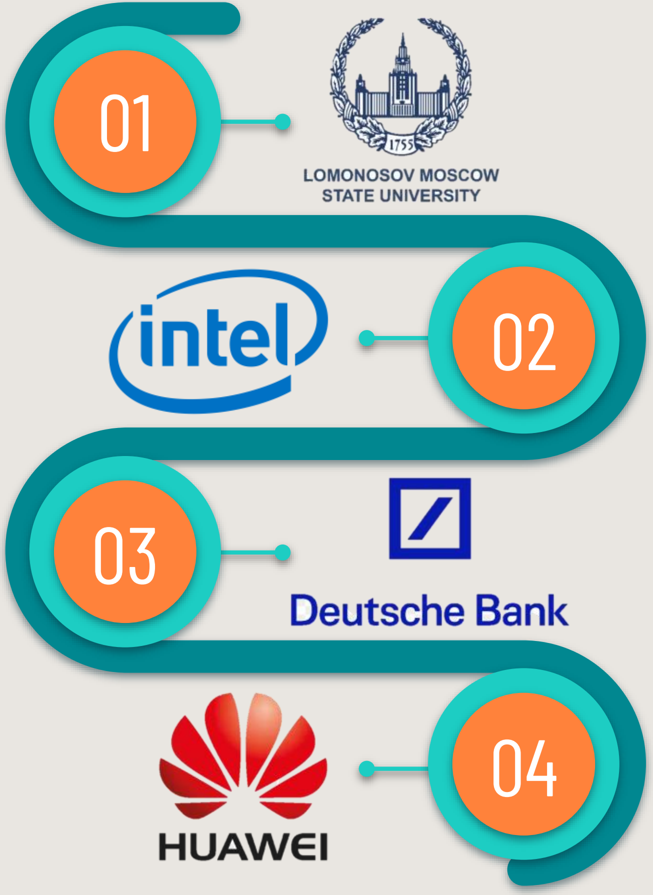
your CODE SMELLS*

or how to avoid issues with OOP

*Main part of a talk will be about Java code, but we will also discuss some basic principles

$ whoami



Now we are here in 2k20

What is a code smell?

In computer programming, a **code smell** is **any characteristic** in the source code of a program **that possibly indicates a deeper problem**. (c) Wikipedia

Application with a good code quality

Reusable and maintainable

- Addition of new features takes little time
- Developer can easily deploy and support application in production

Stable and has less bugs

- Developer can identify problems in production
- App can survive in emergency situations
- Code works as planned, if not - you can workaround it

Readable

- Business functionality can be understood from code. Code becomes better than documentation **for developers**
- Easy for a review process

Sustainable

- Can survive over time with minimal changes (written once – functions for a long time)

How to achieve quality

Obvious things (industry standard)

- **Use design patterns (start from reading Gang Of Four book)**
- Always setup a review process
- **Avoid code smells**
- Use CI/CD and automate as much as possible
- Test your code (at least with unit tests)

Antipattern typical general causes

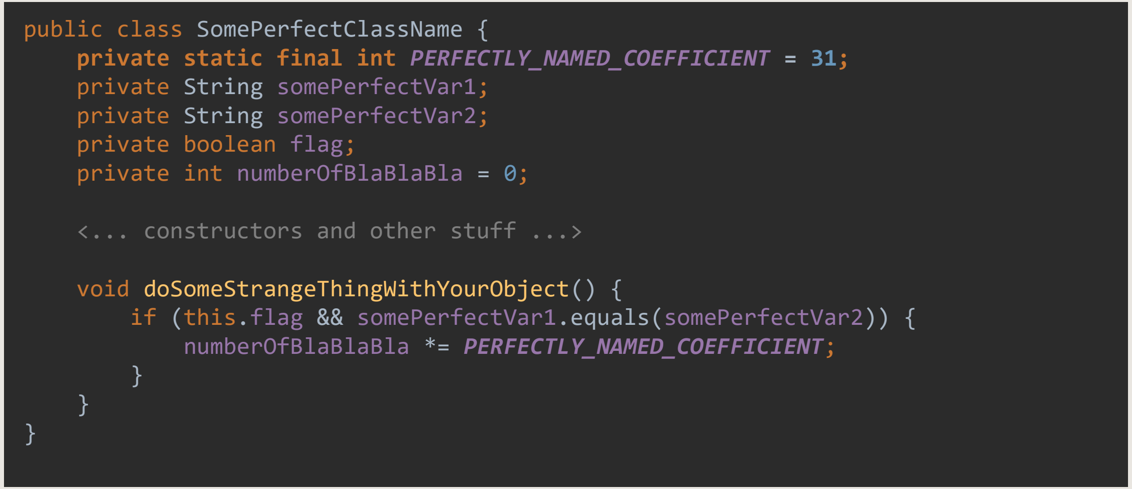
Obvious things

- Lack of an object-oriented architecture
- Lack of (any) architecture
- Lack of architecture enforcement
- Too limited intervention **(fear-driven develoment)**
- Specified disaster

Anti-pattern #1: **Comments – two extremes and balance**

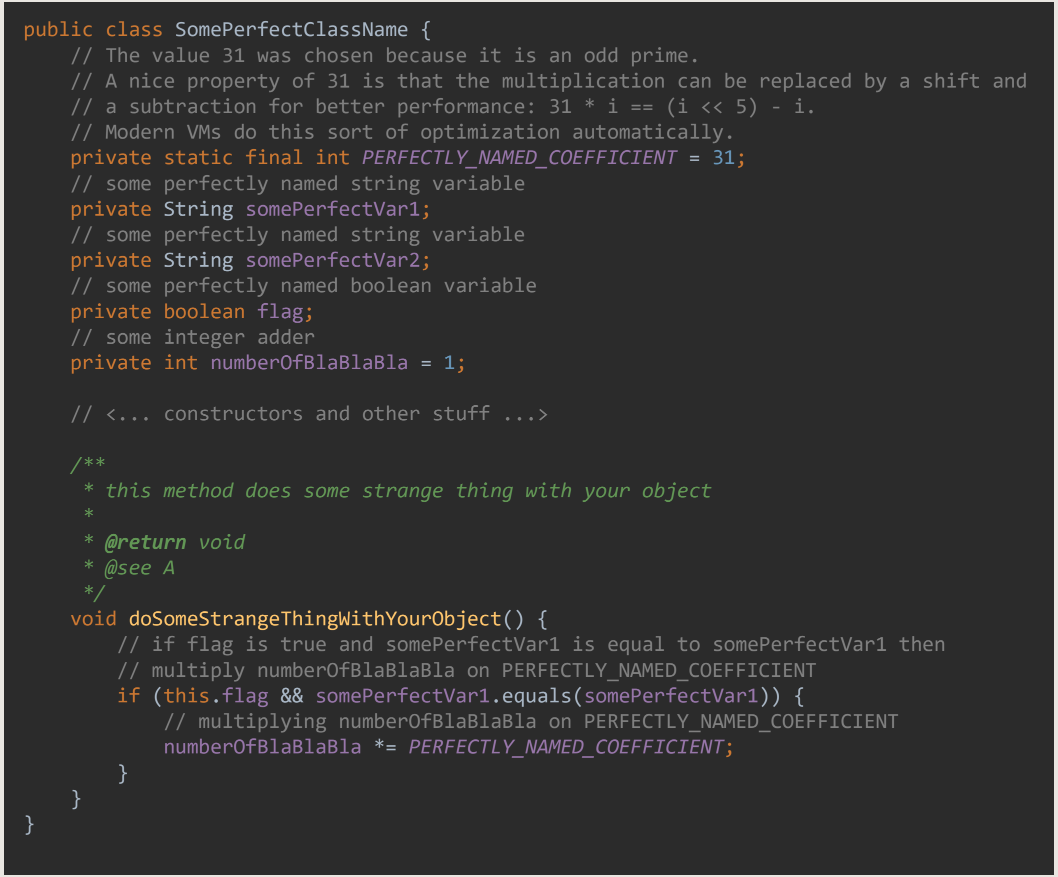
"Self documented" code
Each line has obvious comment
... As all things should be.

Anti-pattern #1: **No comments at all**



```java
public class SomePerfectClassName {
    private static final int PERFECTLY_NAMED_COEFFICIENT = 31;
    private String somePerfectVar1;
    private String somePerfectVar2;
    private boolean flag;
    private int numberOfBlaBlaBla = 0;

    <... constructors and other stuff ...>

    void doSomeStrangeThingWithYourObject() {
        if (this.flag && somePerfectVar1.equals(somePerfectVar2)) {
            numberOfBlaBlaBla *= PERFECTLY_NAMED_COEFFICIENT;
        }
    }
}
```

Anti-pattern #1: **Each line commented**



```java
public class SomePerfectClassName {
    // The value 31 was chosen because it is an odd prime.
    // A nice property of 31 is that the multiplication can be replaced by a shift and
    // a subtraction for better performance: 31 * i == (i << 5) - i.
    // Modern VMs do this sort of optimization automatically.
    private static final int PERFECTLY_NAMED_COEFFICIENT = 31;
    // some perfectly named string variable
    private String somePerfectVar1;
    // some perfectly named string variable
    private String somePerfectVar2;
    // some perfectly named boolean variable
    private boolean flag;
    // some integer adder
    private int numberOfBlaBlaBla = 1;

    // <... constructors and other stuff ...>

    /**
     * this method does some strange thing with your object
     *
     * @return void
     * @see A
     */
    void doSomeStrangeThingWithYourObject() {
        // if flag is true and somePerfectVar1 is equal to somePerfectVar1 then
        // multiply numberOfBlaBlaBla on PERFECTLY_NAMED_COEFFICIENT
        if (this.flag && somePerfectVar1.equals(somePerfectVar1)) {
            // multiplying numberOfBlaBlaBla on PERFECTLY_NAMED_COEFFICIENT
            numberOfBlaBlaBla *= PERFECTLY_NAMED_COEFFICIENT;
        }
    }
}
```

← Easter egg for juniors

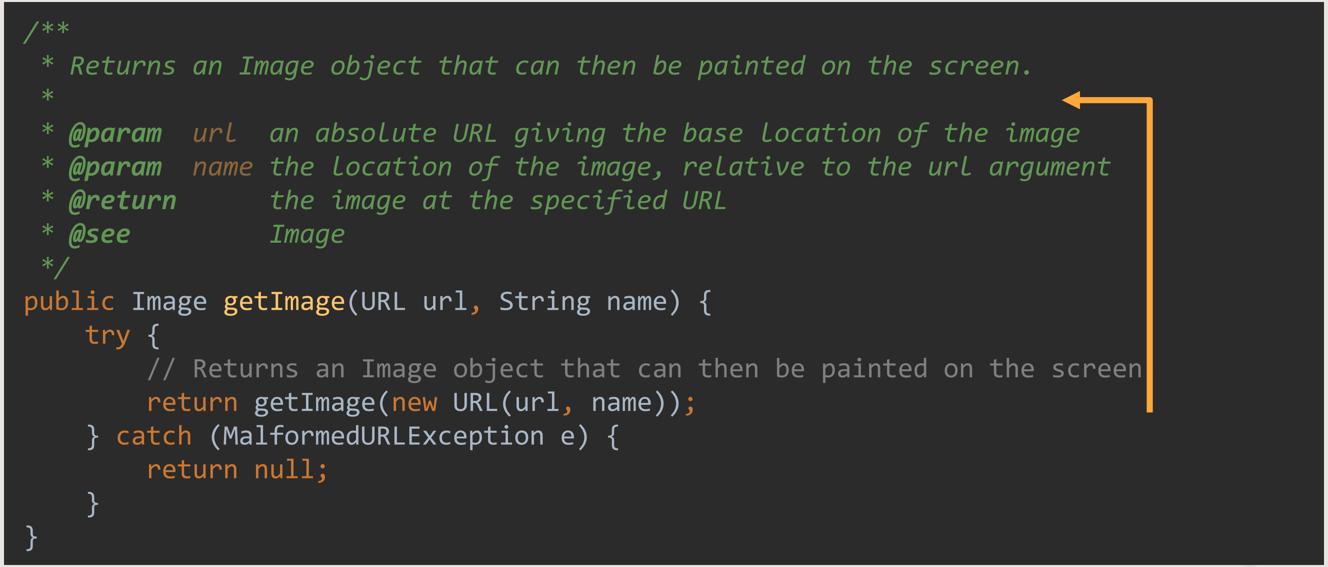
Anti-pattern #1: **Recommendations**

- **Don’t use Russian/Hindi/Chinese/e.t.c**. for comments.
  Use same language that your code is using for keywords.
- Don’t use comments as **an obvious translation for your code**.
- Try to comment **all magic constants** and **all workarounds** you are adding.
- Try to add comments for API you are creating, sometimes it is appropriate to add examples of usage.

Anti-pattern #1: **Recommendations**

- Don’t forget about **javadoc** comments – it should describe **API and functionality** of a method, **pitfalls** and **other code** that you should **@see also**

- No need to put this information inside a method code block

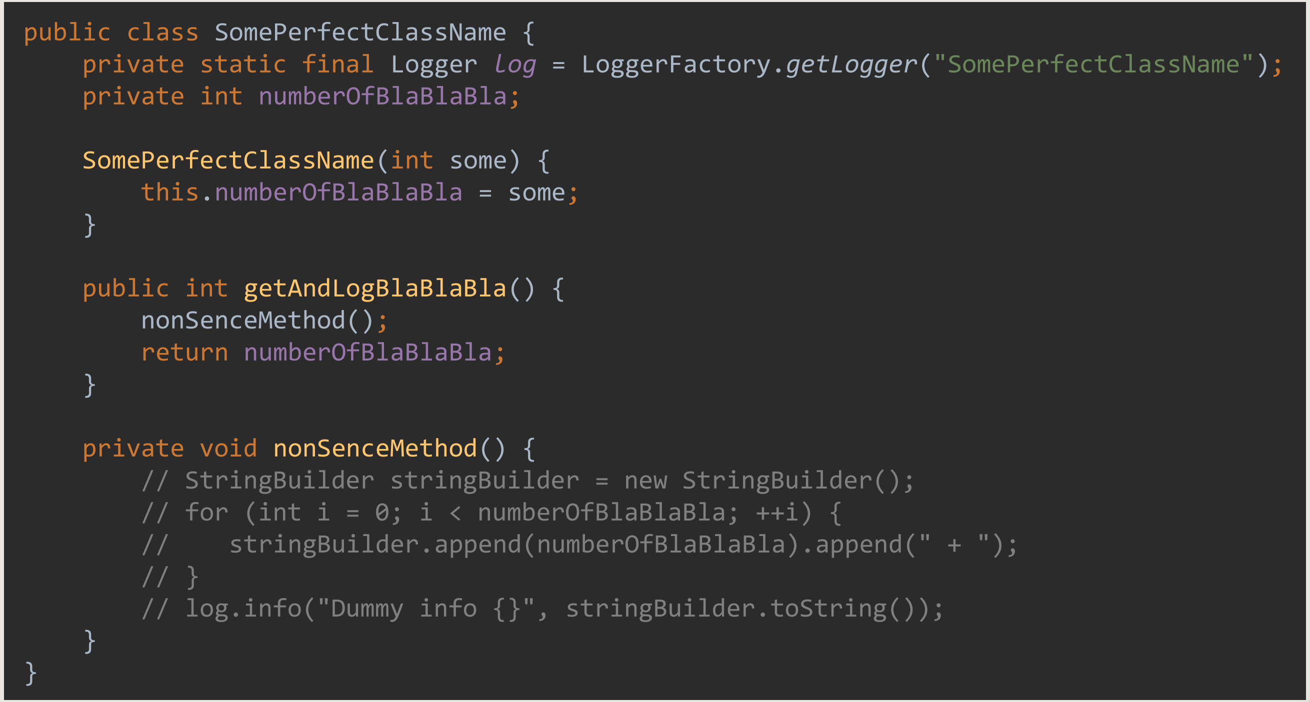


```java
/**
 * Returns an Image object that can then be painted on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
public Image getImage(URL url, String name) {
    try {
        // Returns an Image object that can then be painted on the screen
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

Anti-pattern #1: **Recommendations**

Comments are not a source control:

- avoid commenting of code lines.
  Delete unused code and use version control history if you will need to restore this code in future



```
public class SomePerfectClassName {
    private static final Logger log = LoggerFactory.getLogger("SomePerfectClassName");
    private int numberOfBlaBlaBla;

    SomePerfectClassName(int some) {
        this.numberOfBlaBlaBla = some;
    }

    public int getAndLogBlaBlaBla() {
        nonSenceMethod();
        return numberOfBlaBlaBla;
    }

    private void nonSenceMethod() {
        // StringBuilder stringBuilder = new StringBuilder();
        // for (int i = 0; i < numberOfBlaBlaBla; ++i) {
        //     stringBuilder.append(numberOfBlaBlaBla).append(" + ");
        // }
        // log.info("Dummy info {}", stringBuilder.toString());
    }
}
```

Anti-pattern #1: **Recommendations**

Comments are not a source control:

- Do not add history to comments – you are a developer, not a source control system.

```
// Function: does something
// Copyright Information: Some Company Limited 2015-2018
// Change History: 2015-03-17 12:00 XXX XXXXXXXX Created
//                 2017-03-17 12:00 XXX XXXXXXXX Modified XXX
```

Anti-pattern #2: **D.R.Y. and W.E.T. principles**

- **Don't Repeat Yourself** principle **–** your program should be able to perform it's task and should have small readable code.

- Write Everything Twice/(We enjoy typing) principle – duplicating the logic (Cut-and-Paste Programming)

- **Duplicated code** - the worst and most common industrial code smell (you have that in your code)

- Happens because of:
  - **lazy programmers** that do not know what **inheritance/design patterns/function** is needed for

- **DRY** in rare cases can be harmful (client/server interaction)

Anti-pattern #3: **Long Method/Large Class**

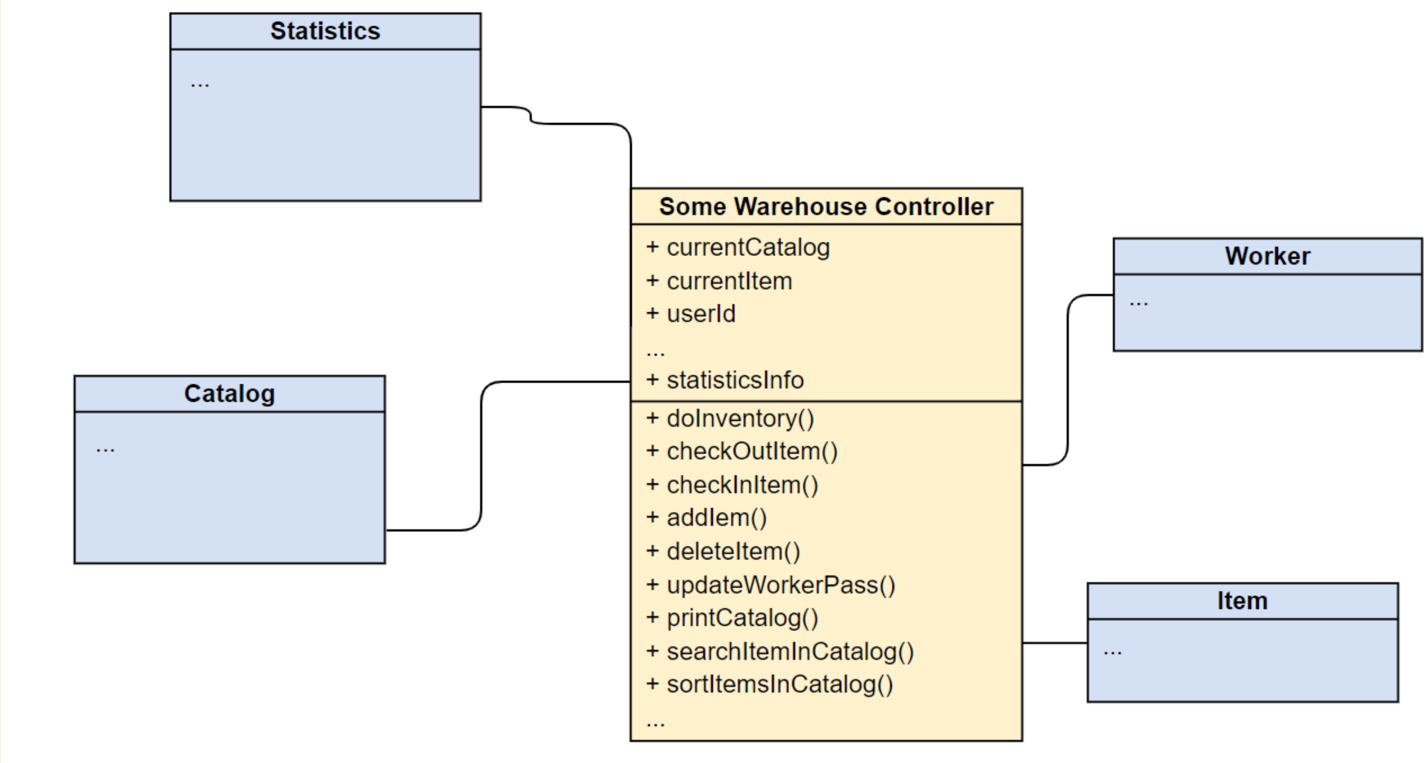
Long method:

- Determining **“how long should method be”** is not a straightforward process.
- Try to split your method that becomes not visible once (~70 lines) on the screen on smaller methods if possible.

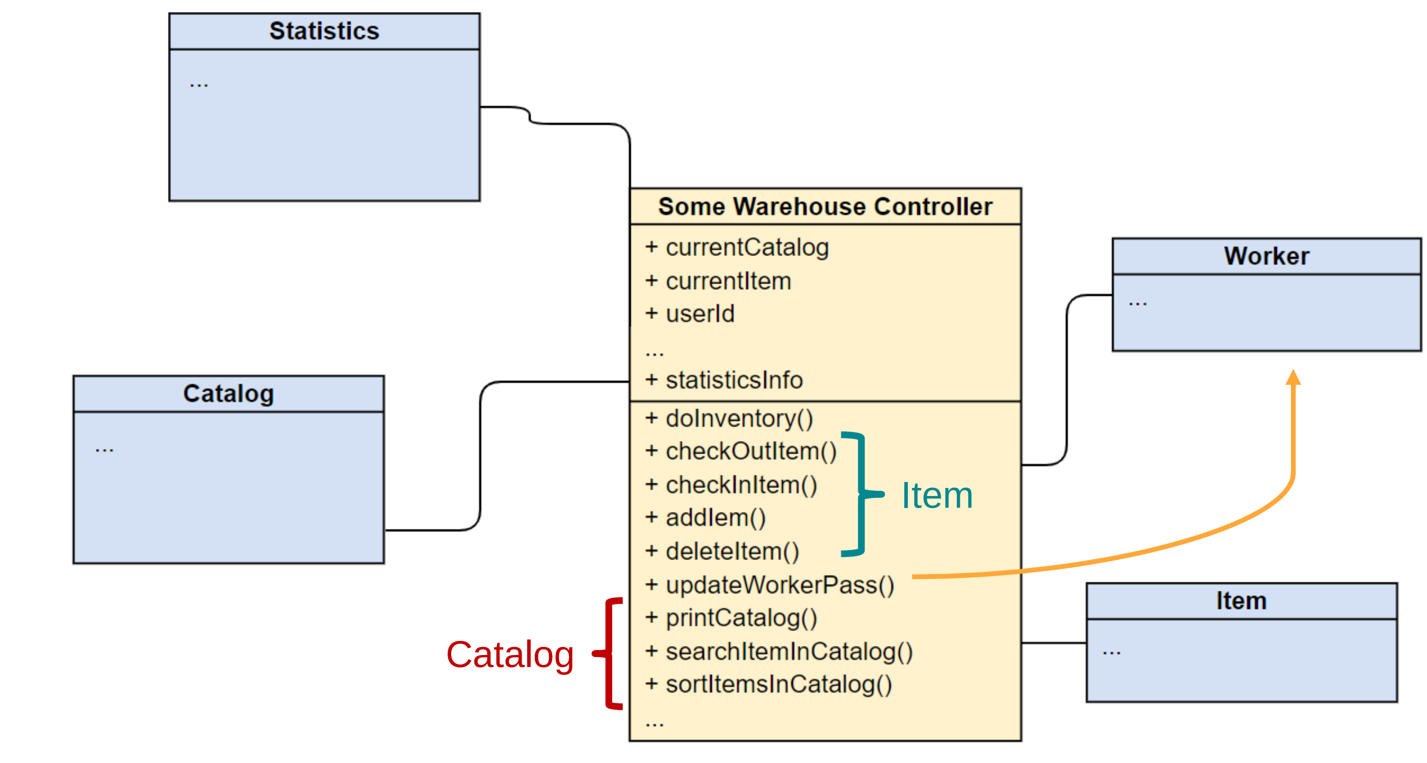
Large class (God class/Blob class):

- Complex controller class surrounded by simple data classes
- Class knows **too much** or does **too much**

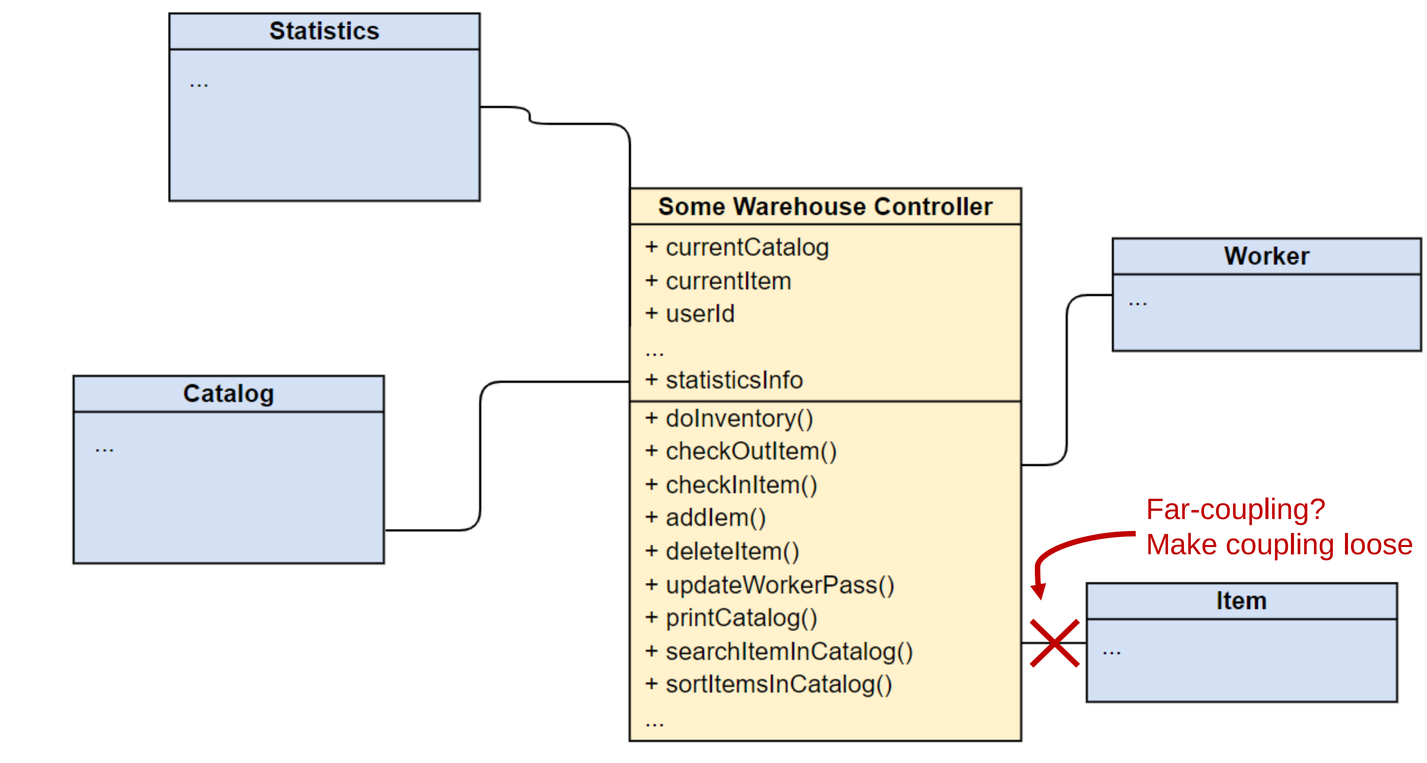
Anti-pattern #3: **Blob class**



Anti-pattern #3: **Blob class**



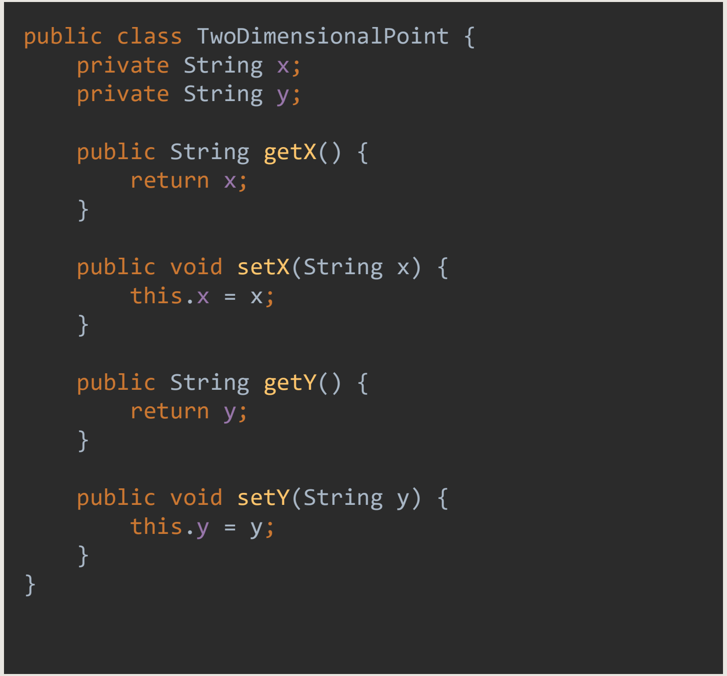
Anti-pattern #3: **Blob class**



Far-coupling?
Make coupling loose

Anti-pattern #4: **Data Class** **(can cause debates)**

Anti-pattern #4: **Data Class**



```java
public class TwoDimensionalPoint {
    private String x;
    private String y;

    public String getX() {
        return x;
    }

    public void setX(String x) {
        this.x = x;
    }

    public String getY() {
        return y;
    }

    public void setY(String y) {
        this.y = y;
    }
}
```

Data class:

- Another opposite extreme of large class.
- Small (in most cases) class that contains only data and no real functionality. Java Beans as an example.
- Breaks the idea of “true OOP” with “rich data models principle”
- BTW: have a look at Scala (case class) and Kotlin (data class)

Anti-pattern #4: **Data Class**

Data classes in mixed languages (FP and OOP):

Scala:

```
case class DataClass1(a: String, b: String)
case class DataClass2(a: String, var b: String)

val someList: List[DataClass2] = List(
 DataClass1(a("1", "2"),
 DataClass1(a("3", "4")
).map(x => DataClass2 (x.a, x.b))
```

*Controlling immutability

Commonly used for modeling in BigData world (Spark) because case classes are **Serializable**

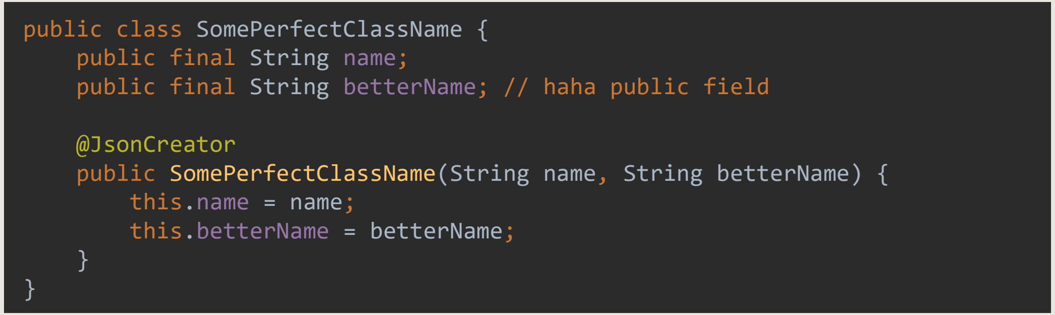
Kotlin:

```
data class DataClass(var a: String, val b: String)
val dtClass = DataClass("1", "2")

fun foo() {
    dtClass.a = “NEW"
}
```

Easy access (omg, where are getters/setters?)

Anti-pattern #4: **Data Class**

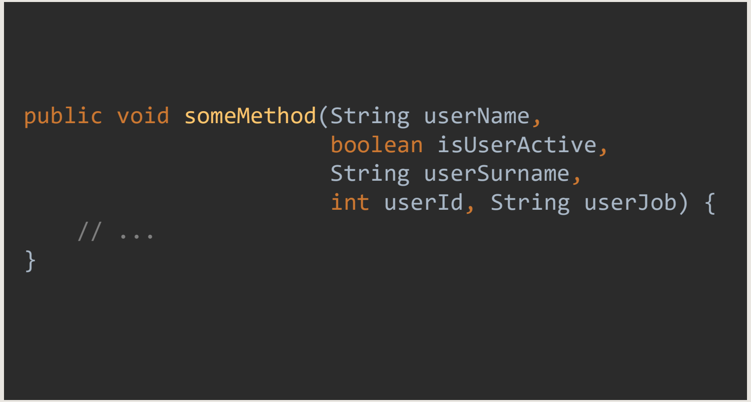
- What to do with perfect libraries for serialization like Jackson?
- Use it! Forget what you have seen on the previous slide.



```java
public class SomePerfectClassName {
    public final String name;
    public final String betterName; // haha public field

    @JsonCreator
    public SomePerfectClassName(String name, String betterName) {
        this.name = name;
        this.betterName = betterName;
    }
}
```

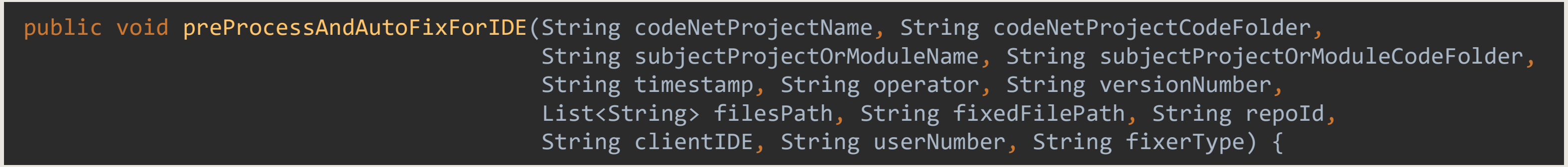
Anti-pattern #5: **Data Clumps**

- Very common code smell
- Group of data that can be unified but anyway is passed to method/class as separate arguments/fields
- Solution: unify in class, but avoid data classes

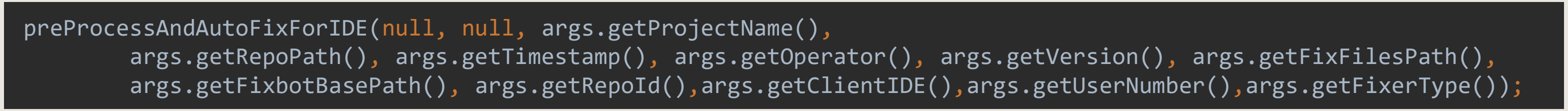


```java
public void someMethod(String userName,
                       boolean isUserActive,
                       String userSurname,
                       int userId, String userJob) {
    // ...
}
```

Anti-pattern #6: **Long Parameter List**



```
public void preProcessAndAutoFixForIDE(String codeNetProjectName, String codeNetProjectCodeFolder,
                                       String subjectProjectOrModuleName, String subjectProjectOrModuleCodeFolder,
                                       String timestamp, String operator, String versionNumber,
                                       List<String> filesPath, String fixedFilePath, String repoId,
                                       String clientIDE, String userNumber, String fixerType) {
```



```
preProcessAndAutoFixForIDE(null, null, args.getProjectName(),
        args.getRepoPath(), args.getTimestamp(), args.getOperator(), args.getVersion(), args.getFixFilesPath(),
        args.getFixbotBasePath(), args.getRepoId(),args.getClientIDE(),args.getUserNumber(),args.getFixerType());
```

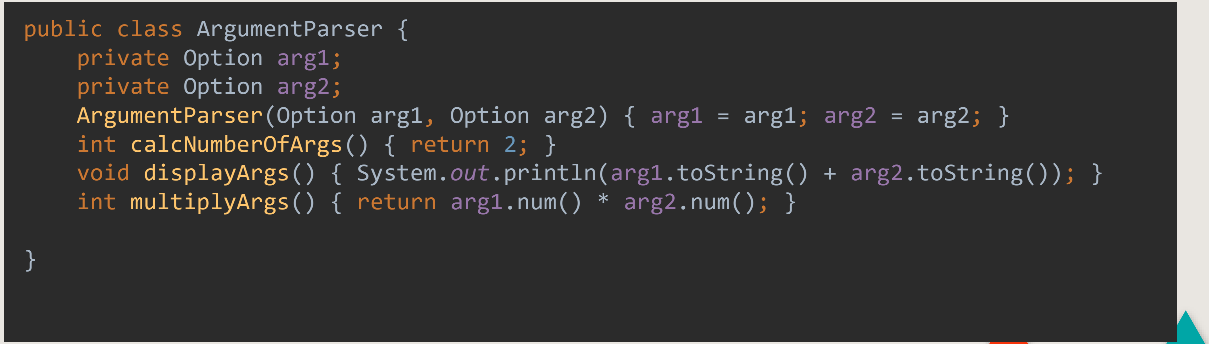
- Very close to Data Clumps problem
- Obvious that it is completely not supportable
- Split the data to subclasses and move the complexity to the function/class body

Anti-pattern #7: **Divergent Class/Divergent Change**

- Divergent change – need to change everything to add one simple thing
- Examples:

  - adding new option or new field in class with hardcoded logic
  - Blob/BlackHole large classes

- Unify behavior, follow single responsibility principle

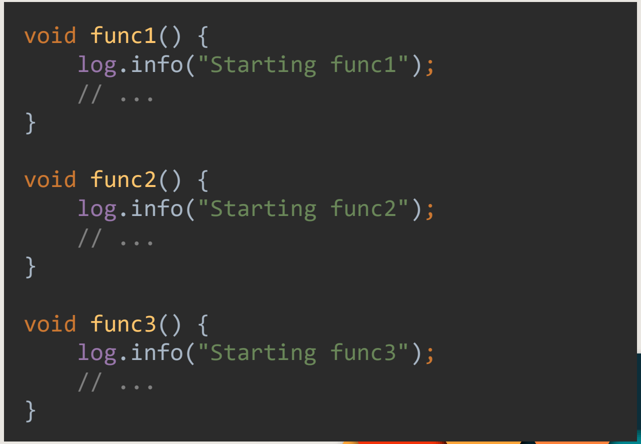


```java
public class ArgumentParser {
    private Option arg1;
    private Option arg2;
    ArgumentParser(Option arg1, Option arg2) { arg1 = arg1; arg2 = arg2; }
    int calcNumberOfArgs() { return 2; }
    void displayArgs() { System.out.println(arg1.toString() + arg2.toString()); }
    int multiplyArgs() { return arg1.num() * arg2.num(); }

}
```

Anti-pattern #8: **Shotgun Surgery**

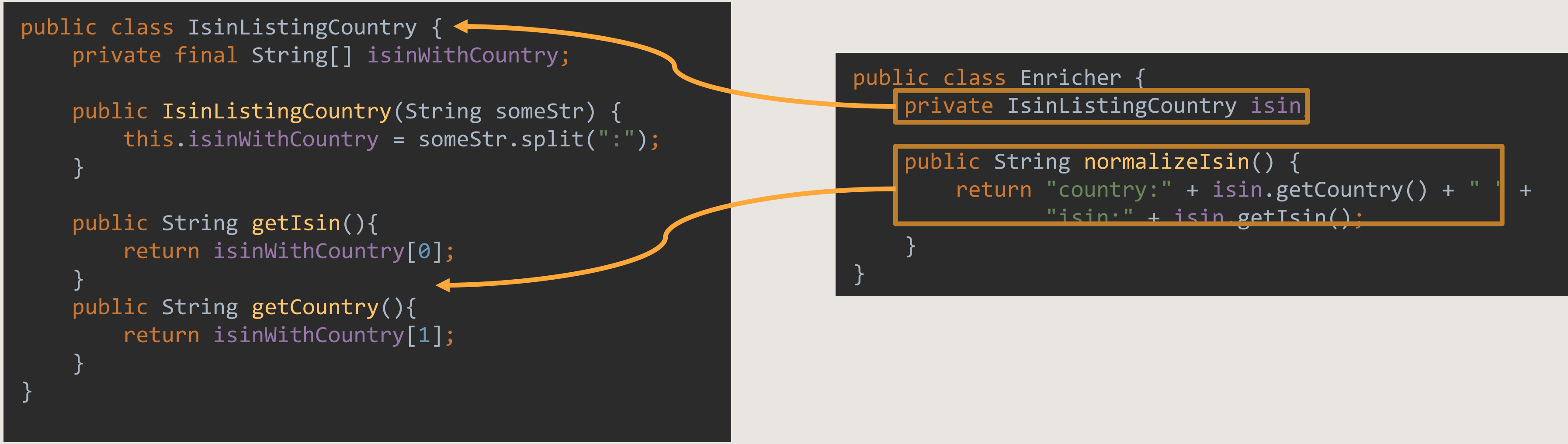
- Small change to class A (method a) causes changes to dozens of other classes (methods)
- Not a ctrl-A + ctrl-V issue
- Normally resolved by unification and move methods to common classes

Classical example



```
void func1() {
    log.info("Starting func1");
    // ...
}

void func2() {
    log.info("Starting func2");
    // ...
}

void func3() {
    log.info("Starting func3");
    // ...
}
```

Anti-pattern #9: **Feature Envy**



```java
public class IsinListingCountry {
    private final String[] isinWithCountry;

    public IsinListingCountry(String someStr) {
        this.isinWithCountry = someStr.split(":");
    }

    public String getIsin(){
        return isinWithCountry[0];
    }
    public String getCountry(){
        return isinWithCountry[1];
    }
}
```

```java
public class Enricher {
   private IsinListingCountry isin;

   public String normalizeIsin() {
       return "country:" + isin.getCountry() + " " +
              "isin:" + isin.getIsin();
   }
}
```

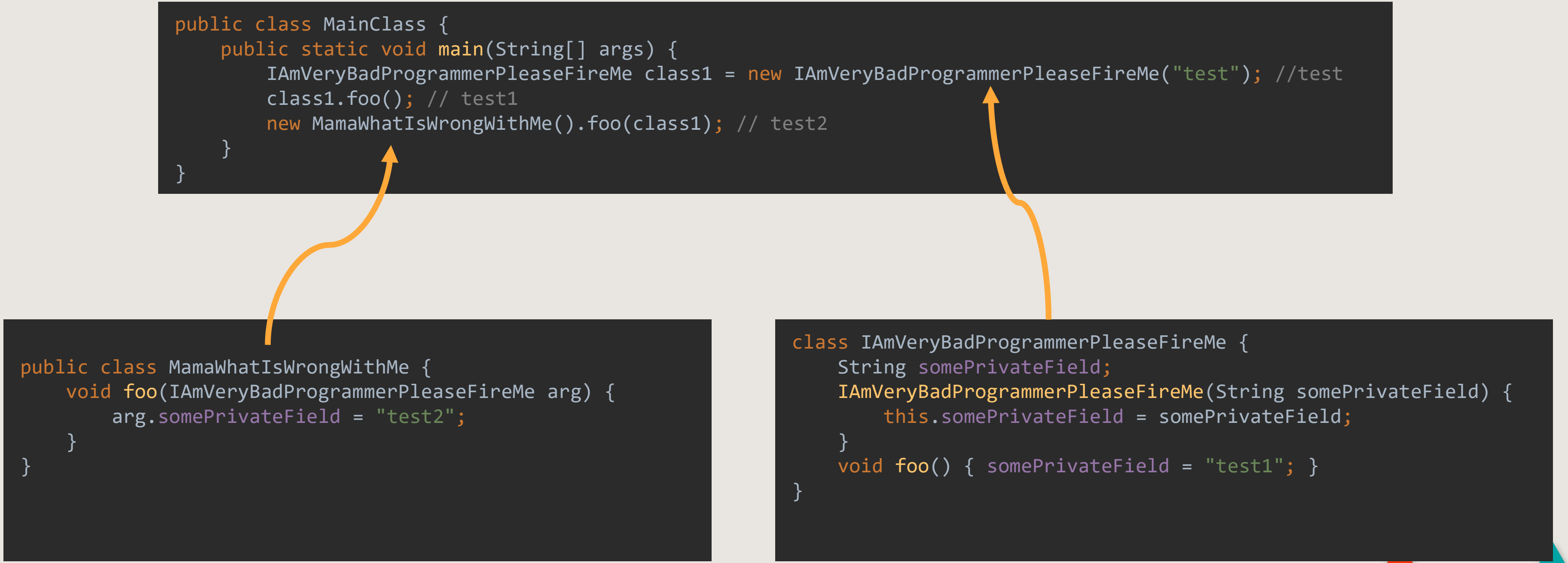
- Method/class is more interested in the details of other class than the one it is in.
- “Romeo and Juliet” pattern
- Don’t refactor if it was done intentionally (Strategy/Visitor pattern)

Anti-pattern #10: **Inappropriate Intimacy**

- Close to **Feature Envy**, but much worse as it breaks OOP main principle (encapsulation)
- One class uses some internal (but not marked private) fields and methods of other class
- Most extreme case:

```
Field f = obj.getClass().getDeclaredField(“somePrivateField");
f.setAccessible(true);
```

Anti-pattern #10: **Inappropriate Intimacy**



```
public class MainClass {
    public static void main(String[] args) {
        IAmVeryBadProgrammerPleaseFireMe class1 = new IAmVeryBadProgrammerPleaseFireMe("test"); //test
        class1.foo(); // test1
        new MamaWhatIsWrongWithMe().foo(class1); // test2
    }
}
```

```
public class MamaWhatIsWrongWithMe {
    void foo(IAmVeryBadProgrammerPleaseFireMe arg) {
        arg.somePrivateField = "test2";
    }
}
```

```
class IAmVeryBadProgrammerPleaseFireMe {
    String somePrivateField;
    IAmVeryBadProgrammerPleaseFireMe(String somePrivateField) {
        this.somePrivateField = somePrivateField;
    }
    void foo() { somePrivateField = "test1"; }
}
```

Anti-pattern #11: **Message Chains**

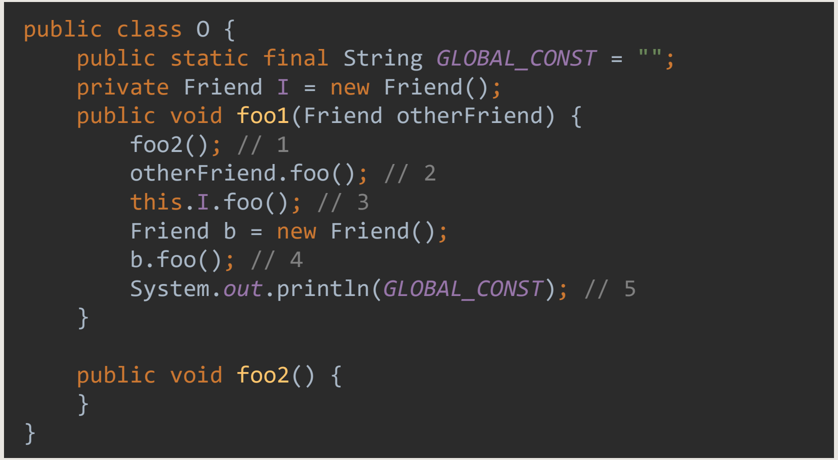
Law of Demeter (in OOP) or a Principle of Least Knowledge

Anti-pattern #11: **Message Chains**

Law of Demeter (in OOP)

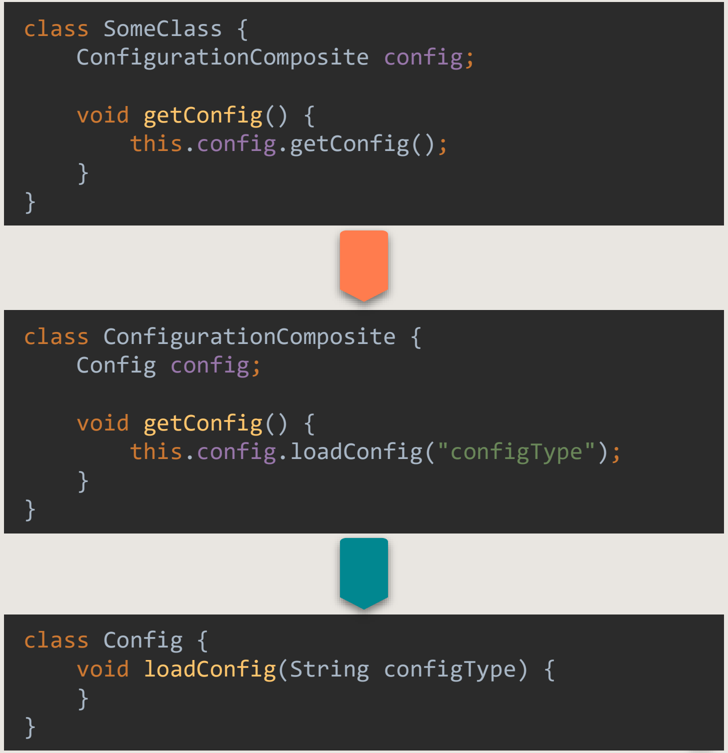
Each program module (read as “method”) M of an object O should be able to call only methods that:

- are encapsulated inside object O
- belong to parameters (arguments) of method M
- belong to other objects, created in the scope of M
- belong to direct component objects of O
- global variables from O



```java
public class O {
    public static final String GLOBAL_CONST = "";
    private Friend I = new Friend();
    public void foo1(Friend otherFriend) {
        foo2(); // 1
        otherFriend.foo(); // 2
        this.I.foo(); // 3
        Friend b = new Friend();
        b.foo(); // 4
        System.out.println(GLOBAL_CONST); // 5
    }

    public void foo2() {
    }
}
```

Anti-pattern #11: **Message Chains**

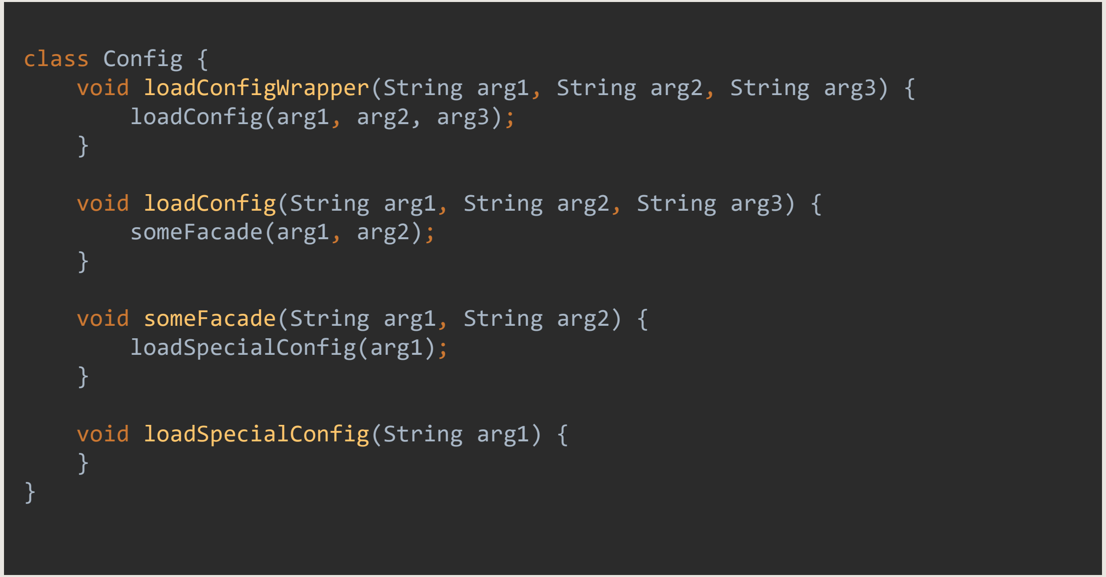


```
class SomeClass {
    ConfigurationComposite config;

    void getConfig() {
        this.config.getConfig();
    }
}
```

```
class ConfigurationComposite {
    Config config;

    void getConfig() {
        this.config.loadConfig("configType");
    }
}
```

```
class Config {
    void loadConfig(String configType) {
    }
}
```

Anti-pattern #11: **Message Chains**



```
class Config {
    void loadConfigWrapper(String arg1, String arg2, String arg3) {
        loadConfig(arg1, arg2, arg3);
    }

    void loadConfig(String arg1, String arg2, String arg3) {
        someFacade(arg1, arg2);
    }

    void someFacade(String arg1, String arg2) {
        loadSpecialConfig(arg1);
    }

    void loadSpecialConfig(String arg1) {
    }
}
```

Anti-pattern #12: **Primitive Obsession**

- Class has fields with **primitive types** for solving small tasks:

```
String name; int amountGBP; String phoneNumber;
long timeInMillis; String currency; String rangeLeft;
String rangeRight; int amountUsd;
```

- Multiple fields that are used as **constants** for coding information:

```
private static final int ADMIN_ROLE = 1;
private static final int USER_ROLE = 2;
private static final int GUEST_ROLE = 3;
```

- Fields with primitive types are used to control state

```
boolean counterParty;
boolean borrow;
boolean market;
boolean client;
```

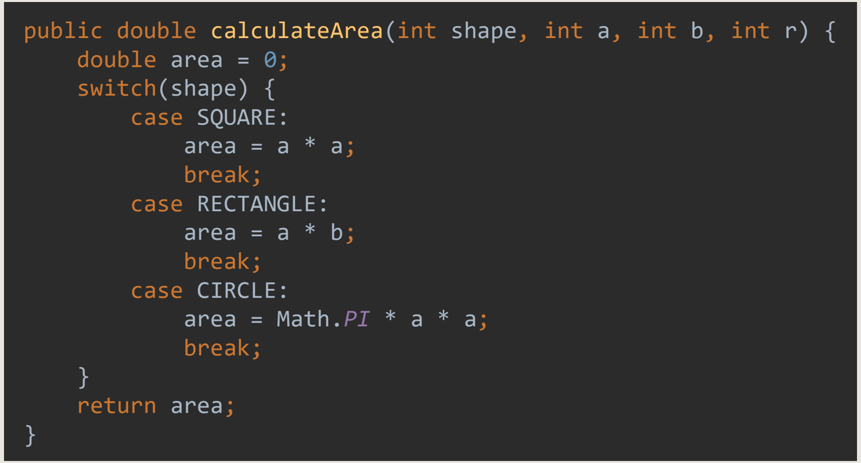
- **Non-OOP way of thinking**

Anti-pattern #13: **Switch Statements**

- Switch statements are a code smells that occur when switch statements are scattered through out a program.

- If a switch is changed, then the others must be found and updated as well.

- What if we have perimeter/e.t.c switch?



```java
public double calculateArea(int shape, int a, int b, int r) {
    double area = 0;
    switch(shape) {
        case SQUARE:
            area = a * a;
            break;
        case RECTANGLE:
            area = a * b;
            break;
        case CIRCLE:
            area = Math.PI * a * a;
            break;
    }
    return area;
}
```

Anti-pattern #14: **Speculative Generality**

- Remember about the **balance**!

- No need to make a superclass, interface, or code that is not needed at the time, but that may (~~or may not~~) be useful someday

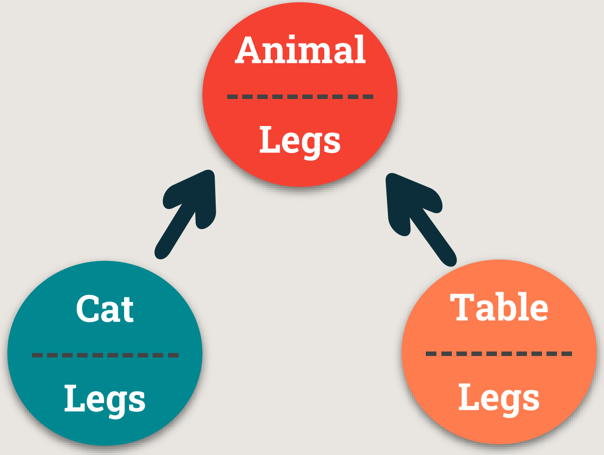
- Software changes frequently

- Follow the idea of **Just in Time Design**

The key to life is balance!

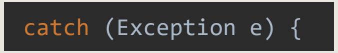
Anti-pattern #15: **Refused Bequest**

- Occurs when a subclass inherits something but does not need it
- Someone was motivated to create inheritance between classes only by the desire to reuse the code in a superclass
- Cat has “legs”, table has “legs” – **cat == table?**



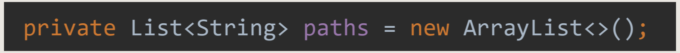
Java-specific problems

- Don’t catch all exceptions:



```
catch (Exception e) {
```

- Please try to initialize simple collections in the beginning of the class (works for 90% of cases):



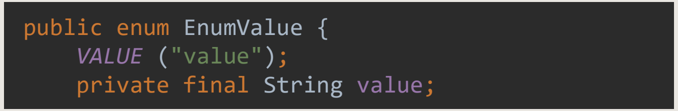
```
private List<String> paths = new ArrayList<>();
```

- Avoid of using static methods, try to follow OOP style if you write on OOP-language

- Try to handle checked exceptions on the lowest level or use “throws”

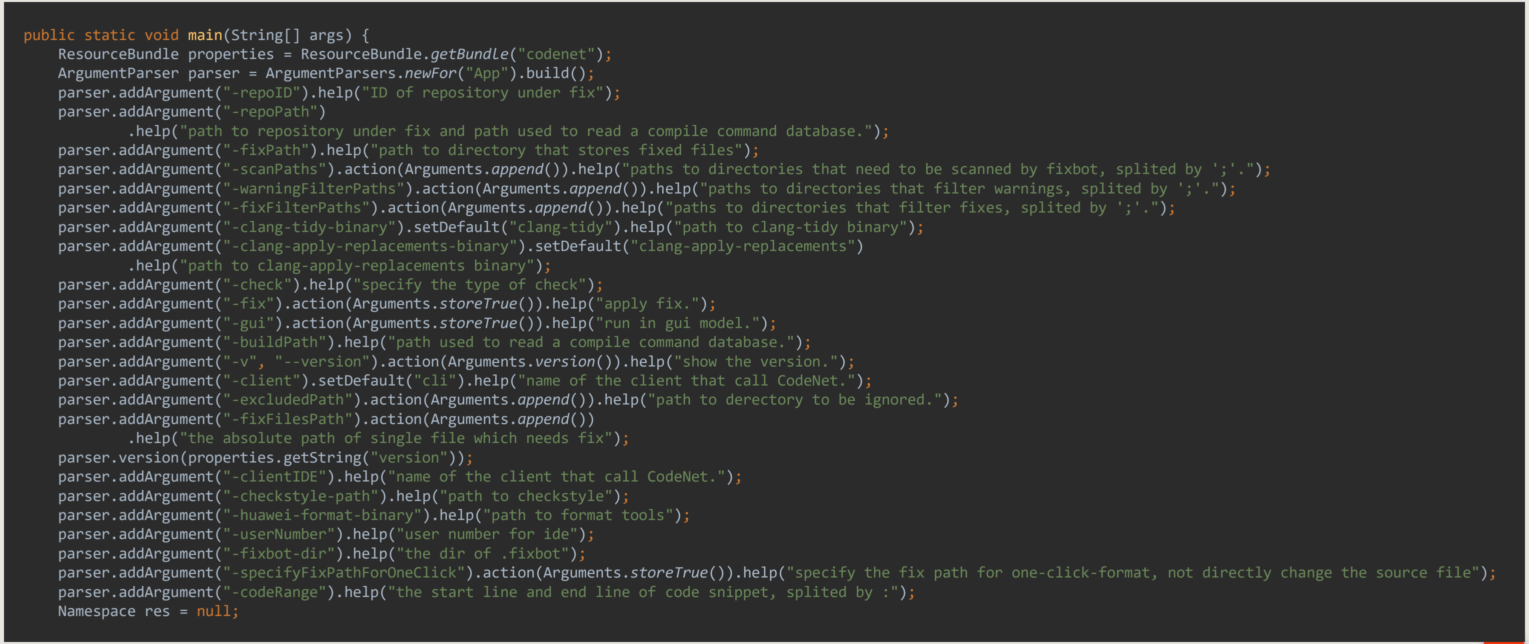
Java-specific problems

- Please use enums instead of hardcoded constants or static final variables:



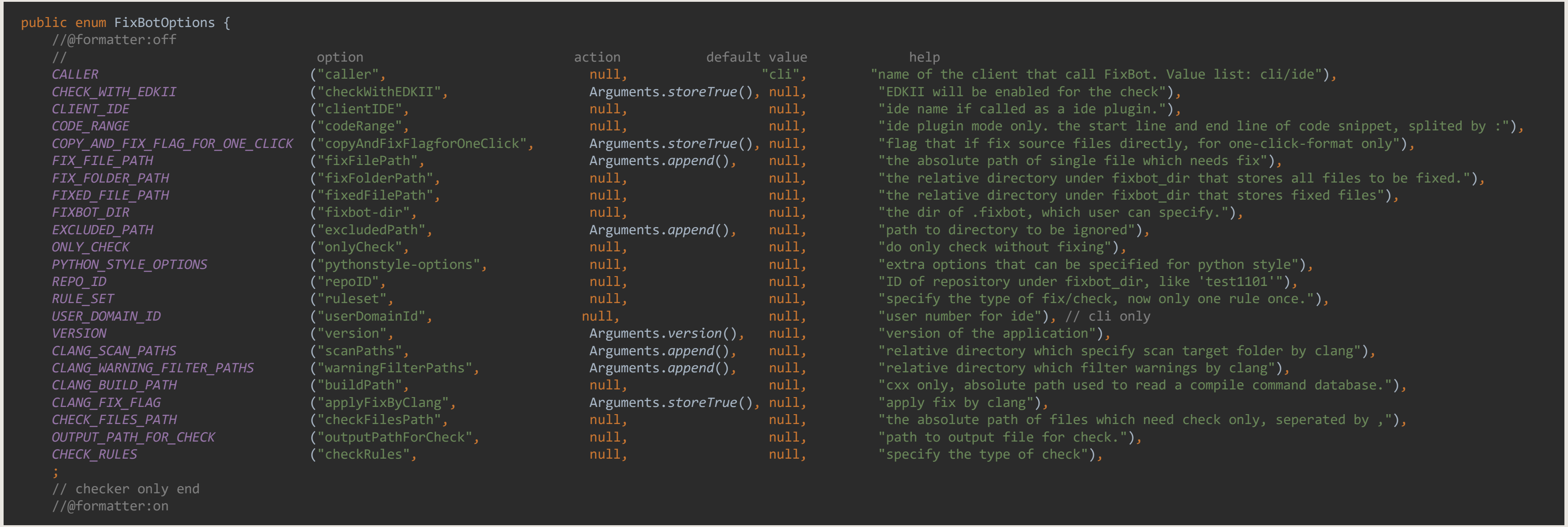
```
public enum EnumValue {
    VALUE ("value");
    private final String value;
```

Java-specific problems



```java
public static void main(String[] args) {
    ResourceBundle properties = ResourceBundle.getBundle("codenet");
    ArgumentParser parser = ArgumentParsers.newFor("App").build();
    parser.addArgument("-repoID").help("ID of repository under fix");
    parser.addArgument("-repoPath")
            .help("path to repository under fix and path used to read a compile command database.");
    parser.addArgument("-fixPath").help("path to directory that stores fixed files");
    parser.addArgument("-scanPaths").action(Arguments.append()).help("paths to directories that need to be scanned by fixbot, splited by ';'.");
    parser.addArgument("-warningFilterPaths").action(Arguments.append()).help("paths to directories that filter warnings, splited by ';'.");
    parser.addArgument("-fixFilterPaths").action(Arguments.append()).help("paths to directories that filter fixes, splited by ';'.");
    parser.addArgument("-clang-tidy-binary").setDefault("clang-tidy").help("path to clang-tidy binary");
    parser.addArgument("-clang-apply-replacements-binary").setDefault("clang-apply-replacements")
            .help("path to clang-apply-replacements binary");
    parser.addArgument("-check").help("specify the type of check");
    parser.addArgument("-fix").action(Arguments.storeTrue()).help("apply fix.");
    parser.addArgument("-gui").action(Arguments.storeTrue()).help("run in gui model.");
    parser.addArgument("-buildPath").help("path used to read a compile command database.");
    parser.addArgument("-v", "--version").action(Arguments.version()).help("show the version.");
    parser.addArgument("-client").setDefault("cli").help("name of the client that call CodeNet.");
    parser.addArgument("-excludedPath").action(Arguments.append()).help("path to derectory to be ignored.");
    parser.addArgument("-fixFilesPath").action(Arguments.append())
            .help("the absolute path of single file which needs fix");
    parser.version(properties.getString("version"));
    parser.addArgument("-clientIDE").help("name of the client that call CodeNet.");
    parser.addArgument("-checkstyle-path").help("path to checkstyle");
    parser.addArgument("-huawei-format-binary").help("path to format tools");
    parser.addArgument("-userNumber").help("user number for ide");
    parser.addArgument("-fixbot-dir").help("the dir of .fixbot");
    parser.addArgument("-specifyFixPathForOneClick").action(Arguments.storeTrue()).help("specify the fix path for one-click-format, not directly change the source file");
    parser.addArgument("-codeRange").help("the start line and end line of code snippet, splited by :");
    Namespace res = null;
```

Java-specific problems

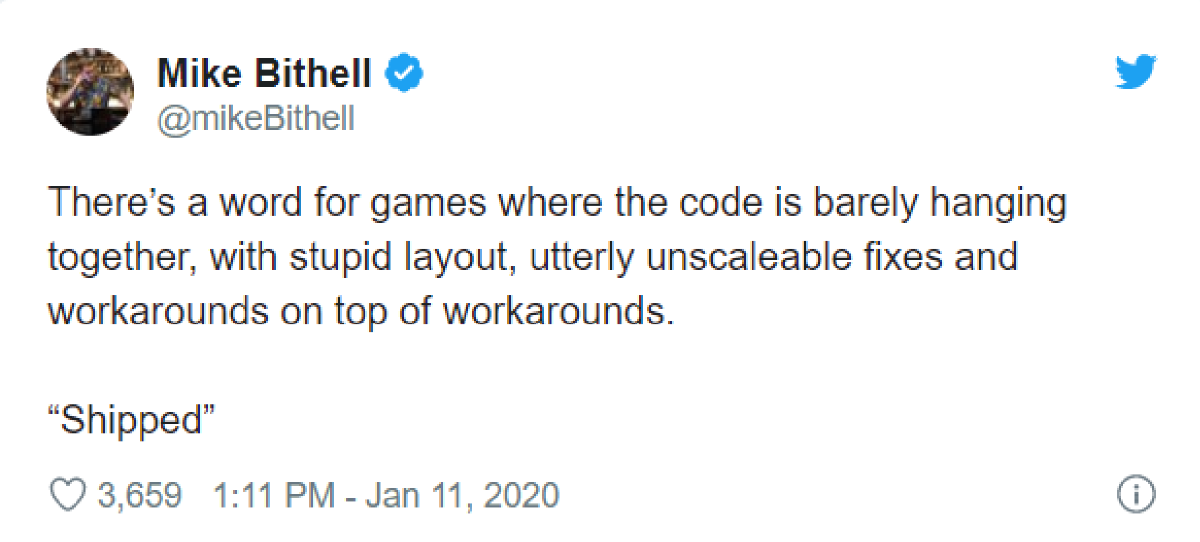


```
public enum FixBotOptions {
    //@formatter:off
    //                                option                           action           default value             help
    CALLER                           ("caller",                          null,                 "cli",        "name of the client that call FixBot. Value list: cli/ide"),
    CHECK_WITH_EDKII                 ("checkWithEDKII",                  Arguments.storeTrue(), null,         "EDKII will be enabled for the check"),
    CLIENT_IDE                       ("clientIDE",                       null,                  null,         "ide name if called as a ide plugin."),
    CODE_RANGE                       ("codeRange",                       null,                  null,         "ide plugin mode only. the start line and end line of code snippet, splited by :"),
    COPY_AND_FIX_FLAG_FOR_ONE_CLICK  ("copyAndFixFlagforOneClick",       Arguments.storeTrue(), null,         "flag that if fix source files directly, for one-click-format only"),
    FIX_FILE_PATH                    ("fixFilePath",                     Arguments.append(),    null,         "the absolute path of single file which needs fix"),
    FIX_FOLDER_PATH                  ("fixFolderPath",                   null,                  null,         "the relative directory under fixbot_dir that stores all files to be fixed."),
    FIXED_FILE_PATH                  ("fixedFilePath",                   null,                  null,         "the relative directory under fixbot_dir that stores fixed files"),
    FIXBOT_DIR                       ("fixbot-dir",                      null,                  null,         "the dir of .fixbot, which user can specify."),
    EXCLUDED_PATH                    ("excludedPath",                    Arguments.append(),    null,         "path to directory to be ignored"),
    ONLY_CHECK                       ("onlyCheck",                       null,                  null,         "do only check without fixing"),
    PYTHON_STYLE_OPTIONS             ("pythonstyle-options",             null,                  null,         "extra options that can be specified for python style"),
    REPO_ID                          ("repoID",                          null,                  null,         "ID of repository under fixbot_dir, like 'test1101'"),
    RULE_SET                         ("ruleset",                         null,                  null,         "specify the type of fix/check, now only one rule once."),
    USER_DOMAIN_ID                   ("userDomainId",                   null,                   null,         "user number for ide"), // cli only
    VERSION                          ("version",                         Arguments.version(),   null,         "version of the application"),
    CLANG_SCAN_PATHS                 ("scanPaths",                       Arguments.append(),    null,         "relative directory which specify scan target folder by clang"),
    CLANG_WARNING_FILTER_PATHS       ("warningFilterPaths",              Arguments.append(),    null,         "relative directory which filter warnings by clang"),
    CLANG_BUILD_PATH                 ("buildPath",                       null,                  null,         "cxx only, absolute path used to read a compile command database."),
    CLANG_FIX_FLAG                   ("applyFixByClang",                 Arguments.storeTrue(), null,         "apply fix by clang"),
    CHECK_FILES_PATH                 ("checkFilesPath",                  null,                  null,         "the absolute path of files which need check only, seperated by ,"),
    OUTPUT_PATH_FOR_CHECK            ("outputPathForCheck",              null,                  null,         "path to output file for check."),
    CHECK_RULES                      ("checkRules",                      null,                  null,         "specify the type of check"),
    ;
    // checker only end
    //@formatter:on
```

In the end

Don’t invent your own bicycles!
ALWAYS use existing solutions

In the end



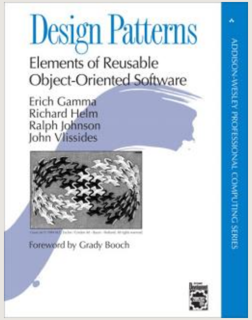
Mike Bithell
@mikeBithell

There's a word for games where the code is barely hanging together, with stupid layout, utterly unscaleable fixes and workarounds on top of workarounds.

"Shipped"

♡ 3,659 1:11 PM - Jan 11, 2020

Must read



Design Patterns

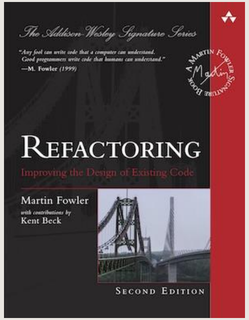
Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design patterns by the Gang of four



The Addison-Wesley Signature Series

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler *(1999)*

A MARTIN FOWLER SIGNATURE BOOK

REFACTORING

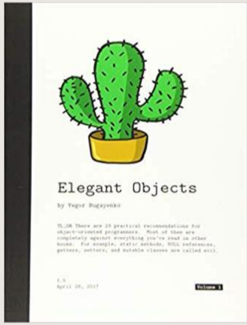
Improving the Design of Existing Code

Martin Fowler
with contributions by
Kent Beck

SECOND EDITION

Refactoring by Martin Fowler

Must read, buy and discuss



Elegant Objects

by Yegor Bugayenko

TL;DR There are 23 practical recommendations for object-oriented programmers. Most of them are completely against everything you've read in other books. For example, static methods, NULL references, getters, setters, and mutable classes are called evil.

1.5
April 28, 2017

Volume 1

Elegant Objects by Egor Bugaenko (Huawei)

This guy

Useful web resources:

- https://sourcemaking.com/
- https://refactoring.guru/

Thanks and let’s connect on linkedIn