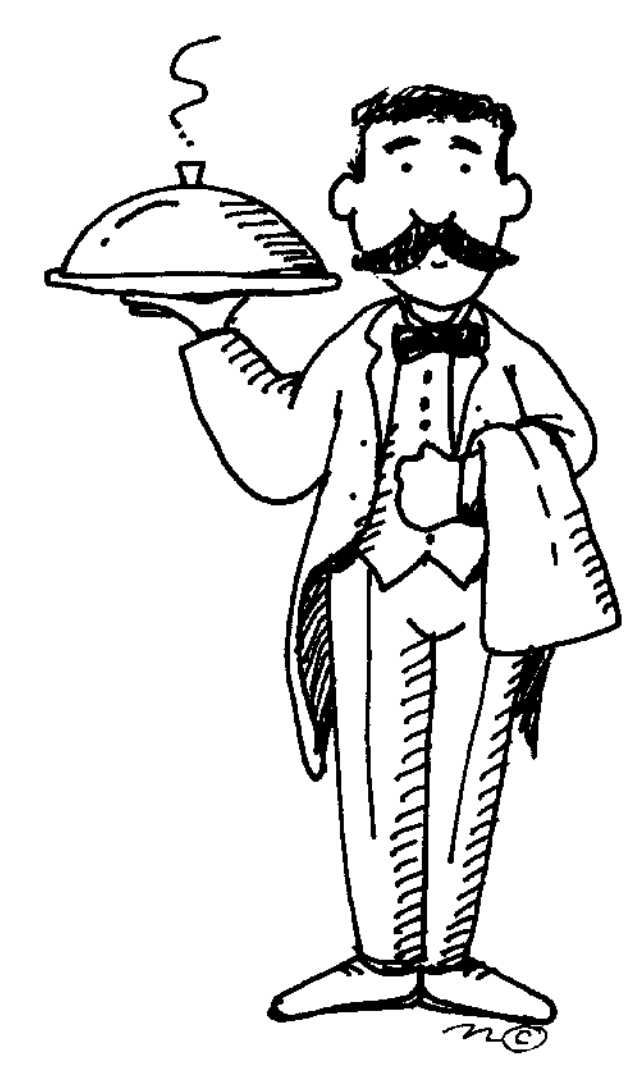
Clean Code*

* Or why is more important how we write code rather what we write

Assoc. prof. Catalin Boja, Asist. Bogdan Iancu, Lect. Alin Zamfiroiu

Which are the discussion topics

- Why clean code?
- Principles
- Naming conventions
- Clean Code in practice
- Short dictionary
- Instruments
- Bonus



Just Clean

Japanese workplace organization methodology called 5S:

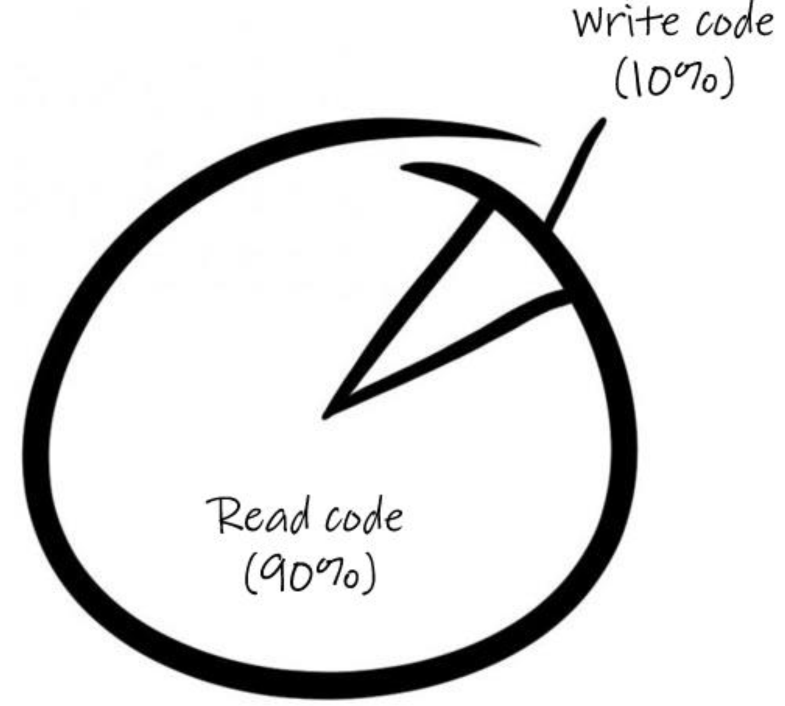
- Seiri, or organization
- Seiton, or tidiness (think "systematize" in English).
- Seiso, or cleaning (think "shine" in English)
- Seiketsu, or standardization.
- Shutsuke, or discipline (self-discipline).

https://www.accuform.com/Plant-Facility/safety-slip-gard-floor-sign-PSR732

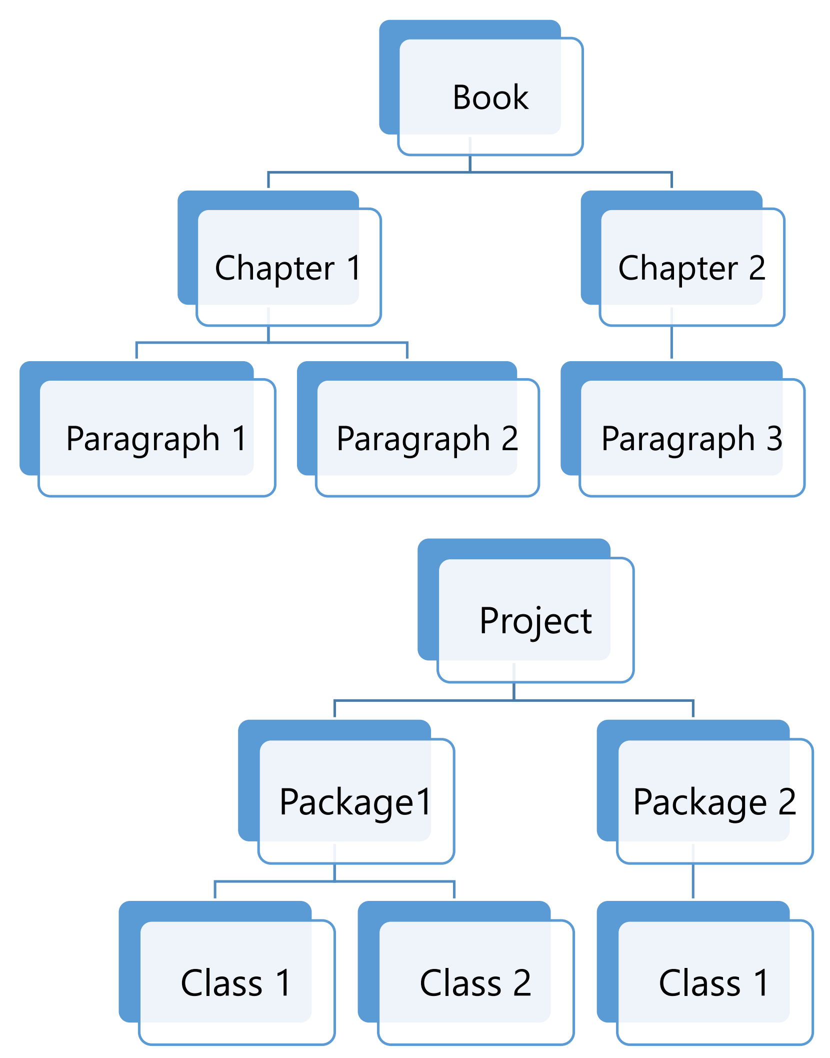
James O. Coplien, Foreward in Robert C. Martin (Uncle Bob) – Clean Code: A Handbook of Agile Software Craftsmanship

Why Clean Code?

- Programming is not what to tell the computer what to do
- Programming is what to tell to another human what we want the computer to do
- Unfortunately sometimes the "other man" is ourselves
- Eventually we become authors
- There is no time to be lazy
- Otherwise we can become a noun

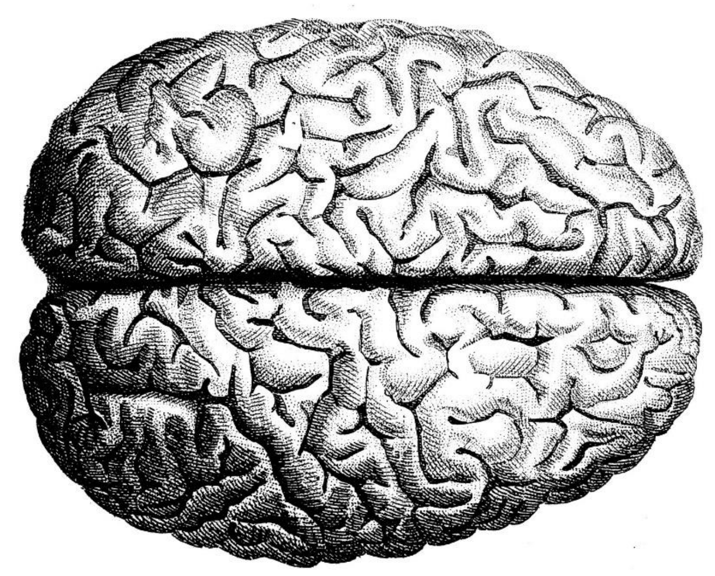


Programmer = Writer



One more thing

- When we read our brains behaves like a code compiler
- Based on scientific studies most humans can focus simultaneously on just 7 elements (±2)
- Rubber Duck Programming (Debugging)

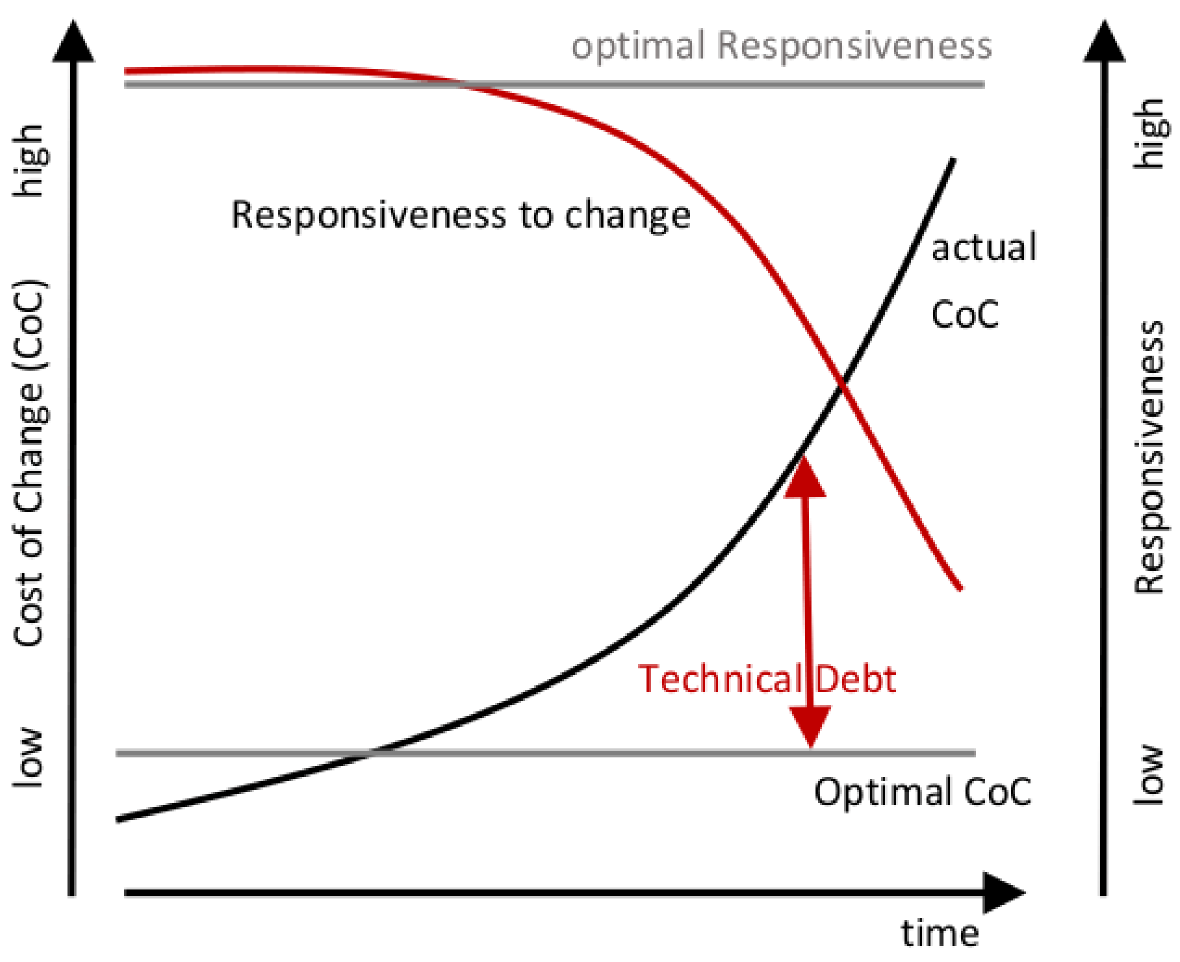


What is Clean Code

- Code must be easily readable
- Code must be easily understandable
- The code should be easily modified
- ... by anyone

http://www.codeoclock.com/2015/03/03/clean-up/

Advantages



http://www.chudovo.com/Blog/CleanCode_General_Principles

What is Good Code

CLEAN Code

=

GOOD Code

GOOD CODERS...

... KNOW WHAT THEY'RE DOING

What is Bad Code

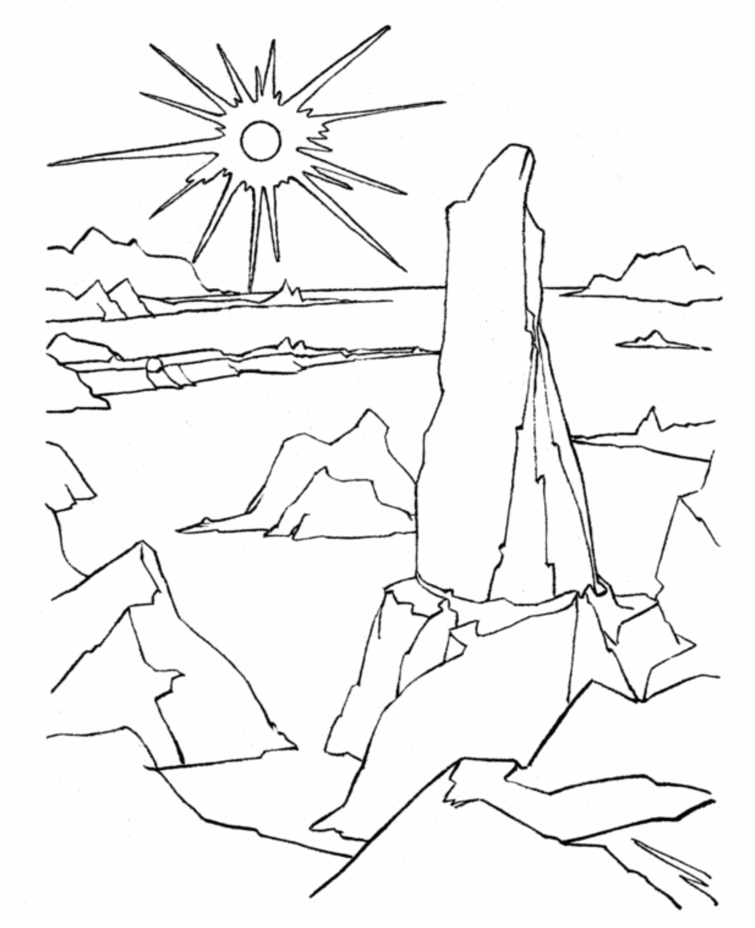
- Hard to read and understand
- Misleading
- It breaks when you change it
- It has several dependencies in external modules - code breaking glass
- Tight coupled with other code sequences

Principles

- DRY
- KISS
- YAGNI
- SOLID

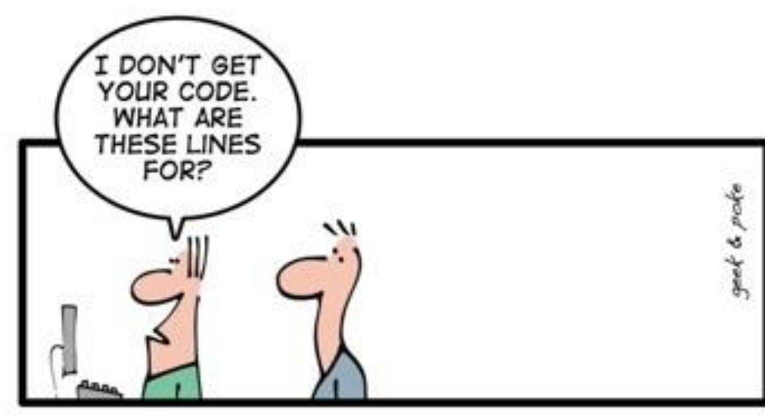
D.R.Y.

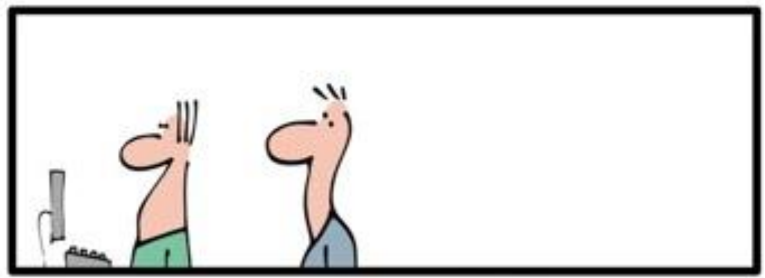
- **D**on't **R**epeat **Y**ourself
- Applicable whenever we Copy / paste a piece of code
- Just use a method or a class



K.I.S.S.

- **K**eep **I**t **S**imple and **S**tupid
- Whenever we want to implement a method to do all things



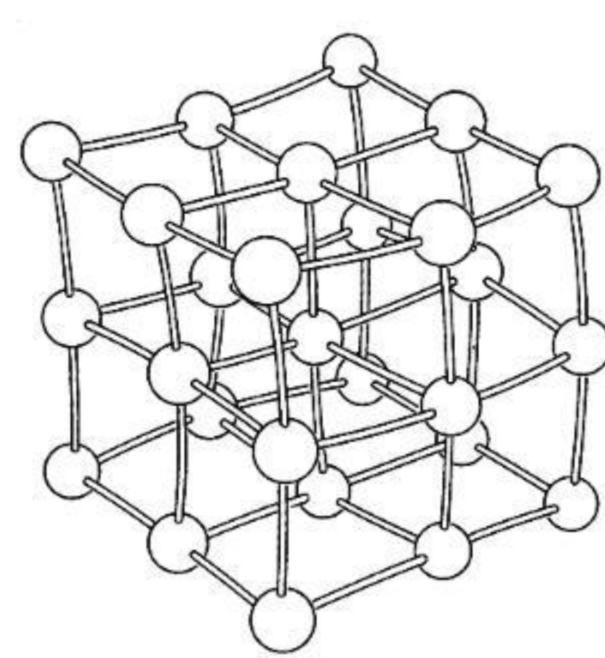


Y.A.G.N.I.

- **Y**ou **A**in't **G**onna **N**eed **I**t
- Don't write methods which are not yet necessary (may be you will never need them)
- Somehow related with KISS

S.O.L.I.D.

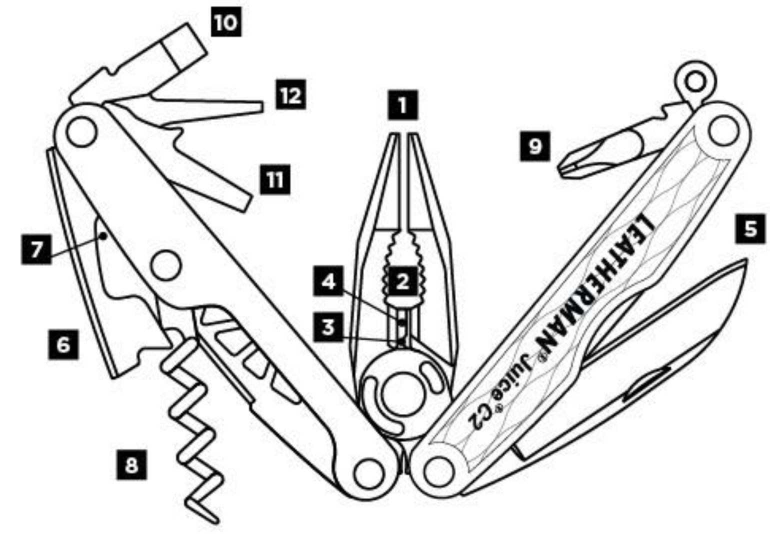
- **S**ingle responsibility (SRP)
- **O**pen-closed (OCP)
- **L**iskov substitution (LSP)
- **I**nterface segregation (ISP)
- **D**ependency inversion



https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

Single Responsability Principle

- A class must always have one and only one responsibility
- Otherwise any change of its specifications will lead to its uselessness and rewriting the entire code
- *A class should have only one reason to change (*Robert C. Martin - Agile Software Development, Principles, Patterns, and Practices)



https://en.wikipedia.org/wiki/Single_responsibility_principle

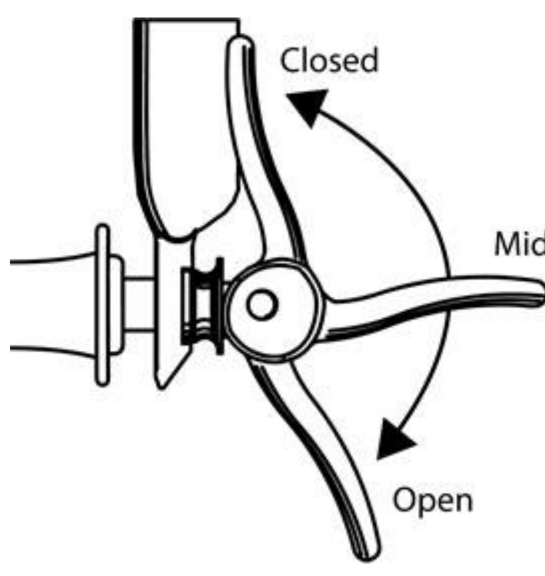
Single Responsability Principle

```
class Student{
	void payTuition(){ }
	void takeOOPExam(){ }
	void saveToDB(){ }
}
```

- A class about a student
- Depends on 3 external factors
  - accounting
  - academic
  - IT department

Open-Close Principle

- Classes must be **open** for extensions
- But closed for changes



https://en.wikipedia.org/wiki/Open/closed_principle

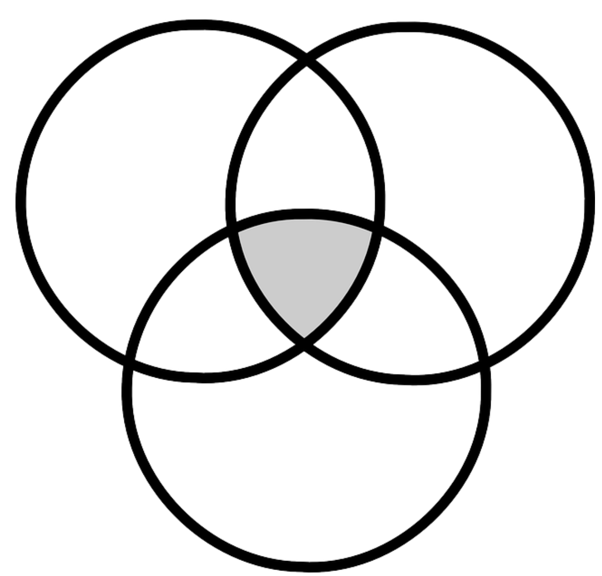
Liskov Substitution Principle

- Objects can be replaced whenever by instances of their derived classes without this affecting functionality
- Also known as „Design by Contract"

https://en.wikipedia.org/wiki/Liskov_substitution_principle

Interface Segregation Principle

- More specialized interfaces are always preferable to a single general interface
- By changing the "contract" you don’t risk to change other clients contracts
- Objects must not be forced to implement methods that are not useful/needed



https://en.wikipedia.org/wiki/Interface_segregation_principle

Interface Segregation Principle

<<Interface>>
Vehicul

startMotor()
accelereaza()
franeaza()
aprindePozitii()
aprindeFaza Scurta()
semnalizeazaDreapta()
semnalizeazaStanga()
schimbaViteza(int)
pornesteRadio()
opresteRadio()
cresteVolum()
scadeVolum()
schimbaPostRadio()
playCD()
nextCD()
prevCD()
stopCD()
pauseCD()
ejectCD()

VS

<<Interface>>
ControlVehicul

startMotor()
accelereaza()
franeaza()
schimbaViteza(int)

<<Interface>>
ControlLumini

aprindePozitii()
aprindeFaza Scurta()
semnalizeazaDreapta()
semnalizeazaStanga()

<<Interface>>
ControlRadio

pornesteRadio()
opresteRadio()
cresteVolum()
scadeVolum()
schimbaPostRadio()

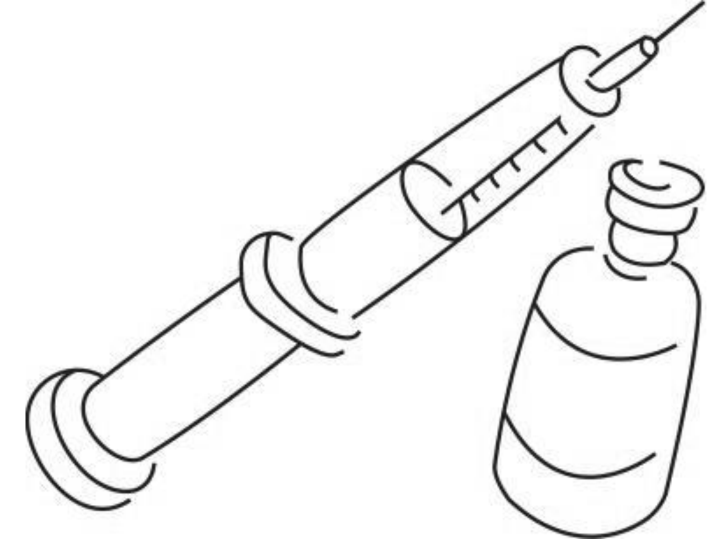
<<Interface>>
ControlCD

playCD()
nextCD()
prevCD()
stopCD()
pauseCD()
ejectCD()

Dependency Inversion Principle

Program to interfaces, not implementations

Depend on abstractions. Do not depend on concrete classes



https://en.wikipedia.org/wiki/Dependency_inversion_principle

What do you think about this code?

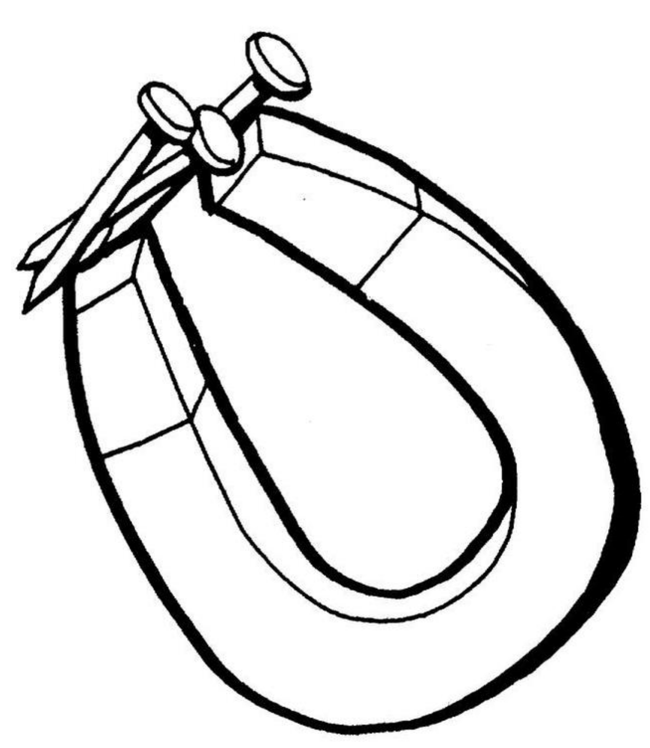
```
public static int Calculeaza() {
    int x = 5;
    ArrayList l = new ArrayList();
    l.add(x);
    int y = 10;
    l.add(y);
    l.add(15);
    int m = 0;
    for(Object k : l) {
        m+= (int)k;
    }
    return m;
}
```

Naming conventions

- UpperCamelCase
- lowerCamelCase
- System Hungarian Notation
- Apps Hungarian Notation

Naming conventions- Classes

- Attention to what names you choose
- Name poorly chosen are a magnet for lazy programmers
- Composed of a specific noun without unnecessary prefixes and suffixes
- We must not forget the Single Responsibility Principle



Naming conventions - Methods

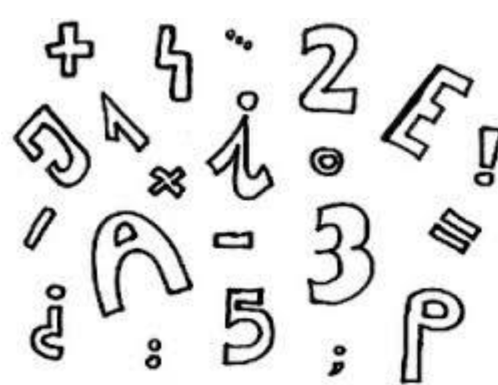
- It must clearly describe what they do
- Where poorly chosen names can not be avoided (automatically generated environment) is indicated that within their body to place only calls to other methods
- If the name of a method contains a conjunction ("and", "or", "or") most likely you need two methods
- *Don't abr met nam*

Naming conventions - Variables

- It's not recommended to use in variable names code strings form other languages (SQL, CSS)
- Boolean variables must sound like a question that can be answered true / false

  isTerminated boolean = false;

- When there are complementary variables, their name must be symmetric



Rules for writing code

```
protected String nume; protected String prenume;protected int varsta;
```

Define each variable on a single line.

```
public class Persoana {
    protected String nume;
    protected String prenume;
    protected int varsta;
```

Rules for writing code

Code blocks start with {

and ends with }.

Even if we have a single

instruction.

```
public double getMedie() {
    return medie;
}
```

Rules for writing code

Instruction blocks are marked using different indentation levels.

```php
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

Rules for writing code

Between function header and the its body starting bracket use a single space

```
public void setMedie(double medie) {
    this.medie = medie;
}
```

Rules for writing code

Write the block closing bracket } on a single line.

Except for *if-else* or *try-catch*.

```
public void setVarsta(int varsta) {
    if (varsta > 0) {
        this.varsta = varsta;
    } else {
        this.varsta = 1;
    }
}
```

Rules for writing code

Methods are separated by a single empty line.

```
public double getMedie() {
    return medie;
}

public void setMedie(double medie) {
    this.medie = medie;
}
```

Rules for writing code

Function arguments are separated by a

, (comma) and a space.

```
public Persoana(String nume, String prenume, int varsta) {
```

Rules for writing code

Use a space ( ) between operators and operands.

```
return nume + " " + prenume;
```

The exceptions are the unary operators.

Clean Code Rules for conditional structures

- Avoid comparisons with true and false

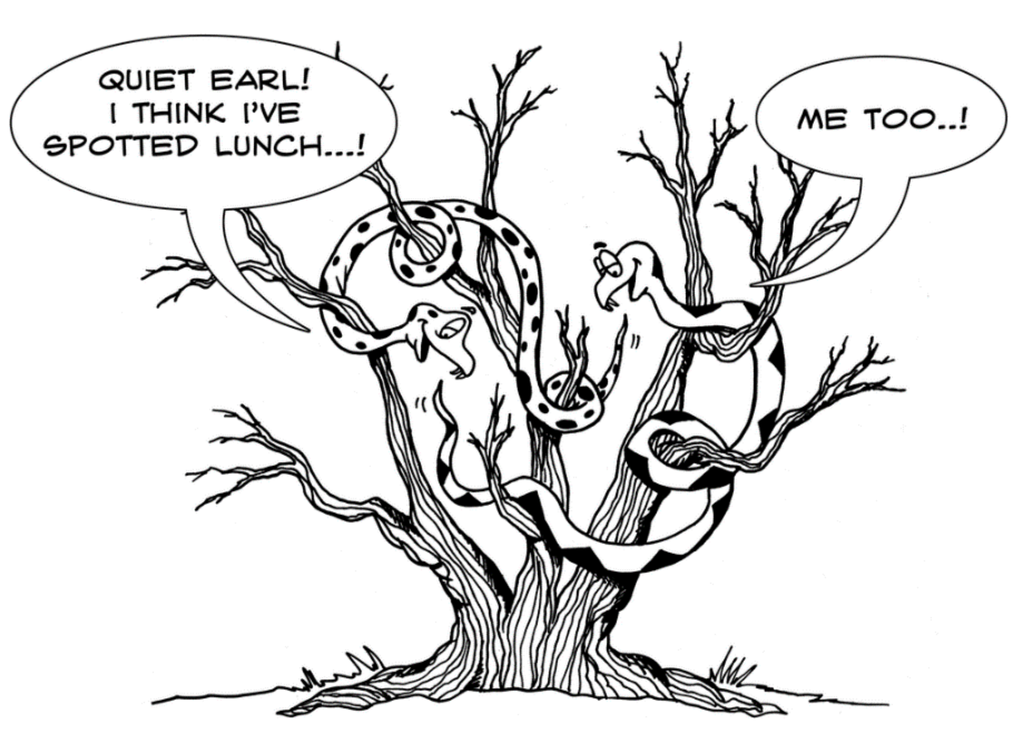
```
if(raspunsCorect == true) {
    //...
}
```

- Boolean variables can be initialized directly

```
boolean raspunsCorect;

if(punctaj == 0) {
    raspunsCorect = false;
}
else {
    raspunsCorect  = true;
}
```

- Don't be negativists!

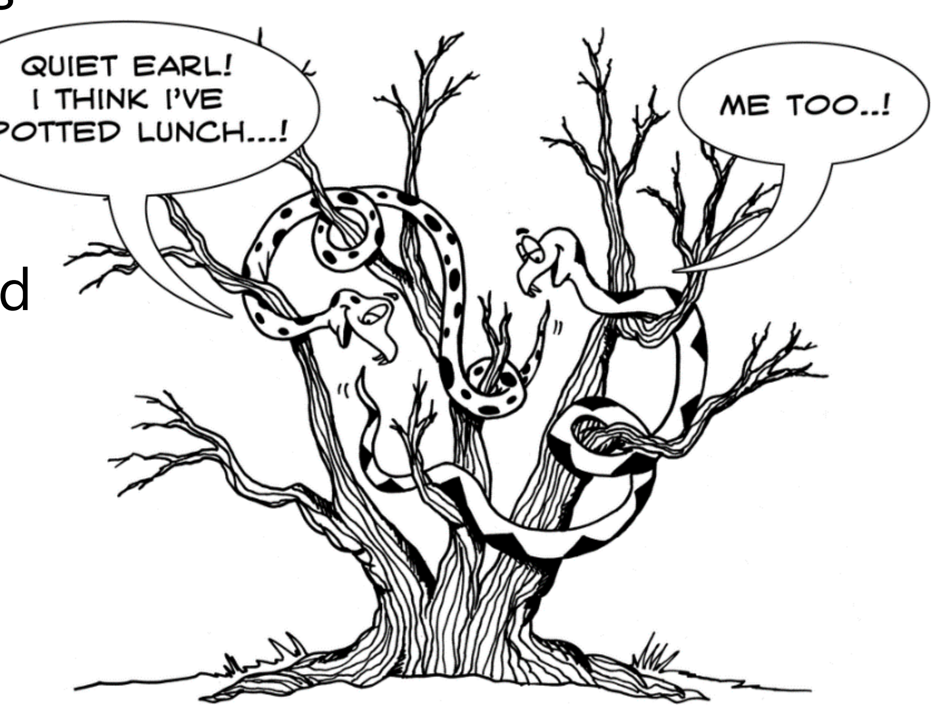


Clean Code Rules for conditional structures

- Use conditional operator for simple *if* structures

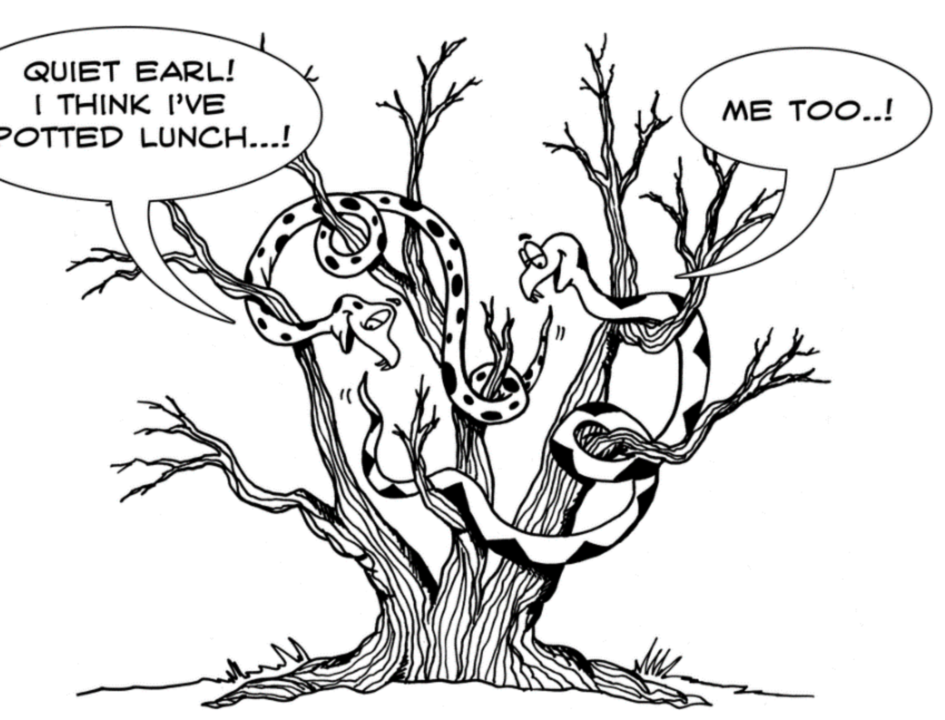
int max = a > b ? a : b;

- Don't compare strings and use *enums* for standard symbols
- Constant values should be named and defined (usually at the beginning of the class)



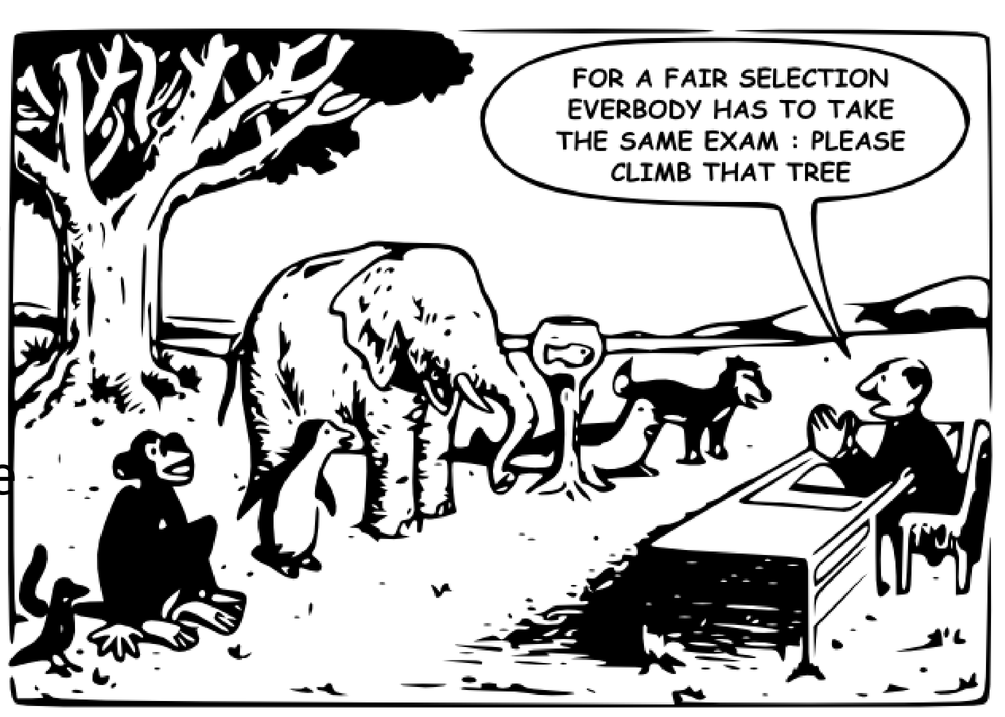
Clean Code rules in Conditional Structures

- When conditions become too large use intermediary variables
- Generally using an *enum* type indicates an improper class design
- Multiple constant values are more properly managed using a DB table or a configuration file



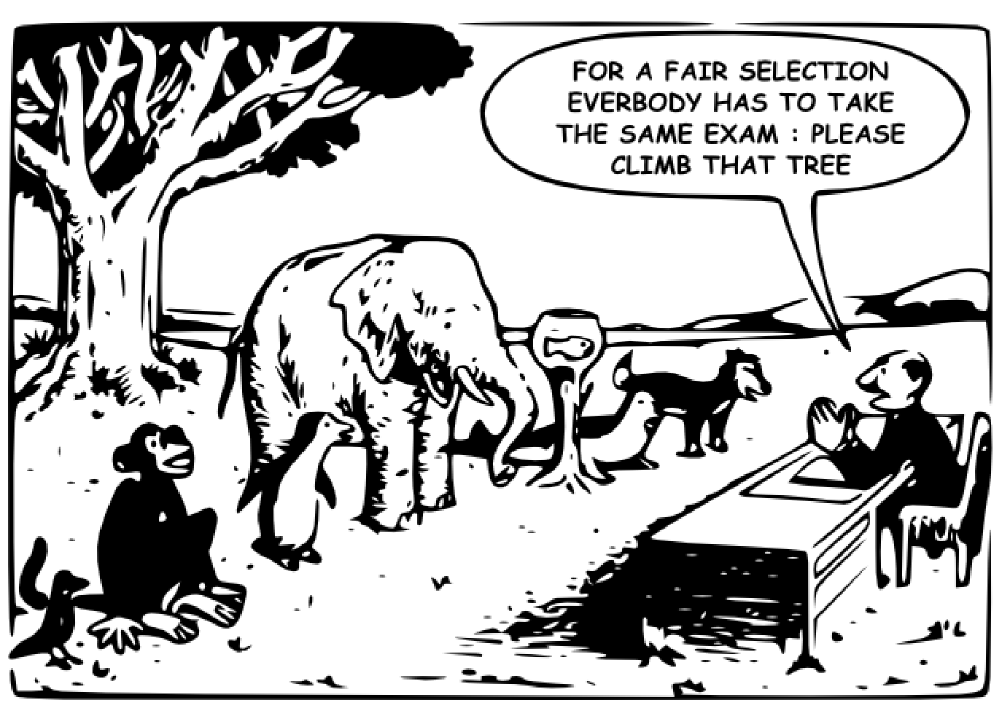
Clean Code rules for Methods

- Any method should have at most 3 imbrication levels (*arrow code*)
- It's better to exit the function as soon as possible (by return or throwing an exception)
- Variables should be declared as closer as possible to the code block that uses them
- Use as much as possible *this* and implement a naming convention for constructor arguments



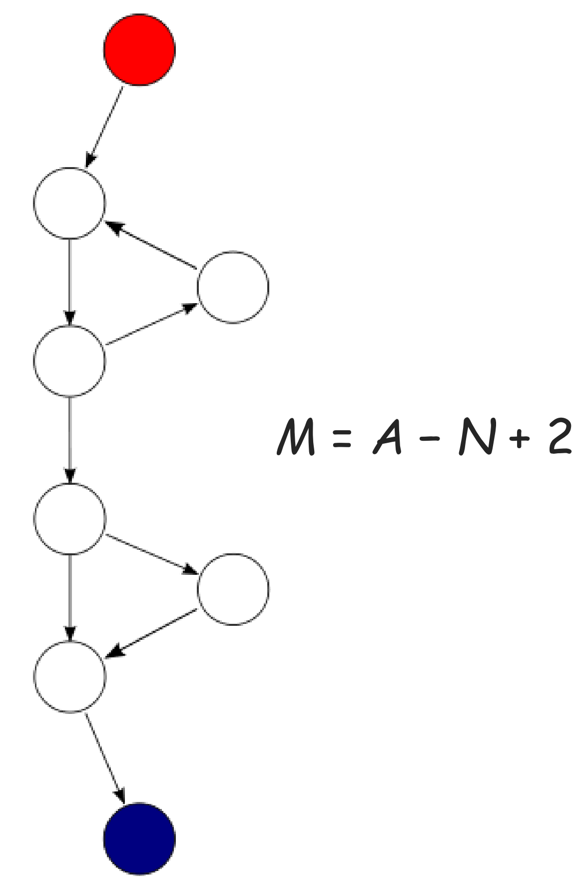
Clean Code rules for Methods

- Avoid methods with more than 2 input arguments
- Avoid long methods (more than 20-30 lines of code) – *one screen rule*
- Complexity should be invers proportional with the number of code lines
- Attention to the exceptions catching order



Clean Code rules for Methods

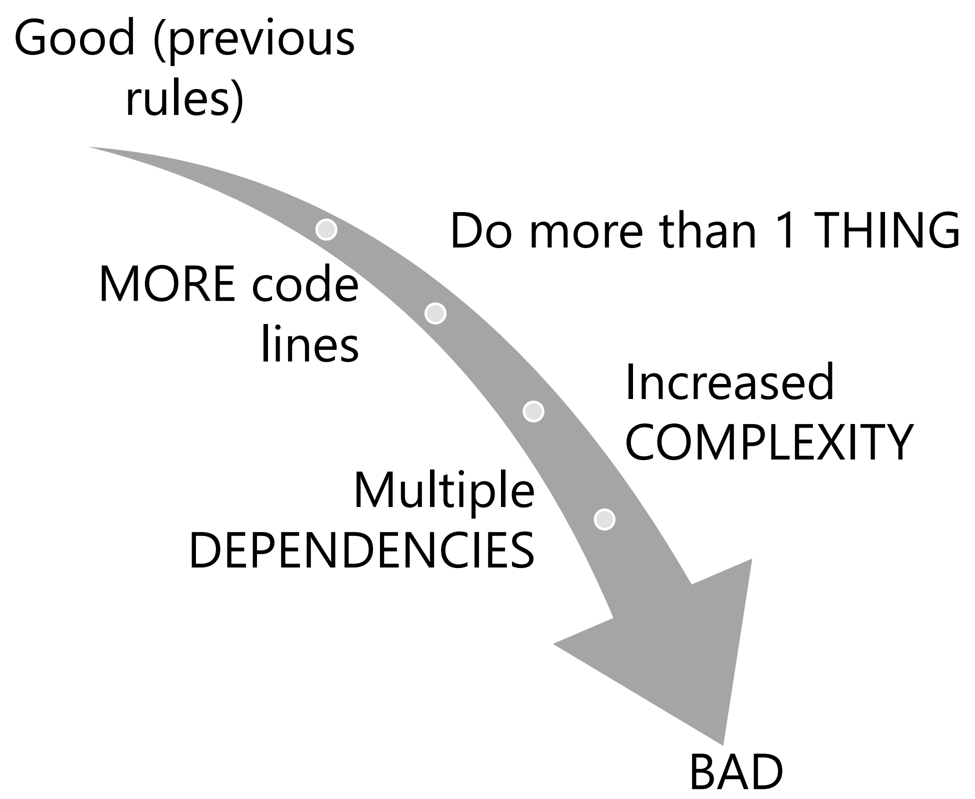
- Check methods cyclomatic complexity
- Simple methods have a complexity = 1
- Conditional structures like *if* si *switch* increase complexity
- Determines the minimum number of tests to increase the coverage

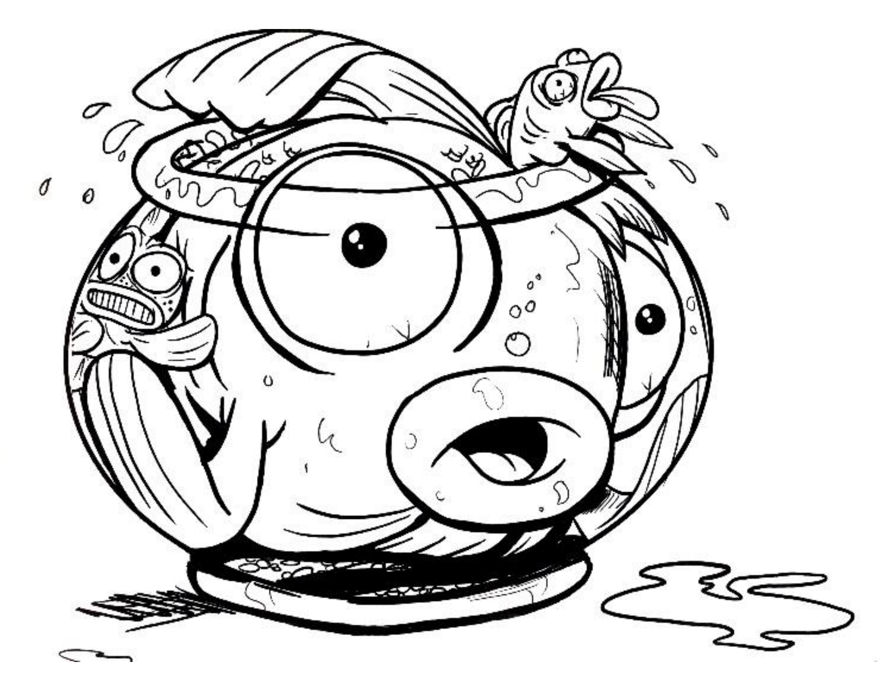


SIMPLE Clean Code rules for methods

- Single responsibility - SRD
- Keep It Simple & Stupid - KISS
- Delegate by references
- Use interfaces

GOOD vs BAD methods

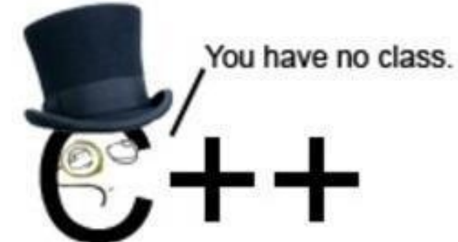




https://ghostlypineapples.wordpress.com

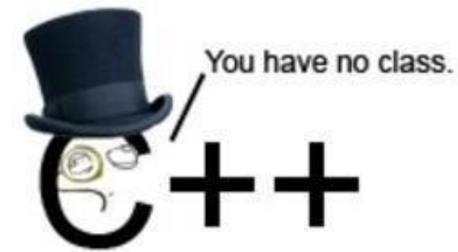
Clean Code rules in Classes

- All class methods should be functional related with the class
- Avoid generated classes and reuse their methods as static ones in your classes
- Avoid primitives types for parameters and use reference types (Wrapper classes in Java) whenever is possible



Clean Code rules in Classes

- Attention to primitives data types and for multi-threading
- Use localization files (locales) for strings displayed on the UI
- Classes that cooperate will be placed in the same module/package
- Use *design patterns* when the problem can be solved with one

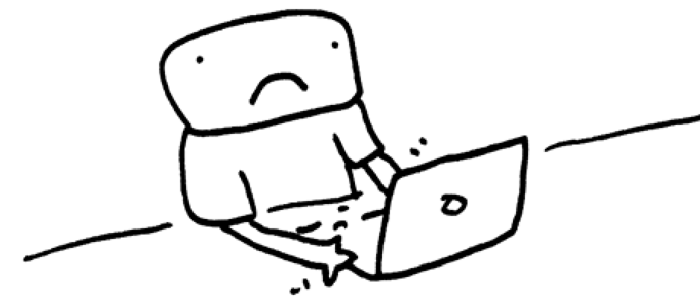


Clean Code rules for Comments

- Many time you don’t need them
- Good code is self-descriptive
- DON’T use comments to apologize

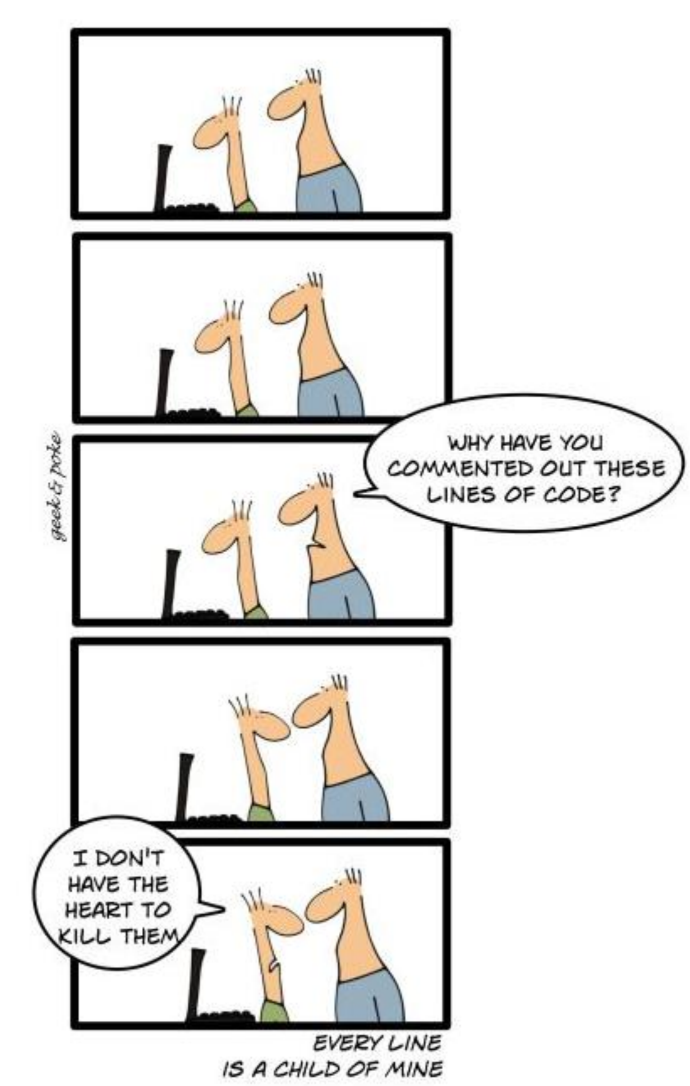
//When I wrote this, only God and I understood what I was doing

//Now, God only knows



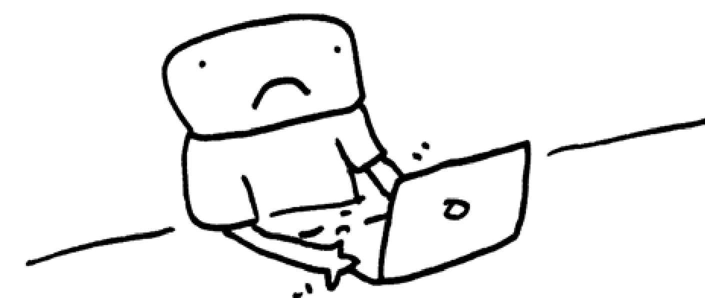
Clean Code rules for Comments

- Don’t comment unused code – becomes *zombie*
- There are versioning solutions to recover changed code
- When you feel you need to add comments to clarify something (a method, some statements, et a), then you should split it in more simple things (methods)



Clean Code rules for Comments

- Avoid introduction type comments
- All the needed details can be put in the commit notes (managed by the versioning system)
- Are recommended for
  - libraries used by other programmers (doc comments) - http://www.oracle.com/technetwork/articles/java/index-137868.html
  - TODO comments

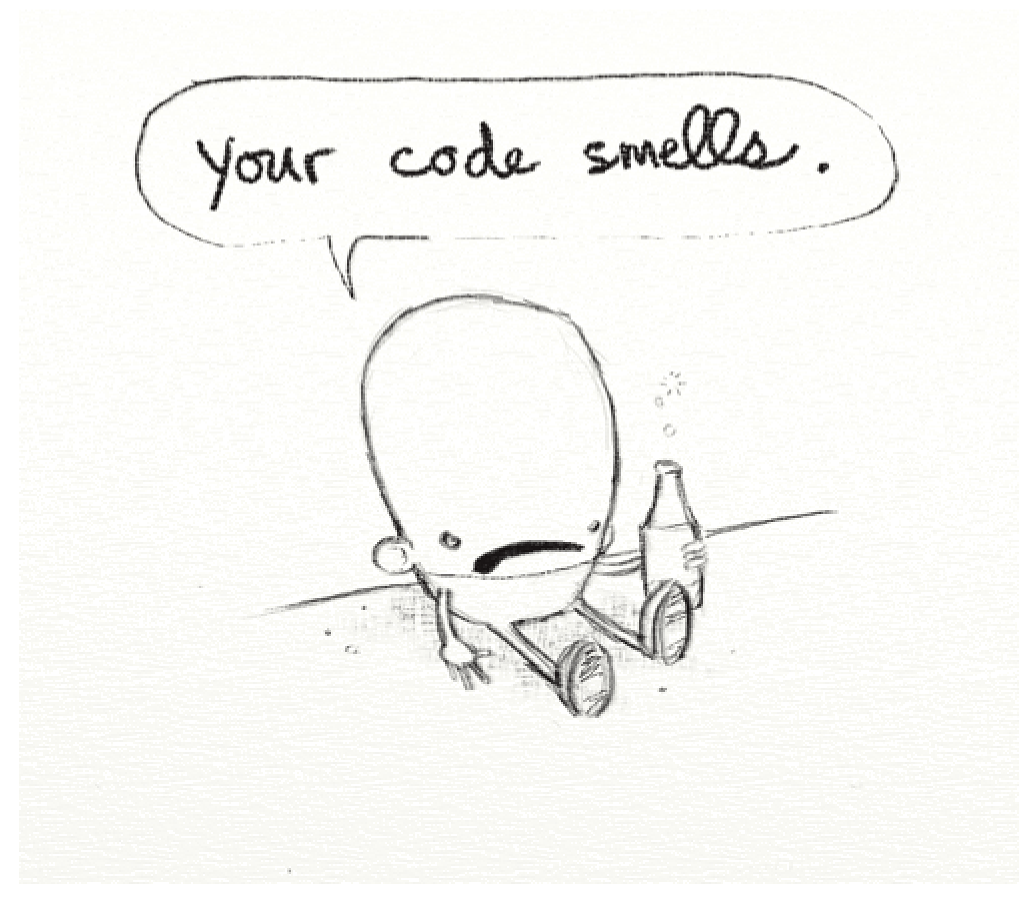


You're lucky I don't hit "Submit" on most of the comments I write.

Bad Code = Code smell

Code smell, (or bad smell) is any symptom in the source code of a program that possibly indicates a deeper problem. [https://en.wikipedia.org/wiki/Code_smell]

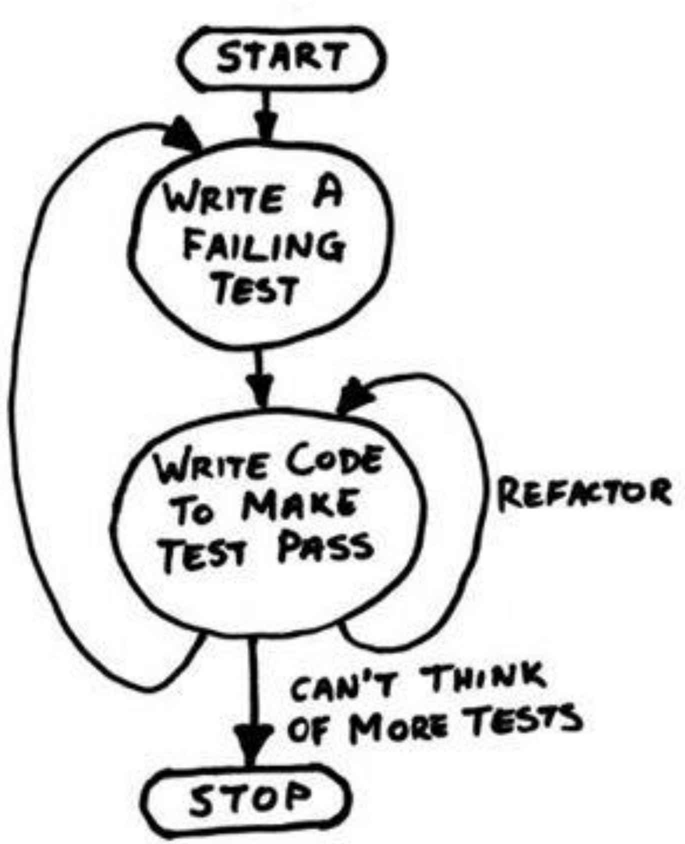
"*A code smell is a surface indication that usually corresponds to a deeper problem in the system*". [Martin Fowler]



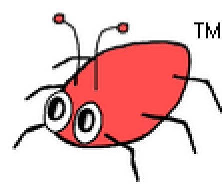
Sursa http://ackandnak.com/comics/your-code-smells.html

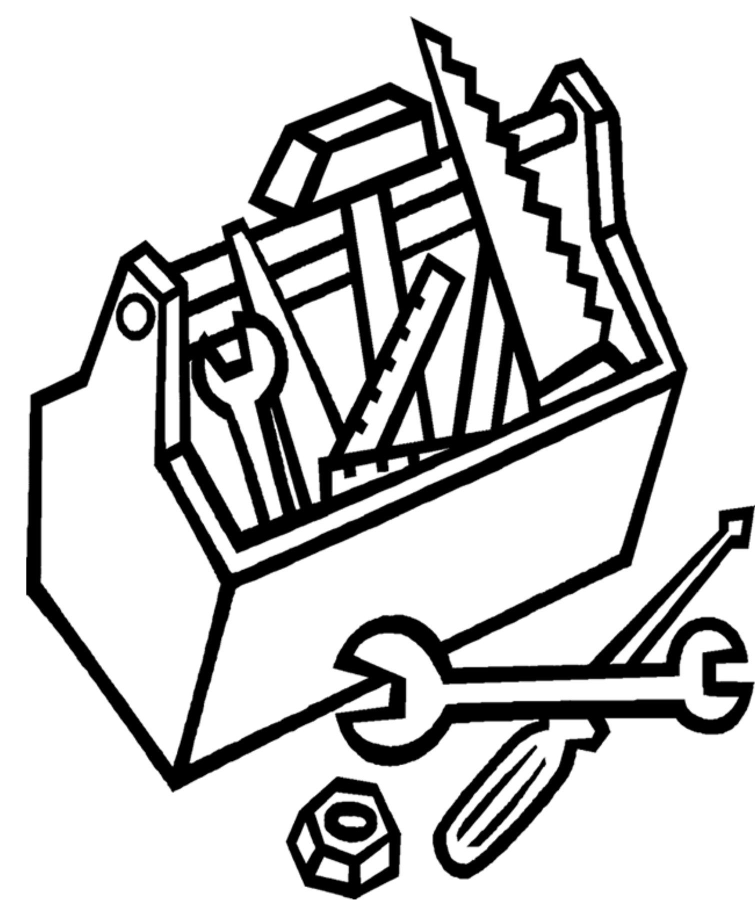
Short Dictionary

- **Test Driven Development (TDD)** – code development based on use cases/test cases
- **Refactoring** – rewriting the code to adapt it to new specifications or to correct it
- **Automatic Testing (Unit Testing)** – Code automatic testing base don use cases. Useful for refactoring phases because helps you check if all the functionalities have been preserved. (regression testing)
- **Code review** – procedure used in AGILE (XP, SCRUM) that requires that any source code should be checked (reviewed) by a different programmer
- **Pair programming** – programming technique specific to AGILE based on which programmers work in teams/pairs to implement complex tasks; this aporach promotes learning and avoid code review



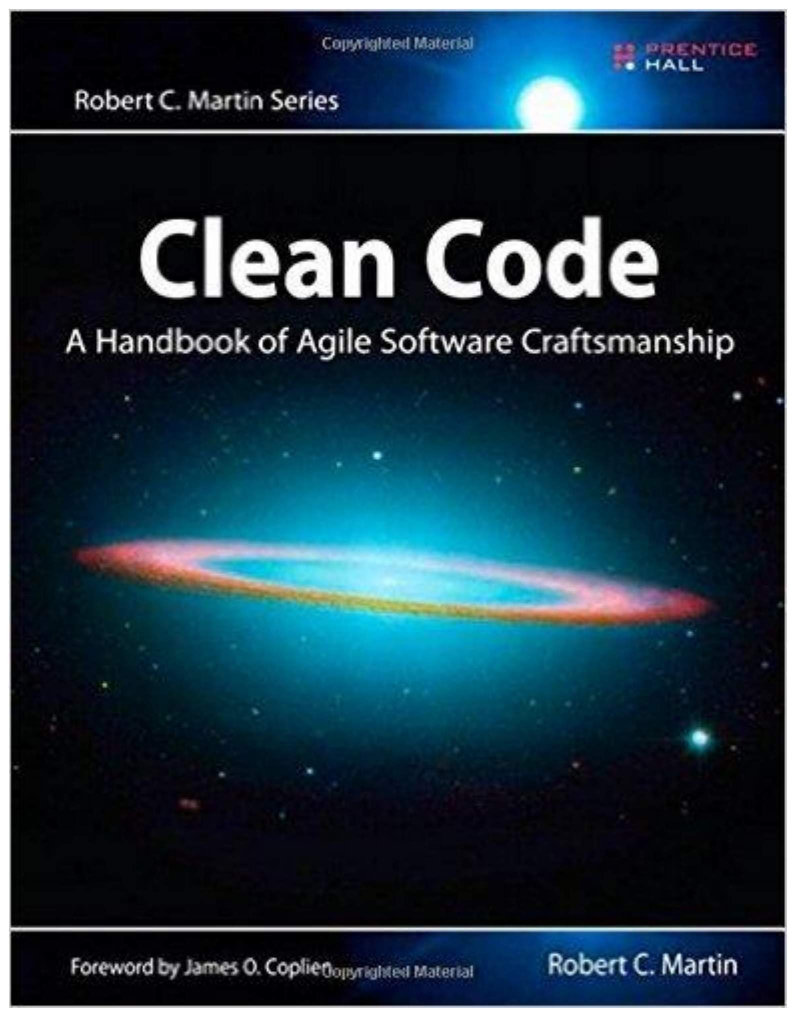
Instrumente





To read

Robert C. Martin (Uncle Bob) – *Clean Code: A Handbook of Agile Software Craftsmanship*



Bonus

- The *Broken Window Principle*: clădirile cu ferestre sparte sunt mult mai vulnerabile la vandalism, care va duce la mai multe ferestre sparte;
- The *Boy Scout Rule*: lasați codul puțin mai curat decât l-ați găsit.
- Supplementary resources :

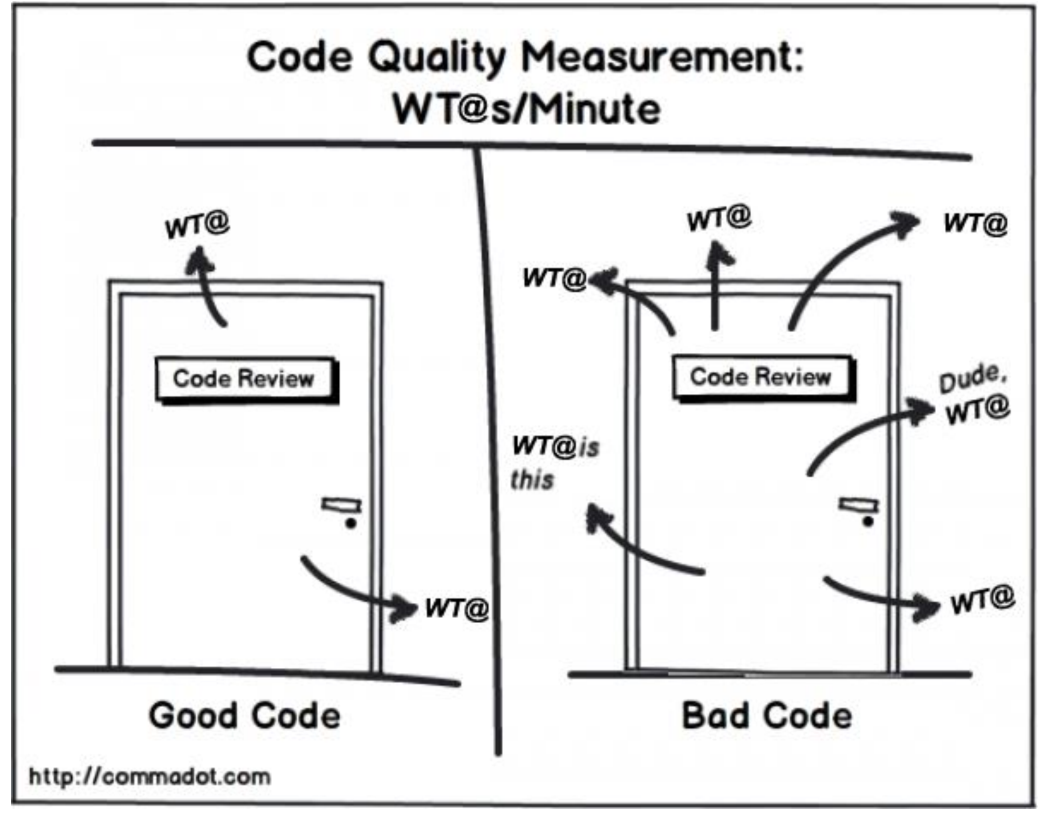
  1. Robert C. Martin (Uncle Bob) – Clean Code: A Handbook of Agile Software Craftsmanship
  2. Clean Code: Writing Code for Humans – Pluralsight series
  3. Design Principles and Design Patterns", Robert C. Martin
  4. Refactoring. Improving the Design of Existing Code, by Martin Fowler (with Kent Beck, John Brant, William Opdyke, and Don Roberts)

SUPER BONUS

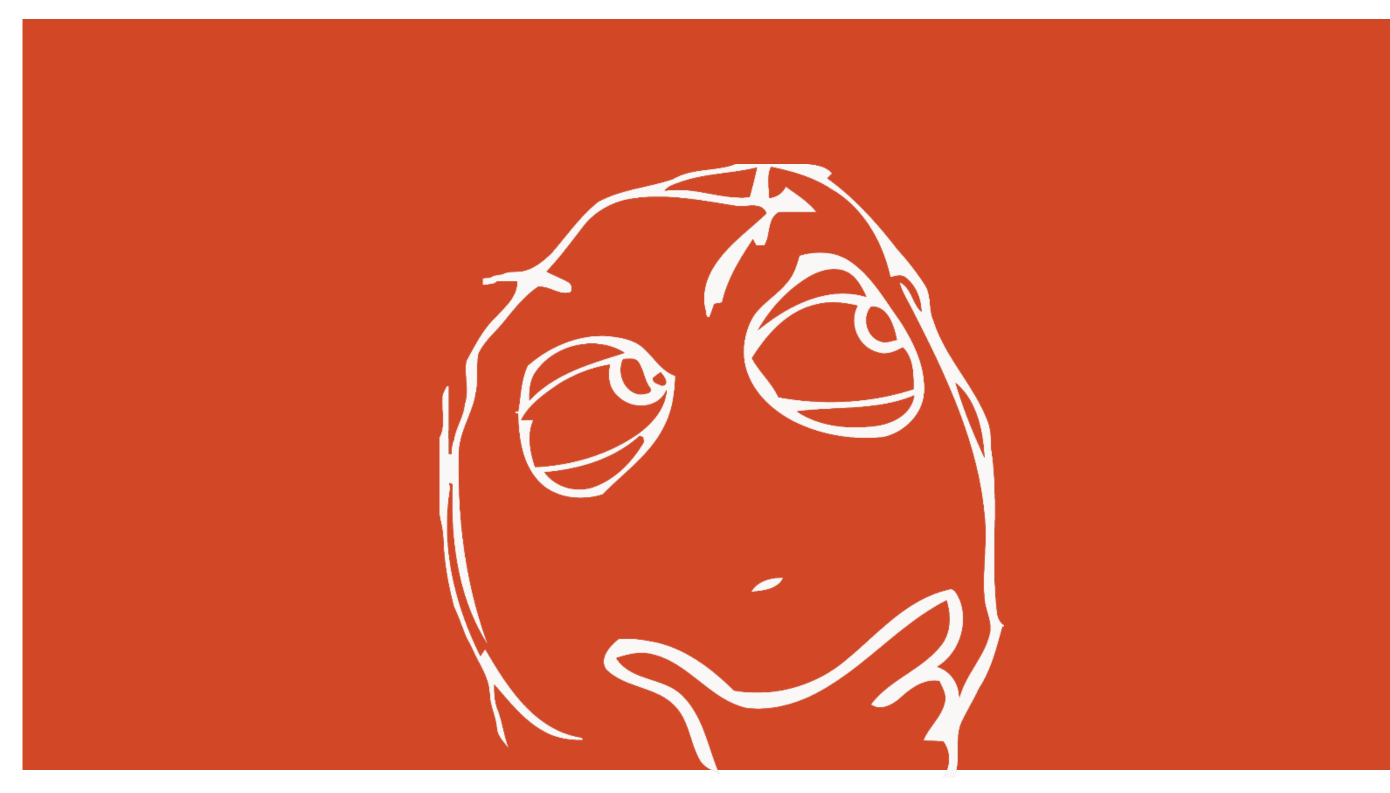
One more thing

- *"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."*

  Martin Golding



Questions?



Thank you!