SOFTHOUSE

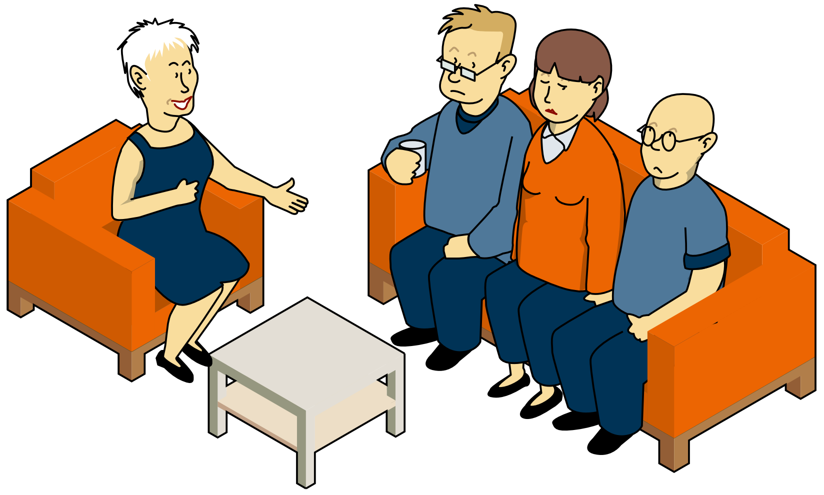
CLEAN CODE

in five minutes

UNDER PRESSURE ...

Acme Inc., 5 p.m. on a Thursday afternoon:

The Product Owner:	*The submission form finally looks good!*
Developer # 3:	*Thanks! At least when you see it in the browser window ...*
The Product Owner:	*What about moving it to the subscription page as well? Could we have that by – let's say – next week?*



Silence falls in the room. What the Product Owner has not yet understood is that the submission form was produced in a hurry. Functions are long and their names do not reveal much about what the code is really supposed to accomplish. Functionality is duplicated in many places. In addition, no one had time to make sure that the form was developed in a way that made it independent of the page that displayed it.

The truth is that there is no way that the Product Owner's suggestion can be accomplished in a week.

CLEAN CODE: SAVING TIME BY SPENDING TIME

The example demonstrates a paradox: in order to save time, the developers have created a situation where they have made their own future work more time-consuming. In addition, some of them probably know that there are a couple of quick-and-dirty solutions hidden deep in the code that could blow up in their face.

Recognize the situation? Then perhaps you should ask yourself the following questions:

1. Are you convinced that "make haste slowly" (in latin: festina lente) or "more haste, less speed" are sound principles to follow?
2. Do you want to maximise your freedom of action during the different phases of your development projects?
3. Do you want to feel confident that your code doesn't start failing in completely unrelated areas when you make changes to it?

Well, of course! It's time to wave goodbye to entangled code, sloppy formatting and the frustrating hunt for hidden bugs! If you want to a have a code base that is **maintainable** and **easy to adapt to requirement changes** you should definitely look into Clean Code!

WHAT EXACTLY IS CLEAN CODE?

According to Wiktionary, clean code is defined as *"software code that is formatted correctly and in an organized manner so that another coder can easily read or modify it".*

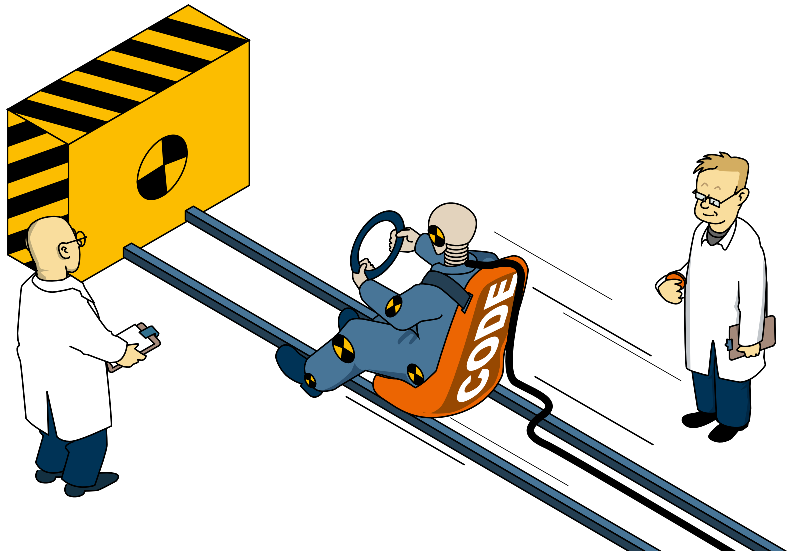
We would like to stretch this definition a little further. In our eyes, Clean Code is **S.W.A.T.**, meaning:

- **Simple, yet effective**
  Clean Code solves the problem at hand without introducing unnecessary complexity.

- **Well tended**
  Clean Code is like a well-kept garden – time and effort is spent continuously tending it and improving it. This does not happen by itself!

- **Adaptable**
  Clean Code is easy for colleagues to understand, update and modify. In addition, it can be adapted when requirements change.

- **Testable**
  Clean Code is designed to be testable. In order to fulfil the requirements above, we need to be covered by tests that can be run automatically. Using tests that can be run as we make changes, we immediately get feedback on whether or not we have broken something. By designing our code in this way we not only make it easier to write tests; we also get a more modular design as a result.



EMBRACING THE BADEN-POWELL RULE

When Robert Baden-Powell, the founder of the Scout Movement, passed away in 1941 a famous letter was found in his drawer. It was addressed to the scouts of the world and contained a line which is now classic:

Try and leave this world a little better than you found it.

This can be viewed both as a code of conduct and an ethical statement. In the Scout Movement, this rule has been applied in many ways and on many levels. A familiar hands-on rule among scouts is "Leave the campground cleaner than you found it!"

The Baden-Powell appeal can also serve as a guideline for developers – just "Leave the code cleaner than you found it!" This means that when we make changes to our code, we should also make some improvements in the surrounding parts. Thus we make the code a little better each time we modify it, instead of letting it degenerate while we are making changes and bug fixes. In addition, if we are improving our code continuously, we conserve its flexibility to change.

Some core values

Core values for Clean Code developers:

- We focus on business value.
- We value refactoring above large upfront designs.
- We value simplicity above complexity.
- We take responsibility for the project as a whole, not only our part.
- We are professionals and are proud of what we are doing.

CODE CARE

To keep your software clean and in good shape, it needs constant refactoring. This is about making continuous improvements to your code all the time (see the Boy Scout Rule!). Avoid having to carry out large refactorings (which is normally a sign of big technical debt). Frequent small refactorings are preferable. Your set of automated tests is your safety net, which allows you to change your code knowing you haven't introduced new defects.

If you practice Test-Driven Development and specify how you'd like to use the new piece of code before you actually implement it, you have a valuable tool to aid you in keeping your design clean. So if you see Clean Code as the end result, Test-Driven Development is the means to take you there.

Test-Driven Development is designed to make developers take more responsibility for the quality of the code.

Applying refactoring without any covering test cases tends to be risky since you won't have instant feedback on whether the code still works as intended. So to make sure that your code does the right thing today and tomorrow, we need to measure its health with tests.

TDD

Test-driven development (TDD) is a process for developing software using a very short development cycle: you begin by writing a test that specifies what new functionality you want your code to exhibit. Since that functionality is not in place yet, this test will fail. You then proceed to write the code needed to make that test pass. Finally, with the functionality in place and the test as verification, you refactor your code to make it more maintainable.

Remember, code is your house, and you have to live in it.

Michael Feathers, one of the leading evangelists of Clean Code

THEN WHY ISN'T EVERYONE DOING IT?

If working to attain clean code is so beneficial, then why don't we see clean code everywhere? There are probably a lot of reasons why we have a hard time getting started. For one, you're probably in a situation where the code base you're working on is far from perfect. It's easy to start feeling overwhelmed and not knowing where to start. Cleaning up your code is hardest in the beginning, and the first steps will probably not look that impressive.

It's probably quite usual for us to work in an environment where we aren't encouraged to spend time cleaning code. If you've completed a task faster than initially estimated, chances are that you'll be praised and no one will ask you if you took any shortcuts to get there. If you report that you spent some time refactoring a piece of code and putting it under test, you'll probably find that you need to defend your actions as you'll get asked if that was really necessary. It's easier to give up and go with the behavior that is rewarded.

The Broken Windows Theory

A very interesting analogy from the social sciences states that an abandoned building with a few broken windows is more likely to be vandalised if the windows aren't repaired (Wilson & Kelling 1982). The authors suggest that a successful strategy is to fix problems as soon as possible. Obviously, this applies to code as well ...

A CODE EXAMPLE

Consider the following function written in Java. It’s probably not that bad compared to some of the code you’ve seen in the wild, but it can be improved.

Does too many things
The function is called setupCall(), yet most of what it does is to parse and validate the phone number passed by the caller. This leaves the function brittle when we later might have to change it. A change to the way phone numbers are parsed runs the risk of introducing a bug in the way calls are made. It’s also not clear where to find the code to validate phone numbers if you later need to change it. A function called setUpCall() is probably not the first place you’ll look.

Different levels of abstraction mixed together
It’s unnecessarily hard to understand what is actually done by the method when giving it a quick read. This is because you have to deduce what steps are performed by looking at the details of how they are performed.

Concepts described with comments instead of classes or methods
Though the comments help us a little to understand what the code does, we shouldn’t use comments to describe something that is better described with code.

Caller passes primitives instead of value objects
By having the caller pass a primitive, such as a string, we cannot make any assumptions about its contents even if it’s named phoneNo. The caller might accidentally pass us a street address or an age instead. As a result, we need to add a number of validations before we can proceed with what we really want to do: set up a call. If we pass a string to every method that wants to do something with a phone number, all this validation needs to be duplicated in each one.

```
public void setupCall(String phoneNo) {
  // Validate that number has correct length
  if ( phoneNo.length() < 10 || phoneNo.length() > 18 ) {
    throw new CallException(“Phone number must be between 10-18 digits long!”);
  }
  // Validate that number only has digits
  if ( ! Pattern.matches(“^\\d*$”, phoneNo) ) {
  throw new CallException(“Phone number is not a digit!”);
  }

  String countryCode = null; String areaCode;
  if ( phoneNo.startsWith(“00”) ) { // International number
    countryCode = phoneNo.substring(2, 2);
    areaCode = phoneNo.substring(3,3);
  } else {
    areaCode = phoneNo.substring(0, 3);
  }

  Exchange exchange = findExchange(countryCode, areaCode);
  exchange.dial(phoneNo);
}
```

A BETTER WAY OF WRITING

Introducing a value object
The setUpCall() function now takes a PhoneNumber as a parameter instead of a string. This means that it can trust that this parameter is valid. By putting the logic associated with phone numbers in a well-named class, we no longer have to perform validation in all the places in the code that really just want to use a phone number without dealing with the internals.

Have the method do one thing
With all the phone number specific code out of the way, the function now does only one thing; it sets up a call. This way we eliminate the risk of us accidentally breaking how calls are made if we need to change the internals of how a phone number is parsed.

Replaced comments with methods
When we moved the validation logic to the PhoneNumber class, we also took care to create one method for each of the steps we need to perform for our validation. This removes the need for unnecessary comments that run the risk of becoming outdated and makes the validation method much easier to read; it only describes what steps are needed without exposing the details of how each step is accomplished.

```
public void setupCall(PhoneNumber number) {
  Exchange exchange = findExchange(number);
  exchange.dial(number);
}

public class PhoneNumber {
   private String number;
   public PhoneNumber(String number) throws InvalidNumber {
      validate(number);
      this.number = number;
   }

   private validate(string number) {
      hasCorrectLength(number);
      containsOnlyDigits(number);
   }

   /* … */
}
```

TEN DAILY HABITS

To get started you might try to make the following practices part of your daily habits.

1 Let your code speak for itself

Names in your code should be self-explanatory. Let your code speak for itself. Try to use the common domain language in your code, instead of inventing your own names. If the terms used by your users or domain experts aren't clearly represented in the code, you add the burden of having to translate from one way of describing the system to another. This increases the risk of misunderstandings and unexpected behavior of the application.

2 Remove unnecessary or misleading comments

When your code starts to speak for itself, you can remove comments that can be misleading. It is better to refactor your code to make it more readable instead of having comments that explain what the code does.

3 Let your classes be one trick ponies

Let your classes do just one thing as well. Let your class have one responsibility and one reason for change. This is commonly known as the Single Responsibility Principle (SRP).

4 And the same goes for functions and methods

Try to have your functions and methods do just one thing and be as small as possible. Let them either answer something or do something.

5 Avoid superfluous input arguments

Try to avoid using more than 3 input arguments to functions and methods. Also avoid Booleans if possible, because they decrease the readability and usability of your code.

6 Avoid returning null & error codes

Avoid returning null and error codes in your functions and methods, because they often hide bugs. To return errors, it is better to raise typed exceptions instead. Instead of returning null, you could return empty lists so that the caller code does not break.

7 Use value objects

A good OO practice is to use value objects for carrying values through your architecture stack. Value objects are immutable classes that concentrate value logic to one place. Examples of typical values to use value objects for: phone number, address, social security number.

8 Format your code for readability

Make your code more readable by formatting it well. Concentrate expressions that belong together and separate those that do not.

9 Don't repeat yourself

Avoid repeating the same piece of logic several times in your code. Otherwise you'll run the risk of having the same concept working differently in different parts of your system. If, for instance, your e-mail validation logic isn't described in one place only, you might end up having the application accept different addresses depending on what page the user enters them.

10 Keep your code as simple as possible

One key aspect of Clean Code is to make your code easy to read and understand. To make it easy to read it needs to be as simple as possible, which is one of greatest challenges of writing clean code. Simple code is easy to maintain and understand. If you can easily understand the code, you can modify it without fear. Even if you have a safety net of tests, you must anyway understand the code and its purpose before you dare to change it.

Software Craftsmanship

Clean Code can be seen as part of the Software Craftsmanship movement, which aims to promote professionality in the field of software development. The focus of the movement is described in the Manifesto for Software Craftsmanship:

"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."

Read more at **http://manifesto.softwarecraftsmanship.org**

REFERENCES

To inspire you on your journey to be a better craftsman, the following books are recommended:

- **Clean Code: A Handbook of Agile Software Craftsmanship**
  by Robert C Martin
  - A comprehensive guide on how to write clean code. Probably the most complete guide on this topic.

- **Clean Coder: A Code of Conduct for Professional Programmers**
  by Robert C Martin
  - An inspiring book about software professionalism, which describes how to behave as a software craftsman. It covers everything from how to handle pressure to how to practice your craft.

- **The Pragmatic Programmer: From Journeyman to Master**
  by Andrew Hunt, David Thomas
  - A classic book about software craftsmanship and the practice of programming. Contains a lot of useful tips to improve your skills and teach you the attitude of a craftsman.

- **Working Effectively with Legacy Code**
  by Michael Feather
  - A great book about how to begin cleaning up your code when you start out with what looks like an unmanageable code base.

- **Refactoring: Improving the Design of Existing Code**
  by Martin Fowler
  - The canonical book on how you can improve the internal structure of your code using small, well defined steps.

What is Clean Code? The short answer is: ”What a skilled, experienced and responsible developer produces.” The core of the Clean Code philosophy is to work at a slow but steady pace and carefully craft code that is

• simple, yet effective
• well-tended
• adaptable
• testable

Teams that make haste slowly can not only speed up the development process considerably, they can also make it more predictable.

Softhouse Consulting

Stockholm
Tegnérgatan 37
SE-111 61 Stockholm
Phone: +46 8 410 929 50
info.stockholm@softhouse.se

Göteborg
Lilla Bommen 1
SE-411 04 Göteborg
Phone: +46 31 760 99 00
info.goteborg@softhouse.se

Malmö
Stormgatan 14
SE-211 20 Malmö
Phone: +46 40 664 39 00
info.malmo@softhouse.se

Karlskrona
Campus Gräsvik 3A
SE-371 75 Karlskrona
Phone: +46 455 61 87 00
info.karlskrona@softhouse.se

www.softhouse.se

SOFTHOUSE