SOFTWARE CRAFTSMANSHIP

software
development
academy

Author: Rafał Roppel

SOFTWARE CRAFTSMANSHIP

Author: Rafał Roppel

WHAT IS CRAFTSMANSHIP?

- skill in a particular craft
- the quality of design and work shown in something made by hand; artistry

Author: Rafał Roppel

SOFTWARE CRAFTSMANSHIP

Is an approach to software development that emphasizes the coding skills of the software developers themselves.

It is a response by software developers to the perceived ills of the mainstream software industry, including the prioritization of financial concerns over developer accountability.

Author: Rafał Roppel

ORIGINS

The primary force that's energized the software craftsmanship community is a reaction to the change in the agile movement.

In the early days of the agile virus, the dominant strain was Extreme Programming, which has a lot to say about technical practices.

Author: Rafał Roppel

ORIGINS

Now the dominant agile strains are Scrum and Lean, which don't care very much about programming - and thus those people who primarily self-identify as programmers feel a large part of their life is no longer important in the agile world.

Author: Rafał Roppel

ORIGINS

The software craftsmanship world, therefore, is place where programming can become front-and-central again.

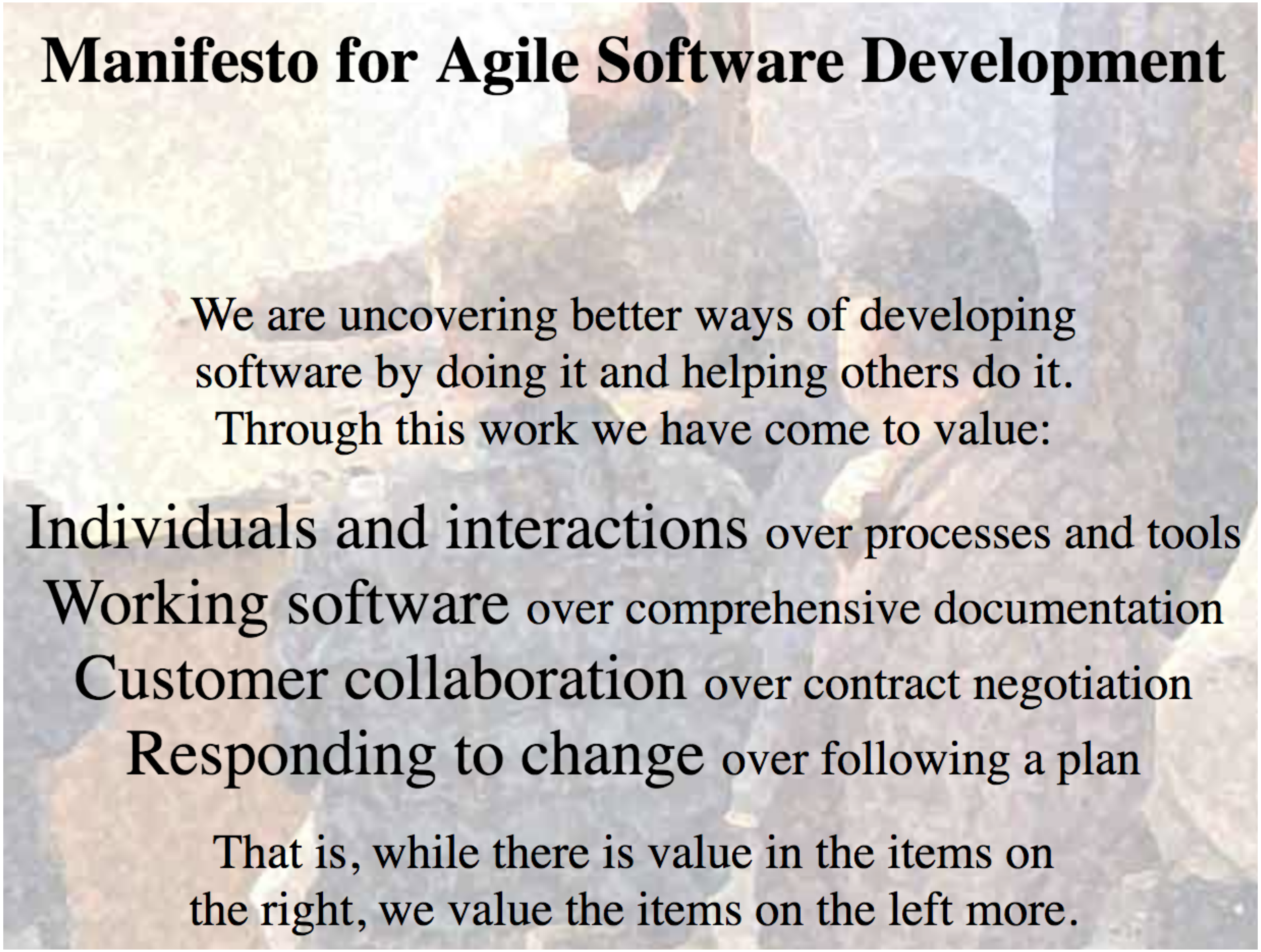
Martin Fowler CraftmanshipAndTheCrevasse

SOFTWARE CRAFTSMANSHIP

Historically, programmers have been encouraged to see themselves as practitioners of the well-defined statistical analysis and mathematical rigor of a scientific approach with computational theory.

This has changed to an engineering approach with connotations of precision, predictability, measurement, risk mitigation, and professionalism.

Practice of engineering led to calls for licensing, certification and codified bodies of knowledge as mechanisms for spreading engineering knowledge and maturing the field.



Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

It is worth to read Principles behind the Agile Manifesto as well.

Author: Rafał Roppel

SOFTWARE CRAFTSMANSHIP

The *Agile Manifesto*, with its emphasis on "*individuals and interactions over processes and tools*" questioned some of these assumptions.

The *Software Craftsmanship Manifesto* extends and challenges further the assumptions of the *Agile Manifesto*, drawing a metaphor between modern software development and the apprenticeship model of medieval Europe.

SOFTWARE CRAFTSMANSHIP - MOTIVATION

We are tired of writing crap

We will not make messes in order to meet a schedule

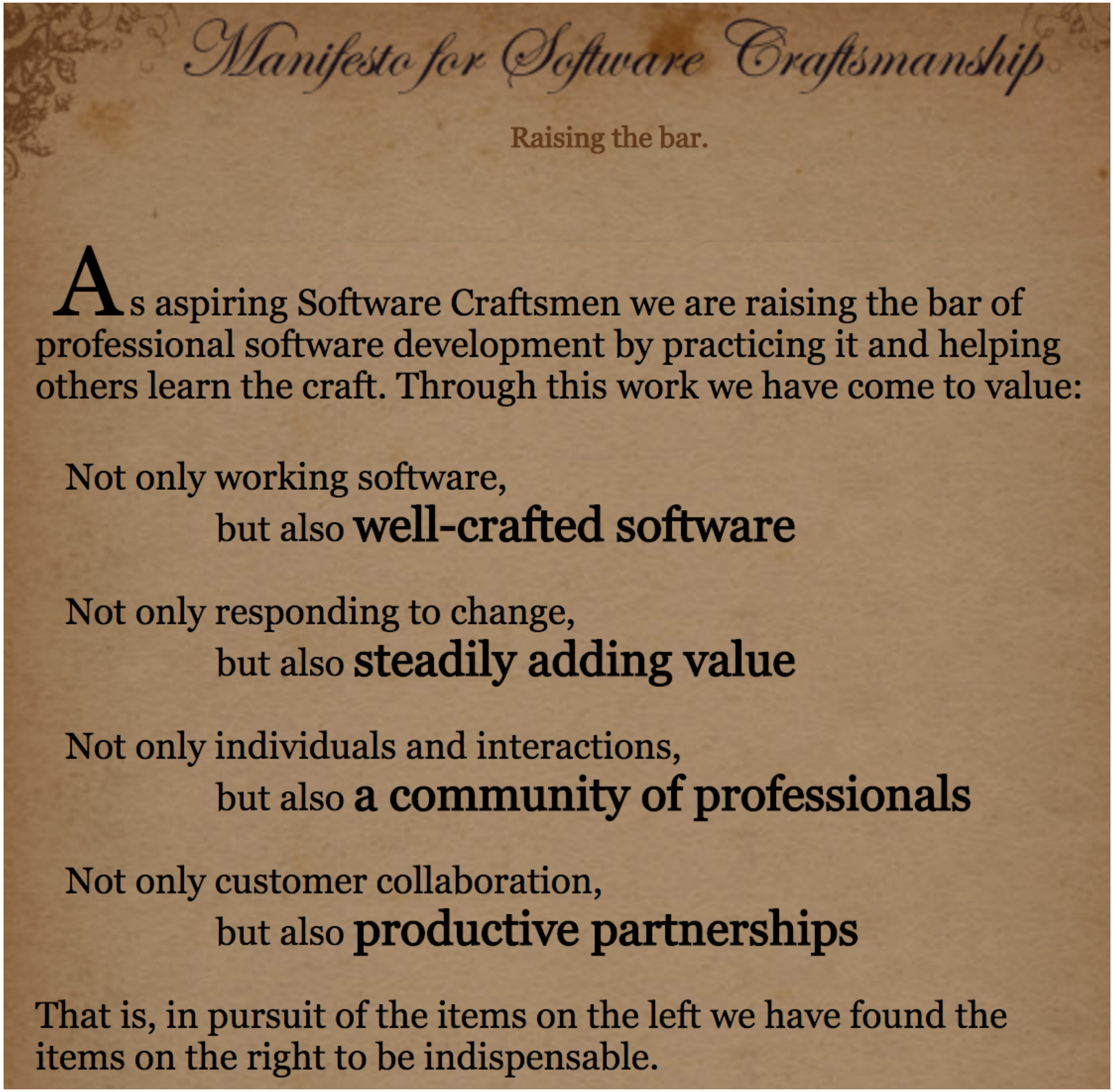
We will not accept the stupid old lie about cleaning things up later

We will not believe the claim that quick means dirty

We will not accept the option to do it wrong

We will not allow anyone to force us to behave unprofessionally

Author: Rafał Roppel



Manifesto for Software Craftsmanship

Raising the bar.

As aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping others learn the craft. Through this work we have come to value:

Not only working software,
but also **well-crafted software**

Not only responding to change,
but also **steadily adding value**

Not only individuals and interactions,
but also **a community of professionals**

Not only customer collaboration,
but also **productive partnerships**

That is, in pursuit of the items on the left we have found the items on the right to be indispensable.

Author: Rafał Roppel

JAVA CODE CONVENTIONS

September 12, 1997

Notice, that this is a quite old document.
In real-world-projects you can meet with specific Code of Conducts.

WHY?

Code conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes to maintenance
- hardly any software is maintained for its whole life by the original author
- code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create

FILE ORGANIZATION

A file consists of sections that should be separated by blank lines and an optional comment identifying each section.

JAVA SOURCE FILES

Each Java source file contains a single public class or interface.

When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class.

The public class should be the first class or interface in the file.

JAVA SOURCE FILES

Java source files have the following ordering:

- beginning comments
- `package` and `import` statements
- `class` and `interface` declarations

CLASS AND INTERFACE DECLARATIONS

The following list describes the parts of a class or interface declaration, in the order that they should appear:

- class/interface documentation comment
- `class` or `interface` statement
- class (`static`) variables
- instance variables
- constructors
- methods

Author: Rafał Roppel

INDENTATION

Four spaces should be used as the unit of indentation.

The exact construction of the indentation (spaces vs. tabs) is unspecified.

Tabs must be set exactly every 8 spaces (not 4).

Author: Rafał Roppel

WRAPPING LINES

When an expression will not fit on a single line, break it according to these general principles:

- break after a comma
- break before an operator
- prefer higher-level breaks to lower-level breaks
- align the new line with the beginning of the expression at the same level on the previous line
- if the above rules lead to confusing code or to code that’s squished up against the right margin, just indent 8 spaces instead

COMMENTS

Java programs can have two kinds of comments:

- implementation comments
- documentation comments

COMMENTS

Implementation comments are those found in C++, which are delimited by `/*...*/`, and `//`.

Documentation comments (known as "doc comments") are Java-only, and are delimited by `/**...*/`.

Doc comments can be extracted to HTML files using the javadoc tool.

Author: Rafał Roppel

COMMENTS

Implementation comments are mean for commenting out code or for comments about the particular implementation.

Doc comments are meant to describe the specification of the code, from an implementation-free perspective, to be read by developers who might not necessarily have the source code at hand.

Author: Rafał Roppel

COMMENTS

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself.

Comments should contain only information that is relevant to reading and understanding the program.

Author: Rafał Roppel

COMMENTS

The frequency of comments sometimes reflects poor quality of code.

When you feel compelled to add a comment, consider rewriting the code to make it clearer.

DECLARATIONS

Number Per Line - one declaration per line is recommended since it encourages commenting

```
int level; // indentation level
int size;  // size of table
```

is preferred over

```
int level, size;
```

DECLARATIONS

Placement - put declarations only at the beginning of blocks. Don’t wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
void MyMethod() {
    int int1;     // beginning of method block

    if (condition) {
        int int2; // beginning of "if" block
        ...
    }
}
```

DECLARATIONS

Initialization - try to initialize local variables where they’re declared.

The only reason not to initialize a variable where it’s declared is if the initial value depends on some computation occurring first.

DECLARATIONS

Class and Interface Declarations - when coding Java classes and interfaces, the following formatting rules should be followed:

- no space between a method name and the parenthesis “(“ starting its parameter list
- open brace “{” appears at the end of the same line as the declaration statement
- closing brace “}” starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the “}” should appear immediately after the “{“
- methods are separated by a blank line

Author: Rafał Roppel

STATEMENTS

Simple Statements - each line should contain at most one statement.

```
argv++; argc--; // AVOID!
```

Do not use the comma operator to group multiple statements unless it is for an obvious reason.

STATEMENTS

Compound Statements are statements that contain lists of statements enclosed in braces “`{ statements }`”.

- the enclosed statements should be indented one more level than the compound statement
- the opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement
- braces are used around all statements, even singletons, when they are part of a control structure, such as a `if-else` or `for` statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces

Author: Rafał Roppel

STATEMENTS

`return` Statements with a value should not use parentheses unless they make the return value more obvious in some way.

```
return;

return myDisk.size();

return (size ? size : defaultSize);
```

Author: Rafał Roppel

STATEMENTS

`if` Statements always use braces {}. Avoid the following error-prone form.

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
    statement;
```

Author: Rafał Roppel

WHITE SPACE

Blank Lines improve readability by setting off sections of code that are logically related.

Two blank lines should always be used in the following circumstances:

- between sections of a source file
- between class and interface definitions

Author: Rafał Roppel

One blank line should always be used in the following circumstances:

- between methods
- between the local variables in a method and its first statement
- before a block or single-line comment
- between logical sections inside a method to improve readability

WHITE SPACE

Blank Spaces should be used in the following circumstances:

- a keyword followed by a parenthesis should be separated by a space
- a blank space should not be used between a method name and its opening parenthesis
- a blank space should appear after commas in argument lists
- all binary operators except `.` should be separated from their operands by spaces

Author: Rafał Roppel

WHITE SPACE

- blank spaces should never separate unary operators such as unary minus, increment (“++”), and decrement (“--”) from their operands
- the expressions in a `for` statement should be separated by blank spaces
- casts should be followed by a blank

Author: Rafał Roppel

NAMING CONVENTIONS

Naming conventions make programs more understandable by making them easier to read.

They can also give information about the function of the identifier — for example, whether it's a constant, package, or class — which can be helpful in understanding the code.

CLASSES

Class names should be nouns, in mixed case with the first letter of each internal word capitalized.

Try to keep your class names simple and descriptive.

Use whole words — avoid acronyms and abbreviations.

INTERFACES

Interface names should be capitalized like class names.

METHODS

Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.

VARIABLES

Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter.
Internal words start with capital letters.

Variable names should be short yet meaningful.
The choice of a variable name should be mnemonic, designed to indicate to the casual observer the intent of its use.

VARIABLES

One-character variable names should be avoided except for temporary “throwaway” variables. Common names for temporary variables are `i`, `j`, `k`.

CONSTANTS

The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores (“_”).

JAVABEANS STANDARD

JavaBeans are Java classes that have *properties*.

Think of properties as `private` instance variables.

Since they're `private`, the only way they can be accessed from outside of their class is through *methods* in the class.

The methods that change a property's value are called **setter** methods, and the methods that retrieve a property's value are called **getter** methods.

Author: Rafał Roppel

JAVABEANS STANDARD

If the property is not a boolean, the getter method's prefix must be `get`.

For example, `getSize()` is a valid JavaBeans getter name for a property named *size*.

You do not need to have a variable named `size`.

The name of the property is inferred from the getters and setters, not through any variables in your class.

What you return from `getSize()` is up to you.

JAVABEANS STANDARD

If the property is a boolean, the getter method's prefix is either `get` or `is`.
For example, `getStopped()` or `isStopped()` are both valid JavaBeans names for a boolean property.

The setter method's prefix must be `set`.
For example, `setSize()` is the valid JavaBean name for a property named *size*.

Author: Rafał Roppel

JAVABEANS STANDARD

To complete the name of a getter or setter method, change the first letter of the property name to uppercase, and then append it to the appropriate prefix (`get`, `is`, or `set`).

Setter method signatures must be marked `public`, with a `void` return type and an argument that represents the property type.

Getter method signatures must be marked `public`, take no arguments, and have a return type that matches the argument type of the setter method for that property.

Author: Rafał Roppel

BEST PRACTICES FROM UNCLE BOB

- Names
  - self-describing
  - easy to pronounce
  - communicate intent
- Constants
  - uppercase
  - long descriptive name
- Variables
  - nouns
  - the smaller scope, the shorter name
  - the larger scope, the longer name

BEST PRACTICES FROM UNCLE BOB

- Functions/methods
  - verbs
  - the smaller scope, the longer name (private); used in several locations
  - the larger scope, the shorter name (public); used from multiple locations
- Classes
  - public - short, concise name
  - private - longer, descriptive name
  - a class that inherits, usually has a longer name (additional determination)

Author: Rafał Roppel

MORE ABOUT CODING CONVENTIONS/STYLE GUIDES

- Code Conventions for the Java ™ Programming Language
- Google Java Style Guide
- Twitter Commons Java Style Guide
- Draft Java Coding Standard by Doug Lea
- Java Programming Style Guide by JavaRanch

JAVA CODE CONVENTIONS

EXERCISES

Author: Rafał Roppel

CLEAN CODE

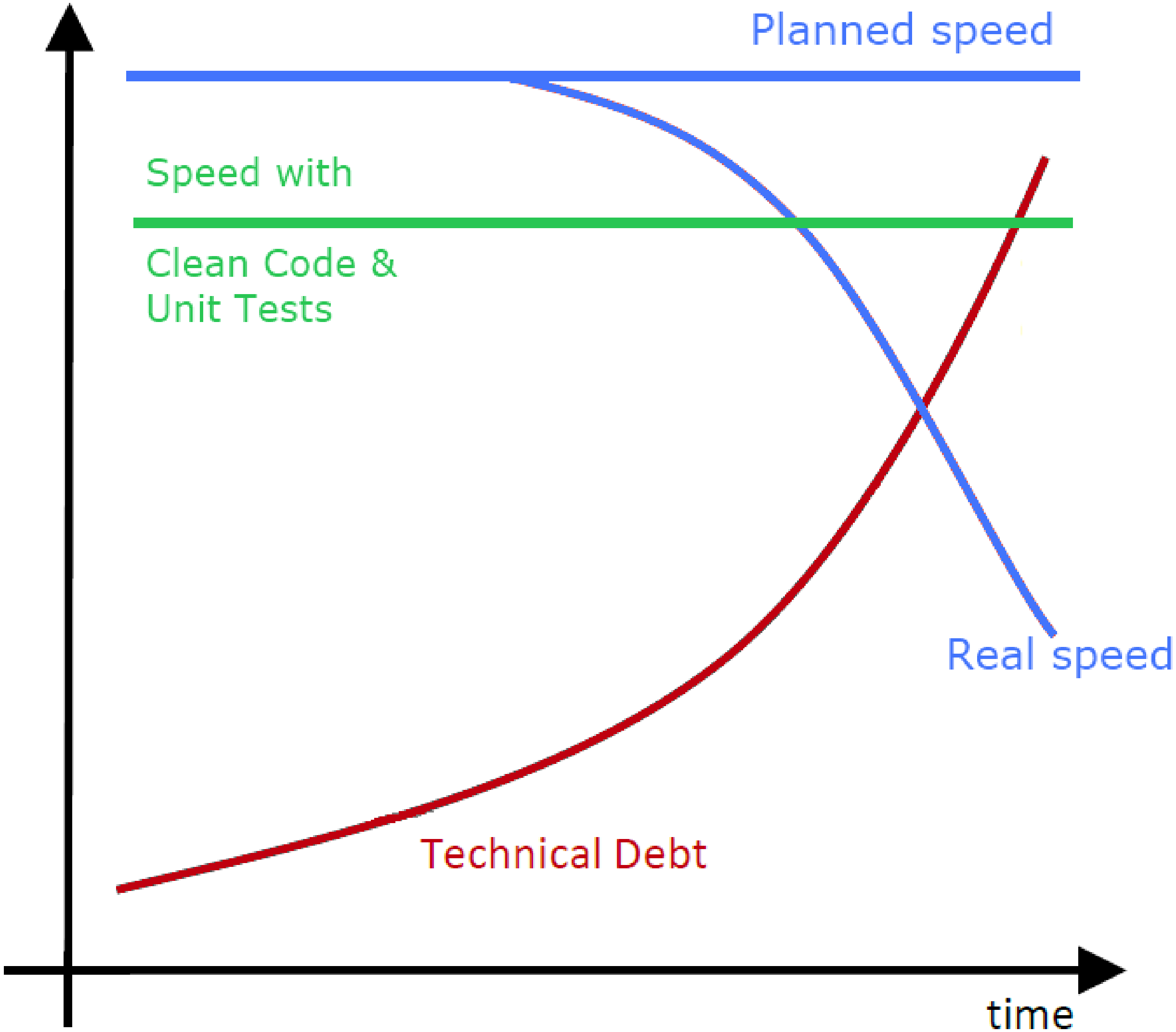
CLEAN CODE

Code is clean if it can be understood easily by everyone on the team.

With understandability comes readability, changeability, extensibility and maintainability.

All the things needed to keep a project going over a long time without accumulating up a large amount of technical debt.

CLEAN CODE



Author: Rafał Roppel

CLEAN CODE

Most software defects are introduced when changing existing code.

The reason behind this is that the developer changing the code cannot fully grasp the effects of the changes made.

Clean code minimises the risk of introducing defects by making the code as easy to understand as possible.

Author: Rafał Roppel

PRINCIPLES

Loose Coupling

Two classes, components or modules are coupled when at least one of them uses the other.
The less these items know about each other, the looser they are coupled.

A component that is only loosely coupled to its environment can be more easily changed or replaced than a strongly coupled component.

PRINCIPLES

High Cohesion

Cohesion is the degree to which elements of a whole belong together.
Methods and fields in a single class and classes of a component should have high cohesion.
High cohesion in classes and components results in simpler, more easily understandable code structure and design.

PRINCIPLES

Change is Local

When a software system has to be maintained, extended and changed for a long time, keeping change local reduces involved costs and risks.
Keeping change local means that there are boundaries in the design which changes do not cross.

PRINCIPLES

It is Easy to Remove

We normally build software by adding, extending or changing features.

However, removing elements is important so that the overall design can be kept as simple as possible.

When a block gets too complicated, it has to be removed and replaced with one or more simpler blocks.

Author: Rafał Roppel

PRINCIPLES

Mind-sized Components

Break your system down into components that are of a size you can grasp within your mind so that you can predict consequences of changes easily.

Author: Rafał Roppel

CLASS DESIGN PRINCIPLES

SOLID

In object-oriented computer programming, the term **SOLID** is a mnemonic acronym for five design principles intended to make software designs more understandable, flexible and maintainable.

The principles are a subset of many principles promoted by Robert C. Martin.

The SOLID acronym was introduced by Michael Feathers.

THE SINGLE RESPONSIBILITY PRINCIPLE

> *A class should have one, and only one, reason to change.*

Author: Rafał Roppel

THE SINGLE RESPONSIBILITY PRINCIPLE

This principle states that an object/class should only have one responsibility and that it should be completely encapsulated by the class.

A responsibility, we mean a reason to change.

This principle will lead to a stronger cohesion in the class and looser coupling between dependency classes, better readability, and code with lower complexity.

THE SINGLE RESPONSIBILITY PRINCIPLE

It is much more difficult to understand and edit a class when it has various responsibilities.

So if we have more than one reason to change, the functionality will be split into two classes and each will handle its own responsibility.

We care about separating the functionalities because each responsibility is an access of change.

Author: Rafał Roppel

THE SINGLE RESPONSIBILITY PRINCIPLE

When a class has more than a single responsibility, those responsibilities become coupled.

This coupling can lead to a fragile code base that is difficult to refactor when your requirements emerge.

THE OPEN CLOSED PRINCIPLE

> *You should be able to extend a classes behavior, without modifying it.*

THE OPEN CLOSED PRINCIPLE

The idea is that an entity allows its behavior to be extended but never by modifying its source code.

Any class (or whatever you write) should be written in such a way that it can be used as is.

It can be extended if need be, but it can never be modified.

You can consider this when you are writing your classes.

THE OPEN CLOSED PRINCIPLE

Use the class in any way you need, but modifying its behavior comes by adding new code, never by modifying the old.

The same principle can be applied to modules, packages, and libraries.

By applying the open-closed principle you will get a loose coupling, you will improve readability and, finally, you will be reducing the risk of breaking existing functionality.

THE LISKOV SUBSTITUTION PRINCIPLE

> *Derived classes must be substitutable for their base classes.*

THE LISKOV SUBSTITUTION PRINCIPLE

As its name says, was defined by Barbara Liskov.

The idea here is that objects should be replaceable by instances of their subtypes without affecting the functioning of your system from a client’s point of view.

Basically, instead of using the actual implementation, you should always be able to use a base class and get the result you were waiting for.

Author: Rafał Roppel

THE LISKOV SUBSTITUTION PRINCIPLE

Often when we want to represent an object, we model our classes based on their properties, but, instead of that, we should actually be putting more of our focus on the behaviors.

This principle basically confirms that our abstractions are correct and helps us to get code that is easily reusable and class hierarchies that are very easily understood.

THE INTERFACE SEGREGATION PRINCIPLE

Make fine grained interfaces that are client specific.

THE INTERFACE SEGREGATION PRINCIPLE

It’s about how to write interfaces.

Once an interface is becoming too large/fat, we absolutely need to split it into small interfaces that are more specific.

And the interface will be defined by the client that will use it, which means that the client of the interface will only know about the methods that are related to them.

THE INTERFACE SEGREGATION PRINCIPLE

Actually, if you add methods that shouldn’t be there, the classes implementing the interface will have to implement those methods as well.

That is why the client shouldn’t be forced to depend on interfaces that they don’t use.

ISP is intended to keep a system decoupled and thus easier to refactor, change, and deploy.

THE DEPENDENCY INVERSION PRINCIPLE

> *Depend on abstractions, not on concretions.*

Author: Rafał Roppel

THE DEPENDENCY INVERSION PRINCIPLE

This principle is primarily concerned with reducing dependencies amongst the code modules.

It will be helpful when it comes to understanding how to correctly tie your system together.

If your implementation detail will depend on the higher-level abstractions, it will help you to get a system that is coupled correctly.

It will influence the encapsulation and cohesion of that system.

MORE ABOUT SOLID

- SRP: The Single Responsibility Principle
- OCP: The Open-Closed Principle
- LSP: The Liskov Substitution Principle
- ISP: The Interface Segregation Principle
- DIP: The Dependency Inversion Principle

PACKAGE COHESION PRINCIPLES

THE RELEASE REUSE EQUIVALENCY PRINCIPLE

> *The granule of reuse is the granule of release.*

Author: Rafał Roppel

THE COMMON CLOSURE PRINCIPLE

Classes that change together are packaged together.

Author: Rafał Roppel

THE COMMON REUSE PRINCIPLE

> *Classes that are used together are packaged together.*

MORE ABOUT PACKAGE COHESION PRINCIPLES

- The Reuse/Release Equivalence Principle (REP) (page 4)
- The Common Closure Principle (CCP) (page 6)
- The Common Reuse Principle (CRP) (page 5)

PACKAGE COUPLINGS PRINCIPLES

Author: Rafał Roppel

THE ACYCLIC DEPENDENCIES PRINCIPLE

The dependency graph of packages must have no cycles.

THE STABLE DEPENDENCIES PRINCIPLE

Depend in the direction of stability.

THE STABLE ABSTRACTIONS PRINCIPLE

Abstractness increases with stability.

MORE ABOUT PACKAGE COUPLINGS PRINCIPLES

- The Acyclic Dependencies Principle (ADP) (page 6)
- The Stable Dependencies Principle (SDP) (page 8)
- The Stable Abstractions Principle (SAP) (page 11)

GRASP

GRASP

General **R**esponsibility **A**ssignment **S**oftware **P**rinciples help guide object-oriented design by clearly outlining WHO does WHAT.

Which object or class is responsible for what action or role?

GRASP also helps us define how classes work with one another.

The key point of GRASP is to have efficient, clean, understandable code.

Within GRASP there are nine principles that we want to cover.

Author: Rafał Roppel

CONTROLLER

- deals with how to delegate the request from the UI layer objects to domain layer objects
- when a request comes from UI layer object, Controller pattern helps us in determining what is that first object that receive the message from the UI layer objects
- this object is called controller object which receives request from UI layer object and then controls/coordinates with other object of the domain layer to fulfill the request

CONTROLLER

- it delegates the work to other class and coordinates the overall activity
- we can make an object as Controller, if:
    - object represents the overall system (*facade controller*)
    - object represent a use case, handling a sequence of operations (*session controller*)

Author: Rafał Roppel

CREATOR

- who creates an Object? Or who should create a new instance of some class?
- “Container” object creates “contained” objects
- decide who can be creator based on the objects association and their interaction

HIGH COHESION

- how are the operations of any element are functionally related?
- related responsibilities in to one manageable unit
- prefer high cohesion
- clearly defines the purpose of the element

Author: Rafał Roppel

INDIRECTION

- how can we avoid a direct coupling between two or more elements
- indirection introduces an intermediate unit to communicate between the other units, so that the other units are not directly coupled
- benefits: low coupling

Author: Rafał Roppel

INFORMATION EXPERT

- given an object `o`, which responsibilities can be assigned to `o`?
- expert principle says – assign those responsibilities to `o` for which `o` has the information to fulfill that responsibility
- they have all the information needed to perform operations, or in some cases they collaborate with others to fulfill their responsibilities

LOW COUPLING

- how strongly the objects are connected to each other?
- coupling – object depending on other object
- when depended upon element changes, it affects the dependant also
- low coupling – how can we reduce the impact of change in depended upon elements on dependant elements

Author: Rafał Roppel

LOW COUPLING

- prefer low coupling – assign responsibilities so that coupling remain low
- minimizes the dependency hence making system maintainable, efficient and code reusable
- two elements are coupled, if:
    - one element has aggregation/composition association with another element
    - one element implements/extends other element

Author: Rafał Roppel

POLYMORPHISM

- how to handle related but varying elements based on element type?
- polymorphism guides us in deciding which object is responsible for handling those varying elements
- benefits: handling new variations will become easy

PROTECTED VARIATIONS

- how to avoid impact of variations of some elements on the other elements
- it provides a well defined interface so that the there will be no affect on other units
- provides flexibility and protection from variations
- provides more structured design

PURE FABRICATION

- fabricated class/ artificial class – assign set of related responsibilities that doesn't represent any domain object
- provides a highly cohesive set of activities
- behavioral decomposed – implements some algorithm
- benefits: high cohesion, low coupling and can reuse this class

Author: Rafał Roppel

GENERAL PRINCIPLES

GENERAL PRINCIPLES

Follow Standard Conventions

Coding-, architecture-, design guidelines

Keep it Simple, Stupid (KISS)

Simpler is always better. Reduce complexity as much as possible.

Boy Scout Rule

Leave the campground cleaner than you found it.

GENERAL PRINCIPLES

Root Cause Analysis

Always look for the root cause of a problem. Otherwise, it will get you again.

Keep Configurable Data at High Levels

If you have a constant such as default or configuration value that is known and expected at a high level of abstraction, do not bury it in a low-level function. Expose it as an argument to the low-level function called from the high-level function.

GENERAL PRINCIPLES

Prefer Polymorphism To If/Else or Switch/Case

There may be no more than one switch statement for a given type of selection. The cases in that switch statement must create polymorphic objects that take the place of other such switch statements in the rest of the system.

Symmetry / Analogy

Favour symmetric designs (e.g. Load – Save) and designs that follow analogies

GENERAL PRINCIPLES

Separate Multi-Threading Code

Do not mix code that handles multi-threading aspects with the rest of the code. Separate them into different classes.

Transitive Navigation

Aka Law of Demeter, writing shy code. A module should know only its direct dependencies.

GENERAL PRINCIPLES

Feature Envy

The methods of a class should be interested in the variables and functions of the class they belong to, and not the variables and functions of other classes. Using accessors and mutators of some other object to manipulate its data, is envying the scope of the other object.

Duplication

Eliminate duplication. Violation of the "Don't repeat yourself" (DRY) principle.

GENERAL PRINCIPLES

Magic Numbers / Strings

Replace Magic Numbers and Strings with named constants to give them a meaningful name when meaning cannot be derived from the value itself.

Encapsulate Boundary Conditions

Boundary conditions are hard to keep track of. Put the processing for them in one place, e.g.

```
nextLevel = level + 1;
```

Author: Rafał Roppel

GENERAL PRINCIPLES

Prefer Dedicated Value Objects to Primitive Types

Instead of passing primitive types like strings and integers, use dedicated primitive types: e.g. AbsolutePath instead of string.

Methods Should Do One Thing

Loops, exception handling, … encapsulate in sub-methods.

Author: Rafał Roppel

GENERAL PRINCIPLES

Method with Too Many Arguments

Prefer fewer arguments. Maybe functionality can be outsourced to a dedicated class that holds the information in fields.

Selector / Flag Arguments

```
public int Foo(bool flag)
```

Split method into several independent methods that can be called from the client without the flag.

GENERAL PRINCIPLES

Catch Specific Exceptions

Catch exceptions as specific as possible. Catch only the exceptions for which you can react in a meaningful manner.

Catch Where You Can React in a Meaningful Way

Only catch exceptions when you can react in a meaningful way. Otherwise, let someone up in the call stack react to it.

GENERAL PRINCIPLES

Use Exceptions instead of Return Codes or null

In an exceptional case, throw an exception when your method cannot do its job. Don't accept or return null. Don't return error codes.

Fail Fast

Exceptions should be thrown as early as possible after detecting an exceptional case. This helps to pinpoint the exact location of the problem by looking at the stack trace of the exception.

GENERAL PRINCIPLES

Using Exceptions for Control Flow

Using exceptions for control flow: has bad performance, is hard to understand and results in very hard handling of real exceptional cases.

Swallowing Exceptions

Exceptions can be swallowed only if the exceptional case is completely resolved after leaving the catch block. Otherwise, the system is left in an inconsistent state.

Author: Rafał Roppel

CLEAN CODE

EXERCISES

- Clean Code
- SOLID
- GRASP
- General Principles

* MASTERING LIBRARIES

- Guava: Google Core Libraries for Java
- Apache Commons
- Vavr (formerly called Javaslang)

*) optional

Author: Rafał Roppel

FURTHER READING

- Agile Software Development, Principles, Patterns, and Practices
- Clean Code: A Handbook of Agile Software Craftsmanship
- The Clean Coder: A Code of Conduct for Professional Programmers
- The Software Craftsman: Professionalism, Pragmatism, Pride
- The Pragmatic Programmer
- Clean Architecture: A Craftsman's Guide to Software Structure and Design
- Working Effectively with Legacy Code

Author: Rafał Roppel