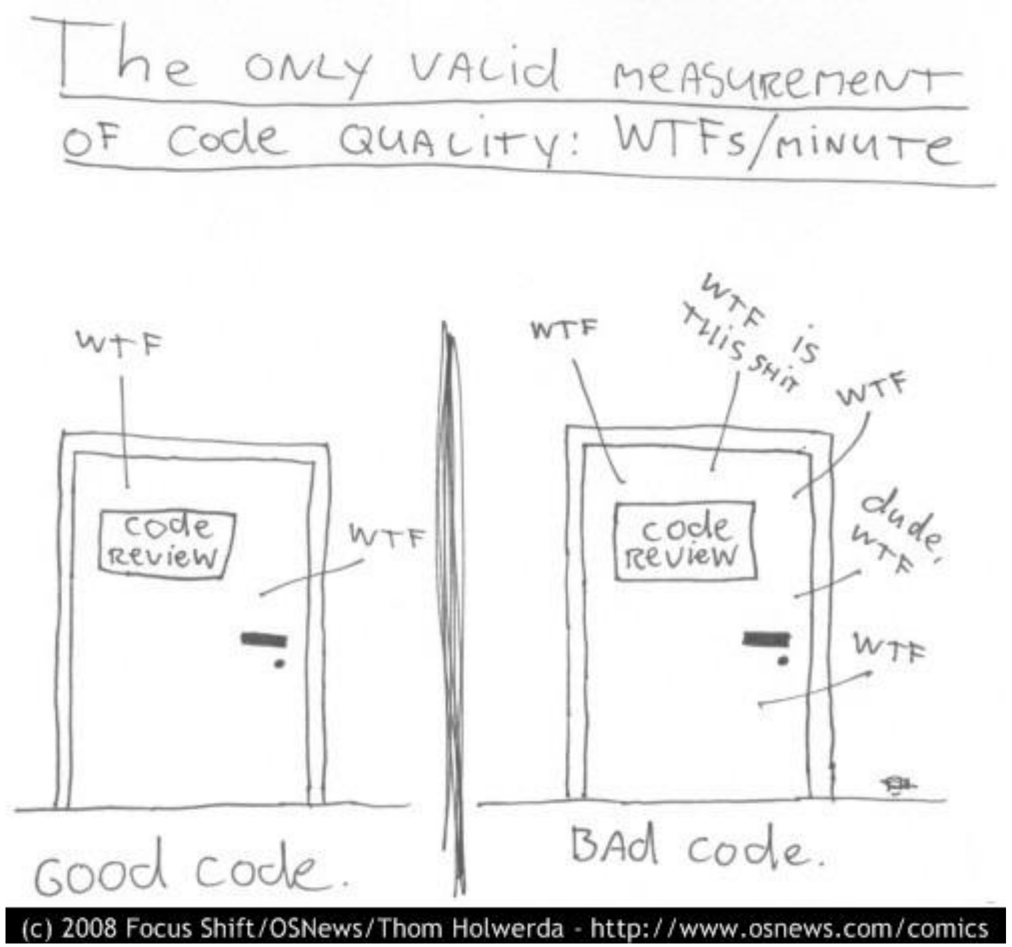
Programmeertechnieken/Programming Techniques

Part 5b
Clean code

Koen Pelsmaekers
Campus Groep T, 2022-2023

Clean code matters

- Readability
- Maintainability
- “Don’t give bugs a place to hide” (Brian Goetz)
- *“Leave the campground cleaner than you found it”*



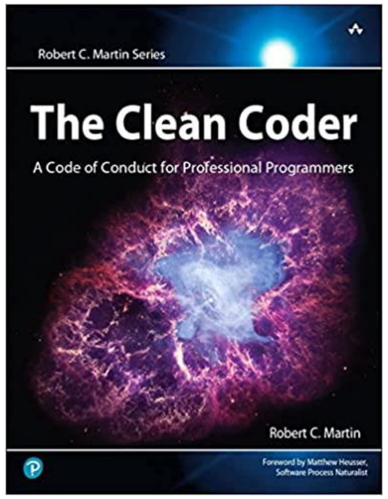
The only valid measurement of code quality: WTFs/minute
WTF
code review
WTF
WTF
WTF is this shit
WTF
code review
dude, WTF
WTF
Good code.
Bad code.
(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

Clean Code in a nutshell

The following slides contain a summary from the Clean Code book by Robert C. Martin (see next slide).

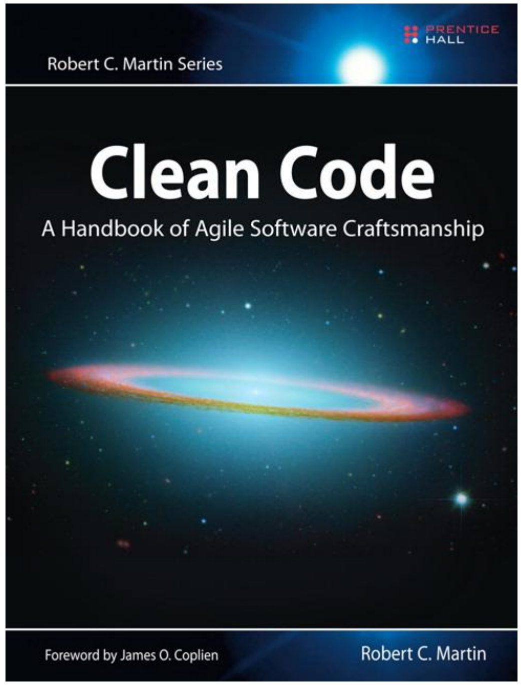
Every programming should have his/her copy of this book!

And then read The Clean Coder…


Robert C. Martin Series
The Clean Coder
A Code of Conduct for Professional Programmers
Robert C. Martin
Foreword by Matthew Heusser, Software Process Naturalist

The bible of writing “Good Code”

Clean Code, A Handbook of Agile Software Craftmanship, Robert “uncle Bob” C. Martin.


PRENTICE HALL
Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

“@author, we are authors”

“code is written to be read”

“ratio of time spent reading vs. writing code is 10:1”

Meaningful names

for variables, functions/methods, arguments, classes, interfaces, packages, …

- use intention-revealing names, use pronounceable names, use searchable names
- “length of a name should correspond to the size of its scope”
- for a “professional” programmer “clarity” is king
- class names should be nouns
- method names should be verbs or verb phrase names
- use static factory methods with names that describe arguments (make constructor private)
- don’t be cute, “say what you mean, mean what you say”
- pick one word per concept

Functions

- Small!
  - no nested blocks (indent level should not be greater than one or two), blocks within if, else, while statements should be one line long
- Do one thing, on the same level of abstraction (SRP = Single Responsibility Principle)
- Arguments: “niladic” > “monadic” > “dyadic”
  - avoid three or more arguments, use parameter objects
  - avoid “output” arguments, in favor of return values
  - avoid boolean “flag” arguments => function is doing more than 1 thing
  - common monadic funtions
    - ask a question about the single argument: boolean fileExists(“filename.txt”)
    - transform the argument in an object of another type (like “map”) and return it: InputStream fileOpen(“filename.txt”)
- Should have no side effects (see later)
- DRY (Don’t Repeat Yourself)
  - avoid duplicate code
- Refactor!

Comments

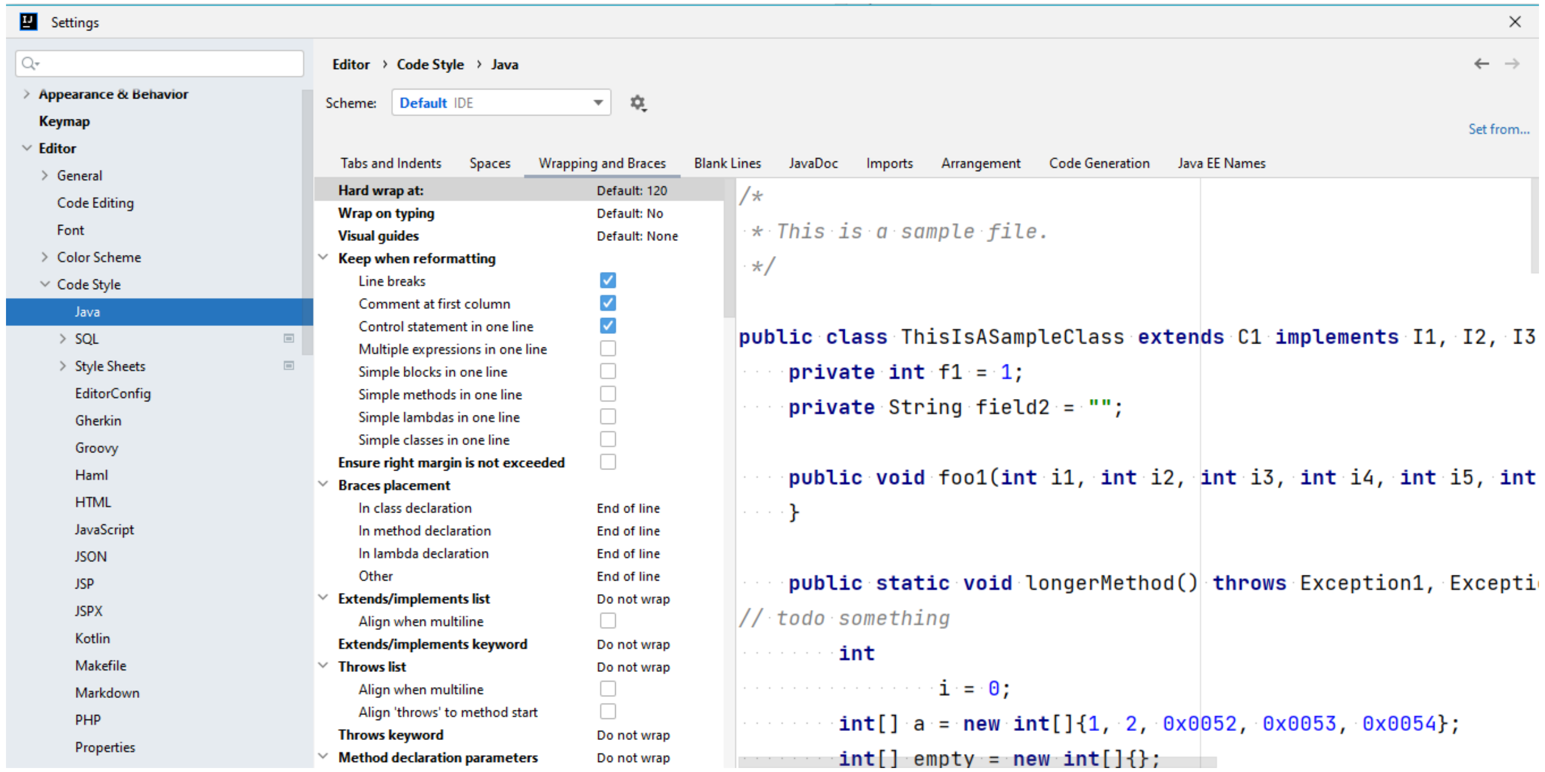
- “Don’t comment bad code – rewrite it.” (Brian W. Kernighan)
- Comments often lie, because they are not maintained together with the code
- Debug your code, not your comments
- No journal comments, should be committed in source control repository
- No commented-out code, should be committed in source control repository as a separate branch

Formatting code

- Vertical formatting
  - vertical openness between concepts
  - vertical density for tightly related code
  - vertical distance: related concepts should be kept vertically close
    - for instance: variable declaration as close to usage as possible
  - vertical ordering: from high-level to low-level
- Horizontal formatting
  - line width? do not scroll? 120 characters
  - horizontal openness
    - spaces around “=“, spaces within expressions (factors vs. terms), arguments, …
  - identation

Formatting code: “be consistent”

- team or company wide coding style
- enforced by IDE



Scheme: Default IDE

Hard wrap at:	Default: 120
Wrap on typing	Default: No
Visual guides	Default: None
Keep when reformatting	
Line breaks	✓
Comment at first column	✓
Control statement in one line	✓
Multiple expressions in one line	☐
Simple blocks in one line	☐
Simple methods in one line	☐
Simple lambdas in one line	☐
Simple classes in one line	☐
Ensure right margin is not exceeded	☐
Braces placement	
In class declaration	End of line
In method declaration	End of line
In lambda declaration	End of line
Other	End of line
Extends/implements list	Do not wrap
Align when multiline	☐
Extends/implements keyword	Do not wrap
Throws list	Do not wrap
Align when multiline	☐
Align 'throws' to method start	☐
Throws keyword	Do not wrap
Method declaration parameters	Do not wrap

```
/*
 * This is a sample file.
 */

public class ThisIsASampleClass extends C1 implements I1, I2, I3
    private int f1 = 1;
    private String field2 = "";

    public void foo1(int i1, int i2, int i3, int i4, int i5, int
    }

    public static void longerMethod() throws Exception1, Excepti
// todo something
        int
                i = 0;
        int[] a = new int[]{1, 2, 0x0052, 0x0053, 0x0054};
        int[] empty = new int[]{};
```

KU LEUVEN

Objects

- Information hiding within objects
  - do not expose the internals (for instance data structures used) of an object: “expose behavior/hide data”
  - Law of Demeter
    - “don’t talk to strangers”

Unit tests

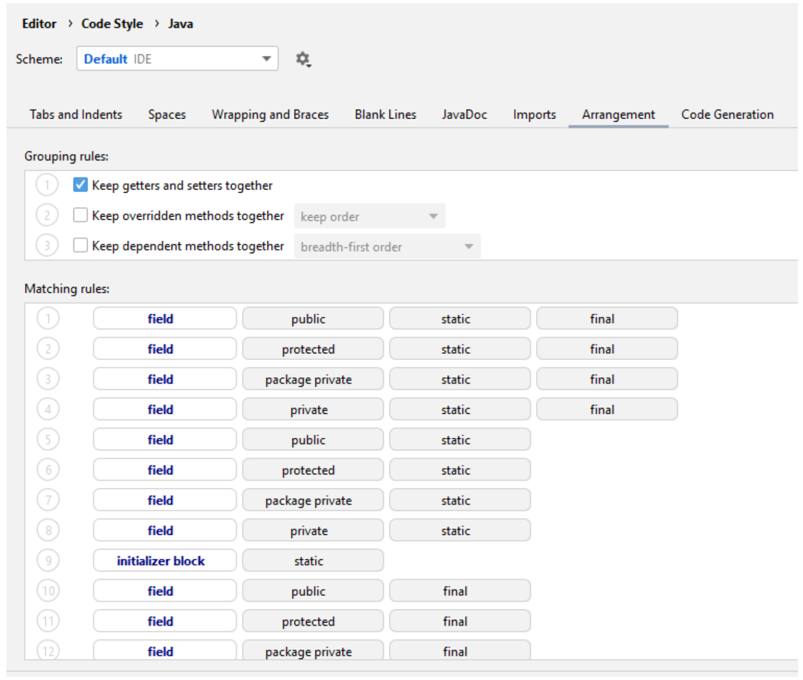
- Test Driven Development
    - write (automatic) unit tests first

will be discussed in

“Software Engineering and Web Technology (B-KUL-T3WSW2)”

Clean Classes

- Encapsulation: private fields, private utility functions
- Classes should be small
  - Single Responsibility Principle (SRP)
- Well organized code
  - IDE support: Arrange code



Scheme: Default IDE

Grouping rules:

1. [✓] Keep getters and setters together
2. [ ] Keep overridden methods together — keep order
3. [ ] Keep dependent methods together — breadth-first order

Matching rules:

1	field	public	static	final
2	field	protected	static	final
3	field	package private	static	final
4	field	private	static	final
5	field	public	static	
6	field	protected	static	
7	field	package private	static	
8	field	private	static	
9	initializer block	static		
10	field	public	final	
11	field	protected	final	
12	field	package private	final	

KU LEUVEN

Other good practices

Program to an interface (good design principle)

- Decouple declaration from implementation

  “What” versus “How”

- Information hiding or Encapsulation

  Do not expose the internals of your implementation

- Defer choice of actual class

Criteria for designing a good interface

- **Cohesion**
  implements a single abstraction
- **Completeness**
  provides all operations necessary
- **Convenience**
  makes common tasks simple
- **Clarity**
  do not confuse your programmers
- **Consistency**
  keep the level of abstraction

Avoid side effects

- A function "promises" to do one thing (and returns a value), but it also does other *hidden* things
  - unexpected changes to fields, to parameters or to global variables
  - temporal couplings: order of evaluation matters
  - perform I/O
- Avoid side effects by introducing functional programming
  - the output of one function is passed to the next, without changing the content of other variables

Core Java, Volume I-Fundamentals
(Cay Horstmann)

- Class design hints
  - Always keep data private
  - Always initialize data
  - Don’t use too many basic types in a class (Introduce Objects)
  - Not all fields need individual field accessors and mutators
  - Break up classes that have too many responsibilities
  - Make the names of your classes and methods reflect their responsibilities
  - Prefer immutable classes