VOLVO

Micael Andersson
Senior
Architect & Developer

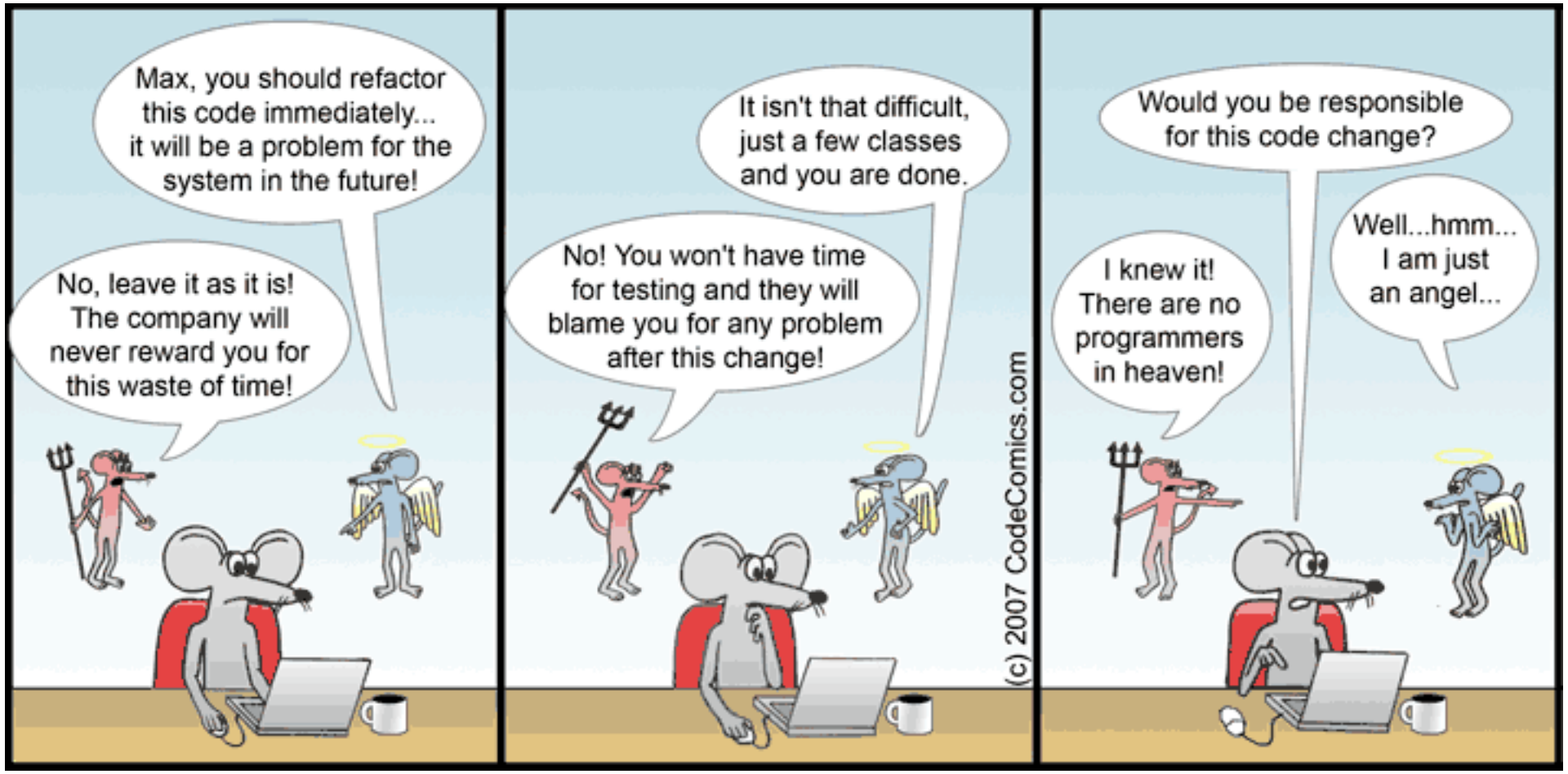
Refactoring

Introduction to developers

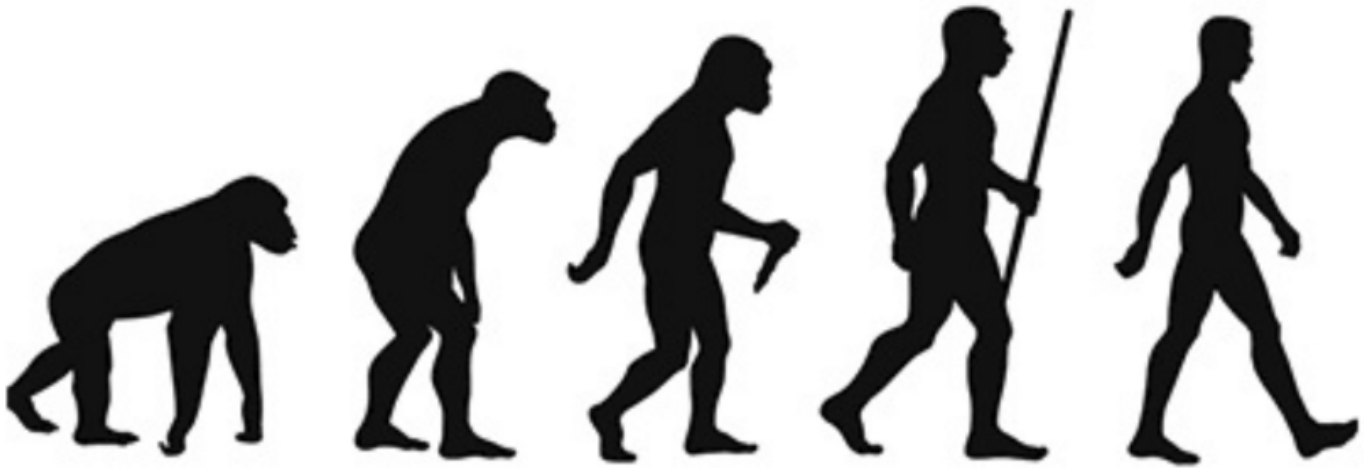
Micael Andersson
Senior Architect and Developer

September 2022

Pitch


Max, you should refactor this code immediately... it will be a problem for the system in the future!
No, leave it as it is! The company will never reward you for this waste of time!
It isn't that difficult, just a few classes and you are done.
No! You won't have time for testing and they will blame you for any problem after this change!
(c) 2007 CodeComics.com
Would you be responsible for this code change?
I knew it! There are no programmers in heaven!
Well...hmm... I am just an angel...

Improvement ideas



https://www.slideshare.net/NeilGreen1/the-roi-of-refactoring-lego-vs-playdoh

Goals of this presentation

- **Introduce and motivate** the Refactoring practice
- Describe how refactoring **fits within** a software development process
- **Hands-on** experience with Refactoring using a simple yet interesting enough sample project
- Hints on Catalog of “Bad smells”
- Hints on Catalog of “Refactorings”

Refactoring - Definition

Refactoring is a disciplined technique for restructuring an existing body of code through a series of small **behaviour preserving transformations**, in order to improve its internal quality

a refactoring (Noun) – A **change made to the internal structure** of software to make it easier to understand and modify **without changing its external behaviour**.

to refactor (Verb) – A disciplined technique for restructuring an existing body of code, **altering its internal** structure **without changing its external** behaviour.

Refactoring - the idea

Each transformation (called a 'refactoring') **does little**, but a sequence of transformations can produce a significant restructuring.

Since each refactoring is **small, it's less likely to go wrong**.

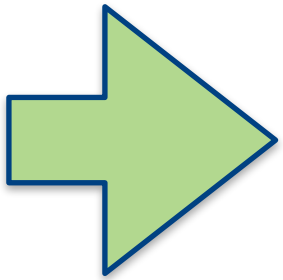
The system is also **kept fully working** after each small refactoring, reducing the risk that a system can get seriously broken during the restructuring.

What is Refactoring? - an example

It is changing the structure of code without changing its functionality.

This is a rather unwieldy (Swedish: tungrodd) code. Even beginners would rarely write something like this, but as an example.

```
public boolean max(int a, int b) {
    if(a > b) {
        return true;
    } else if (a == b) {
        return false;
    } else {
        return false;
    }
}
```



```
public boolean max(int a, int b) {
    return a > b;
}
```

Remember that **refactoring does not mean reduction**.
Its main purpose is to improve the structure of code.

Why do Refactoring?

Several reasons:

- To achieve simplicity and brevity in code. Refactoring improves understanding of code written by other developers.
- It helps find and fix bugs.
- It can accelerate the speed of software development.
- Overall, it improves software design.

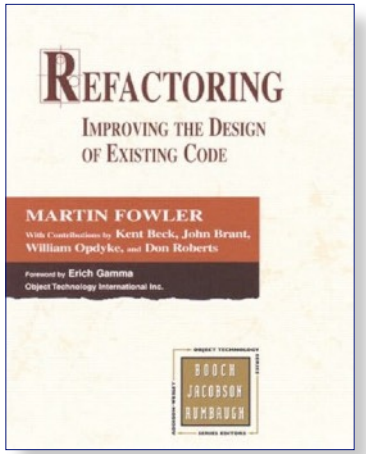
If refactoring is not performed for a long time, development may encounter difficulties, including a complete stop to the work.

Where did Refactoring come from ?

Two of the first people to recognise the importance of refactoring were Kent Beck and Ward Cunningham, working within the **Smalltalk** community.

Ralph Johnson's work with refactoring and frameworks has also been an important contribution.

Martin Fowler systematised the ideas in his excellent book Refactoring from 1999



by Martin Fowler

Context: Change is the norm, not the exception!

External Changes

- Requirements
- Development Team
- Technology
- …

Internal Changes

- Size
- Complexity
- Coupling
- …

External Changes

Requirements mostly focus on **today's needs**, not on tomorrow's

- -> You will have to implement unanticipated features, that you didn't forsee.

Software Development means **constantly mapping needs to solutions**

- -> You will have to reconsider decisions made from time to time

Software is loaded with **implicit assumptions**:
Loss of people typically means loss of knowledge

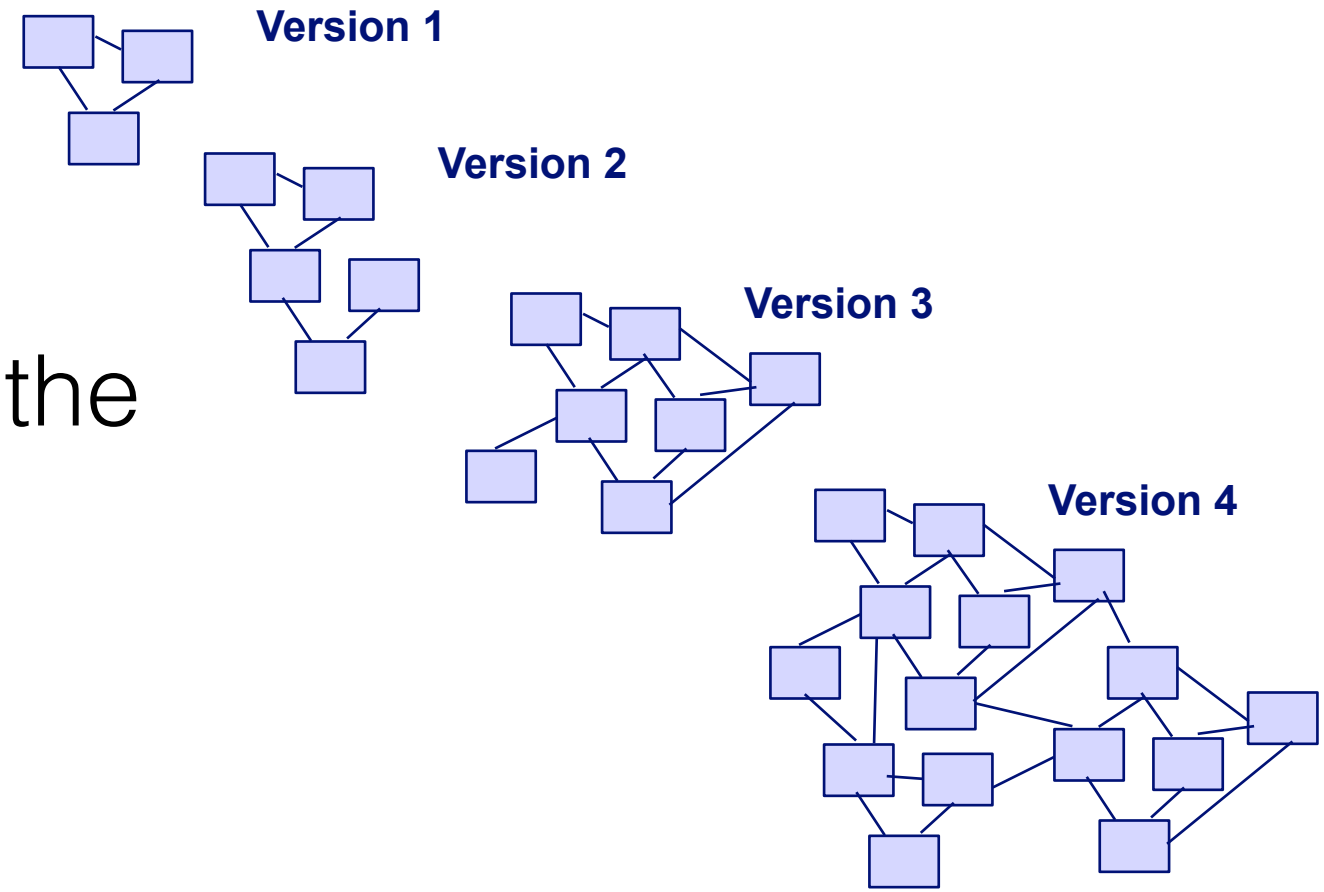
- ->You will have to pin down that knowledge into the system itself (or you'll loose it)

Technologies come and go

- -> You will have to adapt your system to new technology

Internal Changes

You will need to keep the size, complexity and couplings reasonable over time



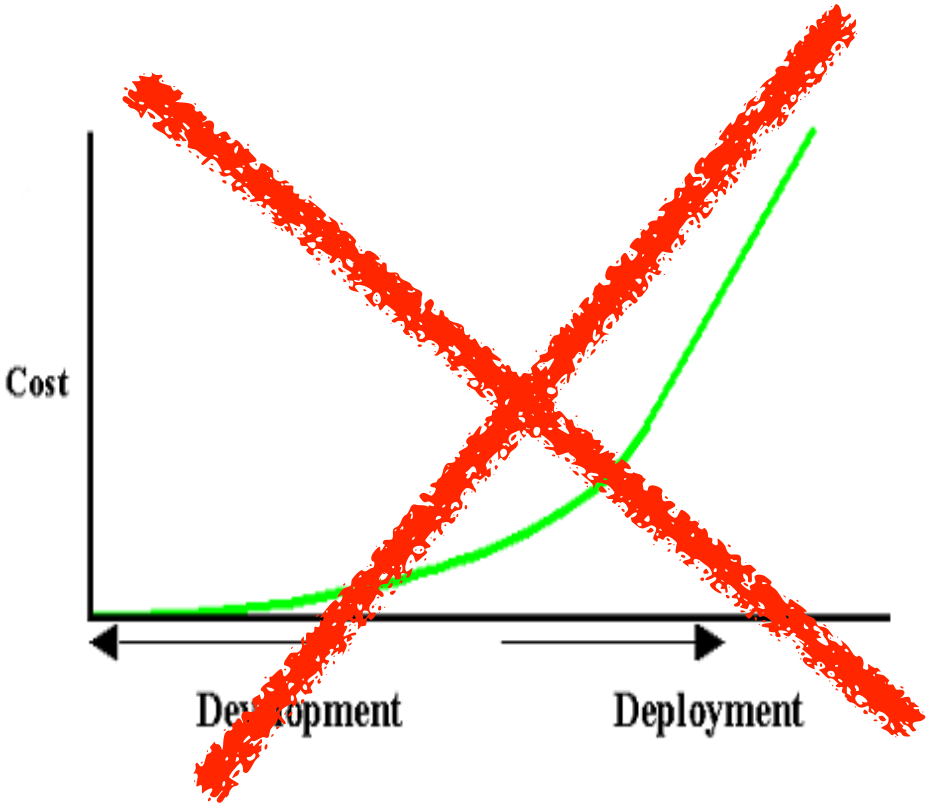
Refactoring Philosophy

It's hard to get the design perfect the first time

- So let's not even pretend we could …
- Make a first reasonable design that should work, but...
- Make sure you have a Plan B
- Mentally **prepare for changes**
  - As implementers discover better designs
  - As your stakeholders change the requirements

But how can we ensure that

- changes are safe?
- doesn't cost a fortune?
- Can we escape from the scenario that changes cost exponential more later in time?



Refactoring Methodology

Make all **changes small and methodical**

- Follow mechanical patterns (which could be automated in some cases) called ***refactorings***, which are *semantics-preserving*

Re-test the system after each change

- By rerunning all of your unit tests
- If something breaks, you know what caused it (you did!)
- Notice: we need fully automated tests for this to work!

Switching Hats

Refactoring hat

- You are updating the design of your code, but not changing what it does. You can thus rerun existing tests to make sure the change works.

Bug-fixing/feature-adding hat

- You are modifying the functionality of the code.

Switch hats frequently …

- … but remember which hat you are currently using, to be sure that you are reaching your end goal.

Refactoring: Hands-on Example

Simple application which **calculates price and frequent renter points based on Movie classifications** (Regular, New Releases and Children), and prints a customer summary statement.

Needs to be changed:

- Add HTML version of customer summary statement
- Change classification system (in a yet unspecified) way

Refactoring: What it isn't

Just any change to the code base

- Refactoring always have a clear purpose, and proceeds in small, mechanical and hence deterministic steps

An excuse for not thinking

- Your initial design should be the best possible effort, given what you know so far
- Before choosing the simplest possible solution, try to imagine a migration route to the alternative solutions. If it seems easy, go ahead. If it seems hard or even impossible, put more effort into make the correct design decision.

Why Refactor?

Refactoring improves design

- Fights against “code decay” as people make changes

Refactoring makes code easier to understand

- Simplifies complicated code, eliminates duplication

Refactoring helps you find bugs

- In order to make refactorings, you need to clarify your understanding of the code. Makes bugs easier to spot.

Refactoring helps you program faster

- Good design = rapid development

When to Refactor ?

Refactor before you add a feature

- Make it easier for you to add the feature

When you need to fix a bug

- If you get a bug report its a sign the code needs refactoring because the code was not clear enough for you to see the bug in the first place.

When to Refactor (Continued…)

When you do a Code Review

- Code reviews help spread knowledge through the development team.
- Works best with small review groups
  - For larger review groups, maybe a Design Review is more appropriate?
- Works excellent with Pair Programming

The “Rule of Three”

The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor.

The “Rule of Three”:

- Three strikes and you refactor

Refactoring in the overall TDD process

Prerequisite: A working (and tested) piece of code

1. Analyse Change Request
2. **Refactor** to prepare for change
3. Test to assure working version
4. Integrate (check-in)
5. Implement the Change Request
6. Test to assure working version
7. **Refactor** to clean up, if necessary
8. Test to assure working version
9. Integrate (check-in)

Refactoring and Performance

Some refactorings may cause the software to run more slowly

This may or may not be a problem

- Normally only a small fraction of the code base really influence the performance
- Hence if all code is optimised equally, >90% of the optimisations may be waste

Refactoring and Performance (contd.)

Most people's intuitions about performance turns out to be wrong;

- a Performance Profiler should be used to get real data

A well-factored program is normally very well suited for highly efficient performance tuning.

Bottom line:

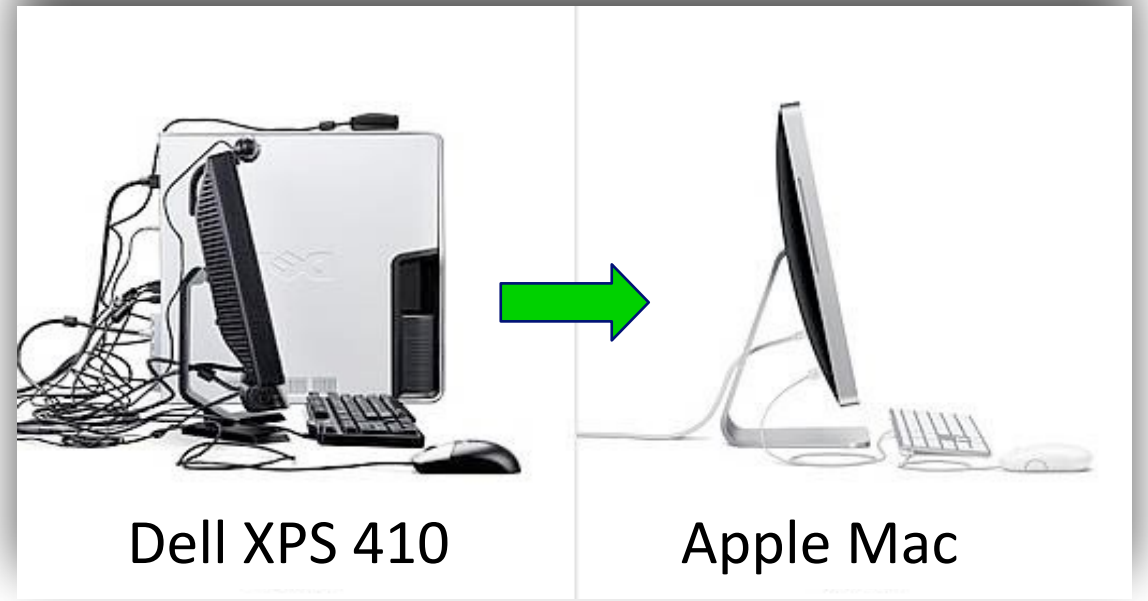
- **Make it work**
- **Make it clean**
- **Make it fast**

Simple, Evolutionary Design

- Take the ”simple is best” approach
- Use refactoring to make complex code simpler
- Gradually, the design will evolve and be refined, slim and fit for its purpose

“Simplicity is more complicated than you think. But it’s well worth it.”

Ron Jeffries



One of the three founders of the Extreme Programming (XP)

Simple Design Criteria (also in TDD Presentation)

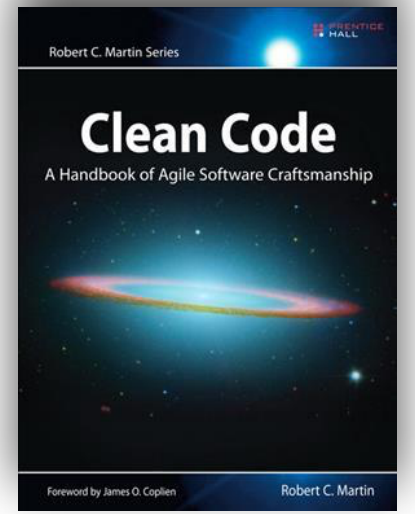
In Priority Order

- The code **is appropriate** for the intended audience (exposed APIs)
- The code **passes** all the tests
- The code **communicates everything** it needs to
- The code has the **smallest number** of classes
- Each class has the **smallest number** of methods

Should we then have all code in one class and only have one method, the “main”-method?

Of course not, but why?

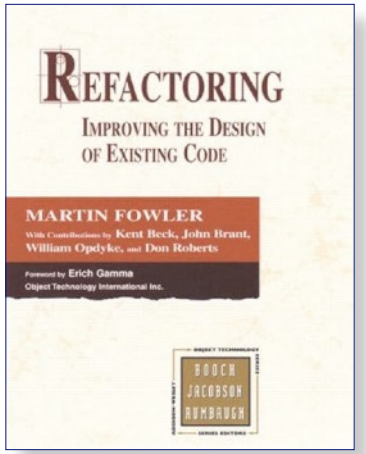
It would break the **SRP** (Single Responsibility Principle) e.t.c.
A lot more answers could be found in the books around Clean Code.



by Robert C Martin

Refactoring Catalog

Add Parameter
Change Association
Reference to value
Value to reference
Collapse hierarchy
Consolidate conditionals
Procedures to objects
Decompose conditional
Encapsulate collection
Encapsulate downcast
Encapsulate field
Extract class
Extract Interface
Extract method
Extract subclass
Extract superclass
Form template method
Hide delegate
Hide method
Inline class
Inline temp
Introduce assertion
Introduce explain variable
Introduce foreign method



by Martin Fowler

Bad Smells Catalog

Duplicated Code

Long Method

Large Class

Long Parameter List

Divergent Change

Shotgun Surgery

Feature Envy

Data Clumps

Primitive Obsession

Switch Statements

Parallel Interface Hierarchies

Lazy Class

Speculative Generality

Temporary Field

Message Chains

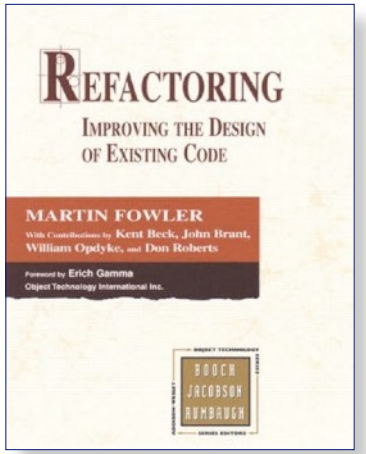
Middle Man

Inappropriate Intimacy

Incomplete Library Class

Data Class

Refused Bequest



by Martin Fowler

VOLVO

Thank you for listening!

Exercises

September 2022