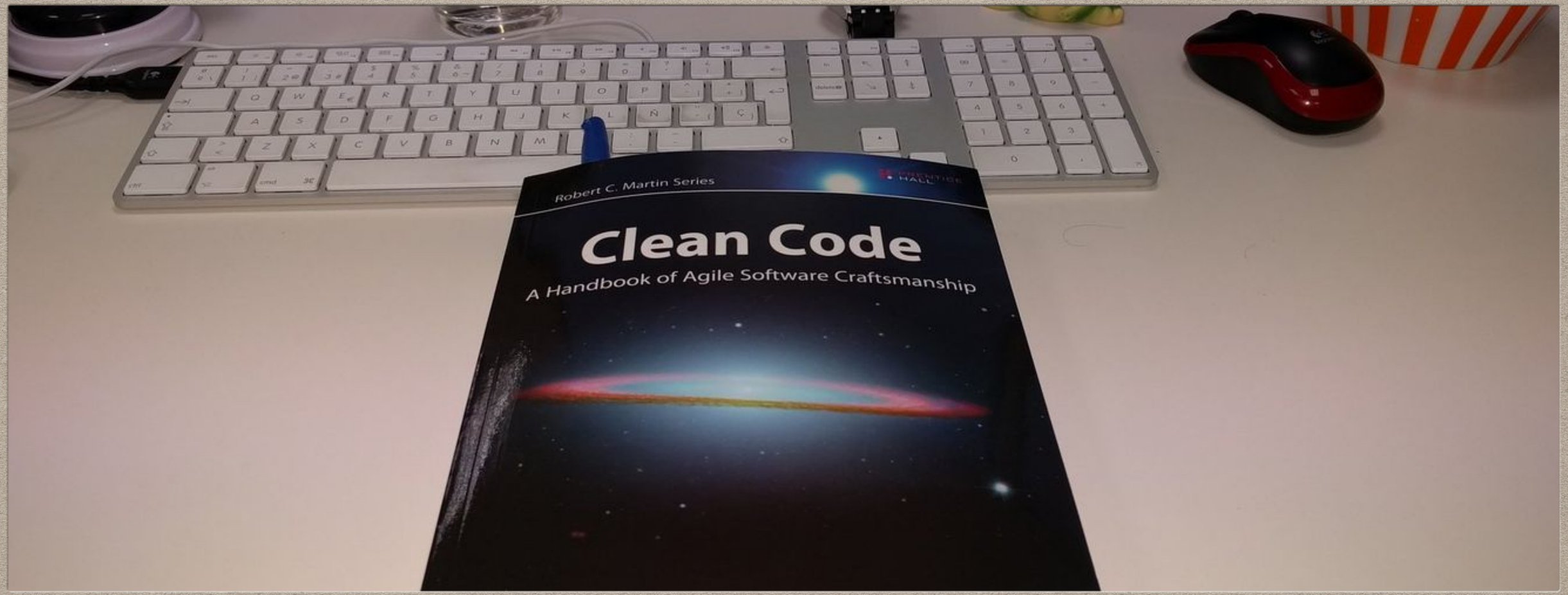


CLEAN CODE

A BRIEF INTRODUCTION TO MAKE YOUR CODE GREAT AGAIN

WHAT IS CLEAN CODE?

- The handbook of Agile Software Craftsmanship
  - Why a handbook?
  - Why Agile?
  - Why Craftsmanship?

WHAT IT CONTAINS?

- PRINCIPLES (SOLID, DRY, FIRST)
- RULES (BOY SCOUT, RUN-ALL-TESTS, REFACTORING)
- CODE EXAMPLES
- FUNNY DRAWS!
- CHAPTERS! (UP TO 17)


EDUCATION FOR ALL
EDUCATION FOR THE MASSES
NOT JUST FOR THE RULING CLASSES
coffee with
principles

SOLID

- SINGLE RESPONSIBILITY
- OPEN CLOSE
- LISKOV SUBSTITUTION
- INTERFACE SEGREGATION
- DEPENDENCY INVERSION

DRY

- Nothing to do to avoid get wet
- It is just “Don’t Repeat Yourself”
  - Don’t (let) Repeat themselves
  - Don’t Repeat (again) Stack Overflow
  - No more ctrl+v/c, shit

DRY

- Design patterns:
  - Method template
  - Decorator
  - Command Dispatcher
  - Strategy

FIRST

- Because I don’t like to repeat myself, this can be checked here!


RULES

THE BOY SCOUT RULE

NAMING!!!

USE INTENTION-REVEALING NAMES

- Choosing good names takes time but saves more than it takes
- The name of a variable, function or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used.
- If the name needs comments, the name then does not reveal intention

USE INTENTION-REVEALING NAMES

- final int d; //elapsed time in days :(
- final int elapsedDays; :)
- final List<User> users = getDisabledUsers(); // :(
- final List<User> disabledUsers = getDisabledUsers(); // :)

AVOID DISINFORMATION

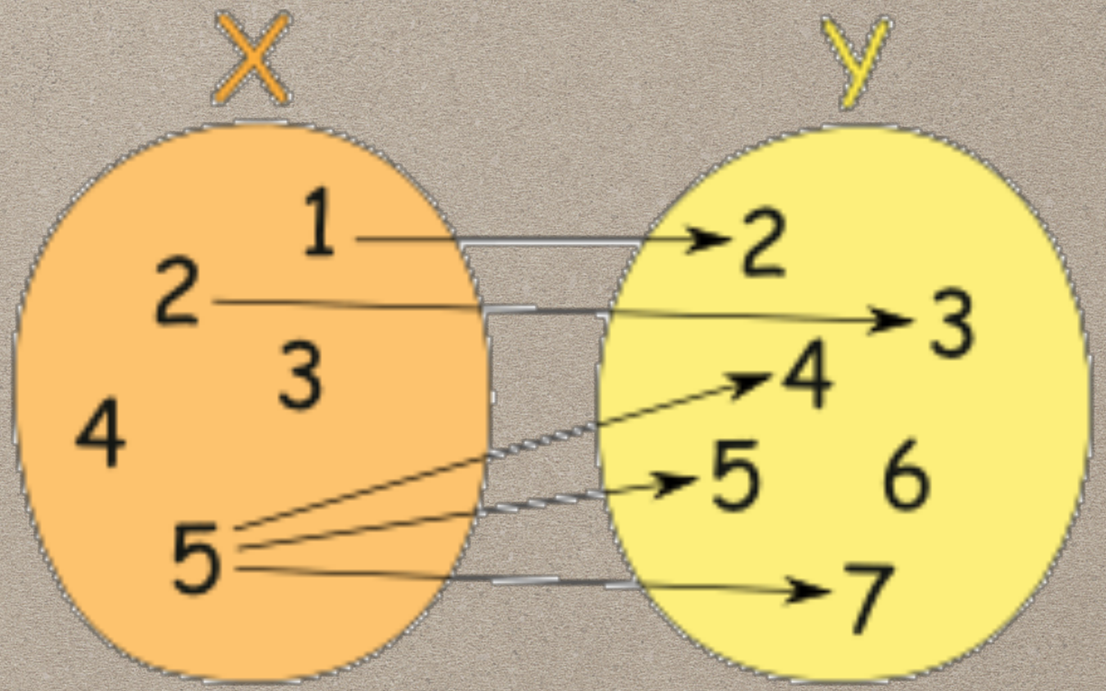
- Avoid abbreviations like “idx” (index) “hp” (hypothenuse) “tmp” (temporal)
- Avoid keywords as suffix “accountList” (just use accounts, List has a meaning in the programming world, and what if accounts is a Set?)

MAKE MEANINGFUL DISTINCTIONS

- Distinctions are needed to don’t get crazy
  - String composeFullName(String arg1, String arg2)// :(
  - String composeFullName(String name, String lastName)// :)
- Avoid Noise Words if it is possible (Info, Data, Ammount…)
  - moneyAmmount;//:(
  - money;:)
  - Product productInfo; Product productData; //??

USE PRONOUNCEABLE AND SEARCHABLE NAMES

FUNCTIONS



FUNCTIONS!!!

- Small, to do only one thing and to do it perfectly
  - void validateAndSave(Person aPerson) ; //:(
  - void validatePerson(Person person); // :|
  - void savePerson(Person person); // :|
  - void validate(Person newPerson); // :)
  - void save(Person newPerson); // :)

FUNCTION ARGUMENTS

- Ideally, zero arguments. But up to three are an “accepted” practice
- Avoid updating input arguments!
  - includeSetupPageInto(StringBuffer pageText) ; // :(
  - StringBuffer appendSetupPage(final StringBuffer pageText); // :)

FLAG ARGUMENTS

BECAUSE THEY ARE UGLY!!!!

AND NOW SERIOUSLY, FLAG ARGUMENTS

- Do this if the arg is true or do that if it is false is a terrible practice
- Loudly proclaiming the function is doing more things than expected
- render(true);// ???? What the hell is rendering
- renderHeader(); renderBody();
- Not only booleans are used as flags be aware of:
  - If arg != null do this if not do that :)
  - Optional params as argument are also aaaaaaawfuuuuuuulll!

NO SIDE EFFECTS

- Your function promise to do one thing but it does more “hidden stuff”, you are a liar developer
  - Unexpected changes in the input parameters
  - Strange temporal dependencies

NO SIDE EFFECTS

```
public class UserValidator {
private Cryptographer cryptographer;
public boolean checkPassword(String userName, String password) {
User user = UserGateway.findByName(userName);
if (user != User.NULL) {
   String codedPhrase = user.getPhraseEncodedByPassword();
   String phrase = cryptographer.decrypt(codedPhrase, password);
   if ("Valid Password".equals(phrase)) {
      Session.initialize();
   return true;
  }
}
return false; }
}
```

EXCEPTIONS RATHER ERRORS

- Because it promotes for “if” “if” “if”, and it force you to have a nested leading architecture

```
if(deleteAccount(account)) {
    if(!deleteBankDetails(account) {
    log.error(“bank details could not be deleted”);
    }
} else {
log.error(“account not deleted”);
}
```

EXCEPTIONS RATHER ERRORS

- Because it promotes for “if” “if” “if”, and it force you to have a nested leading architecture

```
if(deleteAccount(account)) {
    if(!deleteBankDetails(account) {
    log.error(“bank details could not be deleted”);
    }
} else {
log.error(“account not deleted”);
}
```

EXCEPTIONS RATHER ERRORS

```
Try {

deleteAccount(account);

deleteBankDetails(account.getBankDetails());

} catch(Exception e) {

logger.error(e);

}
```

COMMENTS!!!

GOOD COMMENTS

- Legal comments (gnu license, etc.)
- Informative comments

// format matched kk:mm:ss EEE, MMM dd, yyyy

Pattern timeMatcher = Patter.compile("\\d*:\\d*:\\d*\\w*,\\w* \\d*, \
\d*");

GOOD COMMENTS

- Explain intentions and clarifications:

//I’m not interested in mapping the relationship with UserEntity due to performance issues

final Long userId;

- Warning consequences (for example, edge cases to check if anything is changed)
- TODO’s
- Doc (javadoc or swagger) in public API’s

GOOD COMMENTS

- Explain intentions and clarifications:

//I’m not interested in mapping the relationship with User
performance issues

final Long userId;

- Warning consequences (for example, edge cases to ch

- TODO’s

- Doc (javadoc or swagger) in public API’s

BAD COMMENTS

- Doc (javadoc or swagger) in public API's !!!!!!!!

BAD COMMENTS

- Doc (javadoc or swagger) in public API's
  - They can be misleading, dishonest and bad comments as any other comments

BAD COMMENTS

- Usually they are excuses for poor code or justifications for insufficient decisions, amounting to little more than the programmer talking to himself.

BAD COMMENTS

- Mumbling (Murmullos en Español ;))

```
try {
String propertiesPath = propertiesLocation + "/" +
PROPERTIES_FILE; FileInputStream propertiesStream = new
FileInputStream(propertiesPath);
loadedProperties.load(propertiesStream);
}
catch(IOException e) {
// No properties files means all defaults are loaded
}
```

BAD COMMENTS

- Redundant
  - This happens when you write a comment that it takes more time than reading the code itself

```
//this function save of update a given person returning the updated entity

Person saveOrUpdate(final Person changedPerson) {

...

}
```

BAD COMMENTS

- More redundant examples (they typically are requested by a Manager who is playing with a Sonar)

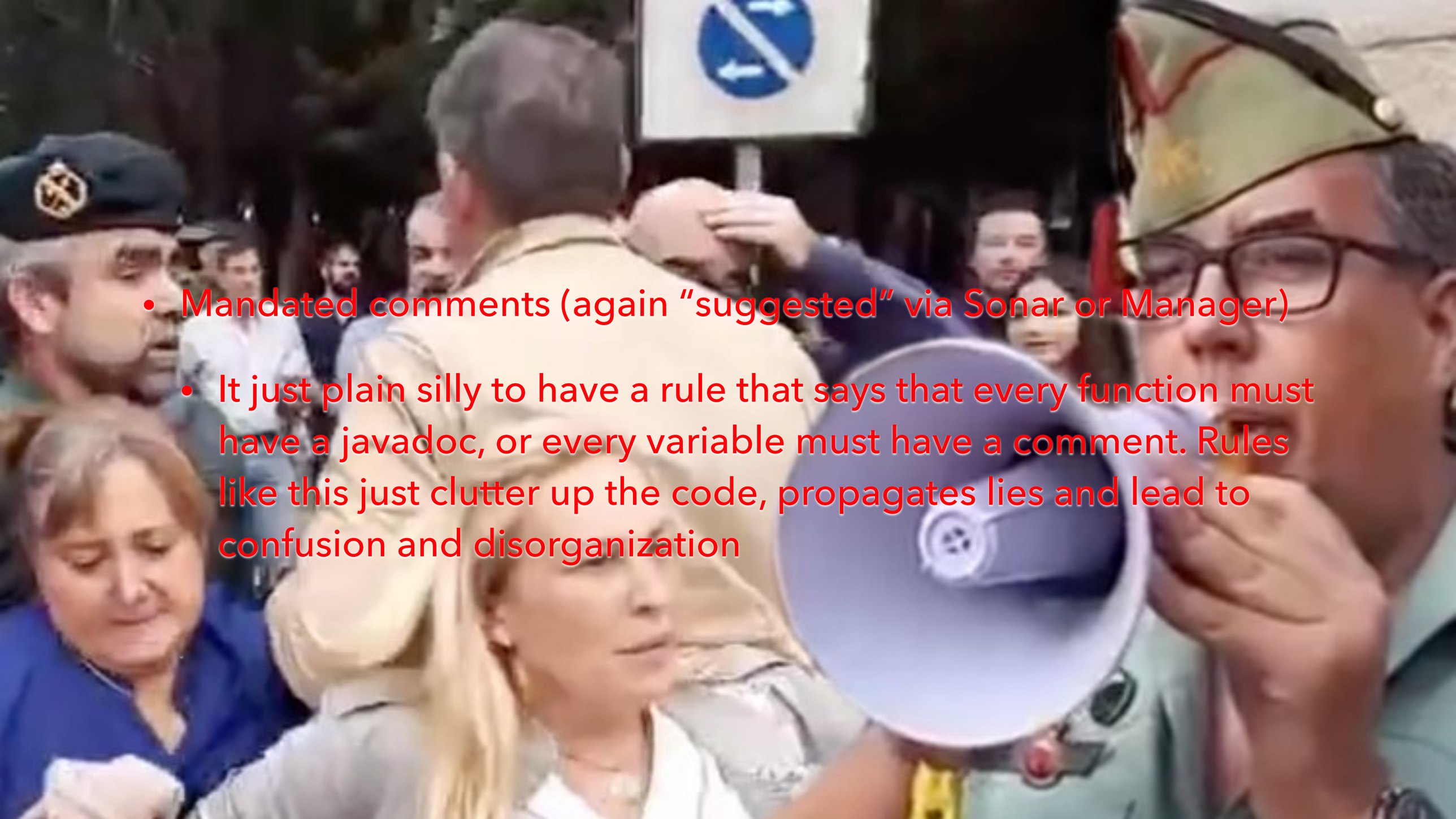
```
//this class represents a person

Class Person {

//this is the id for the person in the System

Final Long id;

//this is their bank details

Final Bank bank;

//his email

Final String email;

}
```

BAD COMMENTS

- Misleading comments
- Broken promises, not talking about side effects.. remember the case of “validatePassword” :S

BAD COMMENTS

- Mandated comments (again “suggested” via Sonar or Manager)
  - It just plain silly to have a rule that says that every function must have a javadoc, or every variable must have a comment. Rules like this just clutter up the code, propagates lies and lead to confusion and disorganization
  - Often they are redundant, noisy and scary comments



- Mandated comments (again “suggested” via Sonar or Manager)
- It just plain silly to have a rule that says that every function must have a javadoc, or every variable must have a comment. Rules like this just clutter up the code, propagates lies and lead to confusion and disorganization

BAD COMMENTS

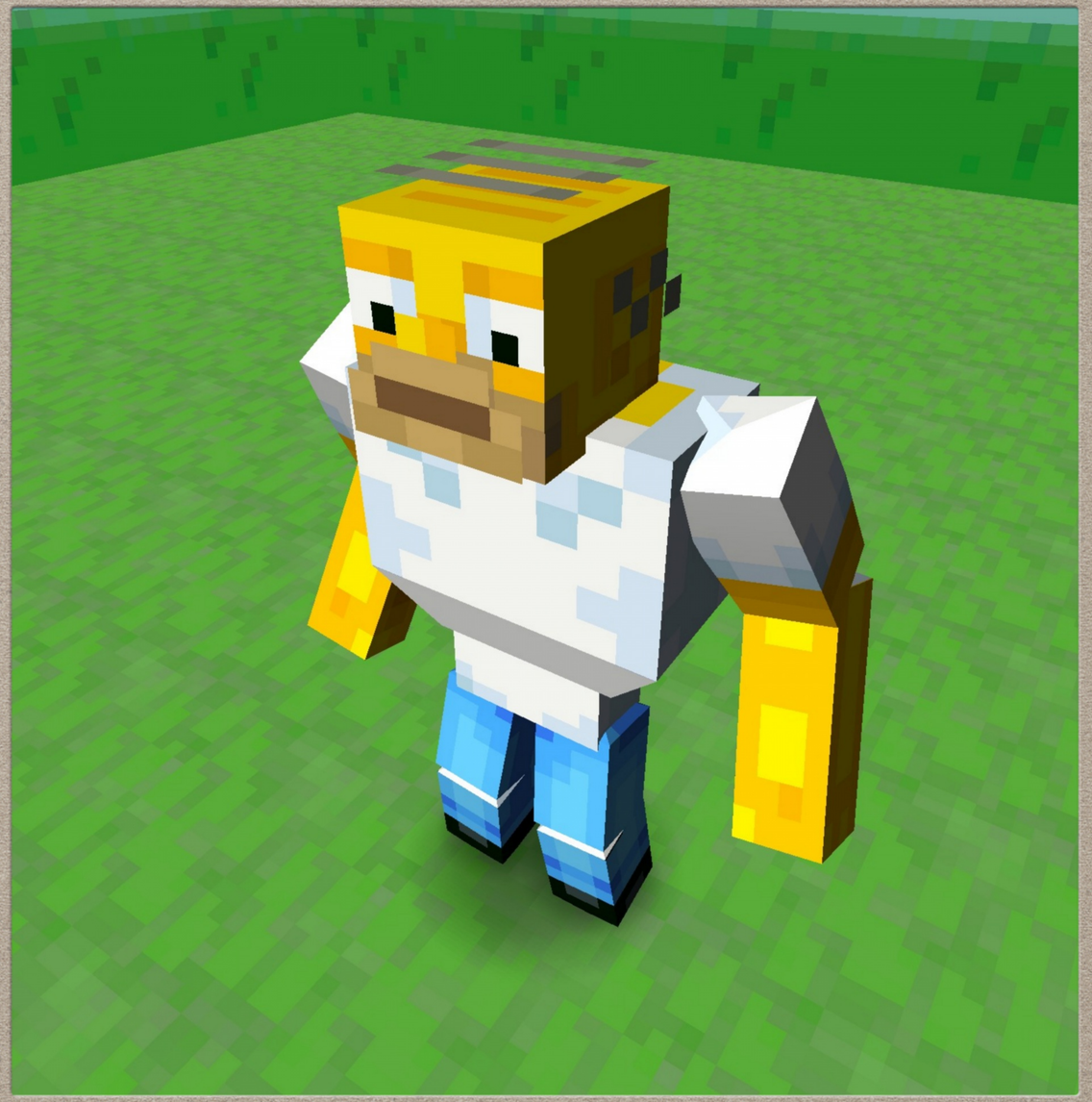
```
/** *
  *  @param title The title of the CD

  *  @param author The author of the CD

  *  @param tracks The number of tracks on the CD

  *  @param durationInMinutes The duration of the CD in minutes */
public void addCD(String title, String author,
int tracks, int durationInMinutes) {
CD cd = new CD();
cd.title = title;
cd.author = author;
cd.tracks = tracks;
cd.duration = duration;
```

BAD COMMENTS

- More bad comments
  - Attributions (//added by Jesus)
  - Commented out code (please don’t!)
  - Html comments or with tags..
  - To much information…

CLASSES

BECAUSE "IS DA MASTA" PIECE FOR OUR PUZZLES, BITCH



NOT EVERYTHING IS PUBLIC OR PRIVATE!

- It’s true we should aim for private visibility
- Take your time to think in a good package structure
  - Rather than by layers I’d suggest to organize your classes by use cases (DDD)
- Think in when the “encapsulation” can be broken (package private and protected visibility are useful)

CLASSES SHOULD BE SMALL!

- Do you know that create classes is free?
- If we're trying to make small functions so why then we should create big classes?
- Having a big class lead to violate the Single Responsibility principle
- A big class ofter lead to have so many external dependencies

THE BIG CLASS!

```
public EmployeeService implements IEmployee {

public List<Employee> getAll();

public List<Employee> getByCriteria(Criteria search);

public Employee getById(Long id);

public void update(Employee existingEmployee)

public void save(Employee newEmployee)

public void delete(Long existingId)

public boolean exists(Long id)

public void validate(Employee unvalidatedEmployee)

Public List<Payrolls> getPayrolls(Long id)

…

}
```

CLASSES ARE FREE!

SearchEmployee implements Searchable<Employee, Criteria>

WriteEmployeeService implements Writable<Employee, Long>

ValidateEmployee implements Validator<Employee>

SearchPayroll implements Searchable<Payroll, Long>

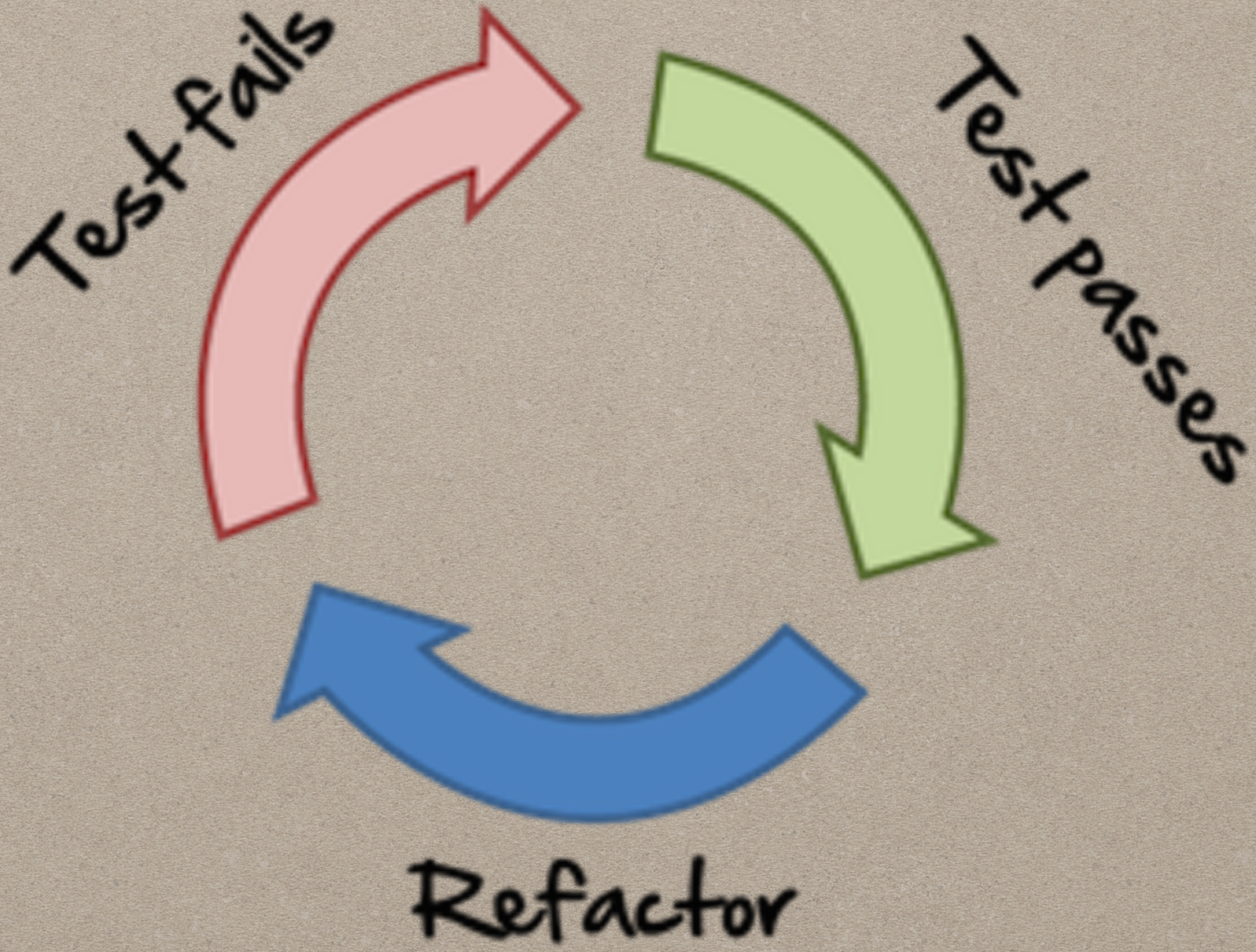
...

SINGLE RESPONSIBILITY

- One reason to change! (one responsibility)
- Identify responsibilities helps us to organize our code and to create better abstractions
- There is nothing wrong in having your system decoupled in too many components. Even it’s a best practice to follow

LOW COHESION

- Small classes aims for a low cohesion.
- Classes should have as less dependencies as we can
- But this is not only about dependencies, is also about, variables and arguments
- A class that needs to handle with so many different types is a bullshit and it would be affected by every single refactor, even if it's a stupid one!!!


Test fails
Test passes
Refactor

TDD

- A whole chapter is only for TDD
- Remember: Test First + Refactoring (eXtremme Programming)
- Will encourage people to collaborate one each other (Pair)
- Will encourage to write small classes
- Will encourage to write meaningful functions
- (!) Please, follow the Clean Code Principles also in tests because they care

QUESTIONS?


WATA
FACTORY
Clean Code
A Handbook of Agile Software Craftsmanship

QUESTIONS?

THANKS FOR THE GOOD MOMENTS BUT...