**Dirty code**

Dirty code is result of inexperience multiplied by tight deadlines, mismanagement, and nasty shortcuts taken during the development process.

**Clean code**

The main purpose of refactoring is to fight technical debt. It transforms a mess into clean code and simple design.

Nice! But what is clean code, anyway? Here are some of its features:

**Clean code is obvious for other programmers.**

And I am not talking about super sophisticated algorithms. Poor variable naming, bloated classes and methods, magic numbers -you name it- all of that makes code sloppy and difficult to grasp.

**Clean code does not contain duplication.**

Each time you have to make a change in a duplicate code, you have to remember to make the same change to every instance. This increases the cognitive load and slows down the progress.

**Clean code contains a minimal number of classes and other moving parts.**

Less code is less stuff to keep in your head. Less code is less maintenance. Less code is fewer bugs. Code is liability, keep it short and simple.

**Clean code passes all tests.**

You know your code is dirty when only 95% of your tests passed. You know you are screwed when your test coverage is 0%.

Clean code is easier and cheaper to maintain!

**Technical debt**

Everyone does their best to write excellent code from scratch. There probably is not a programmer out there who intentionally writes unclean code to the detriment of the project. Well, I wish it was true, but unfortunately it is not. But at what point does clean code become unclean?

The metaphor of “technical debt” in regards to unclean code was originally suggested by Ward Cunningham.

If you get a loan from a bank, this allows you to make purchases faster. You pay extra for expediting the process - you do not just pay off the principal, but also the additional interest on the loan. Needless to say, you can even rack up so much interest that the amount of interest exceeds your total income, making full repayment impossible.

The same thing can happen with code. You can temporarily speed up without writing tests for new features, but this will gradually slow your progress every day until you eventually pay off the debt by writing tests.

**Causes of technical debt**

Business pressure

Sometimes business circumstances might force you to roll out features before they are completely finished. In this case, patches and kludges will appear in the code to hide the unfinished parts of the project.

Lack of understanding of the consequences of technical debt

Sometimes your employer might not understand that technical debt has “interest” insofar as it slows down the pace of development as debt accumulates. This can make it too difficult to dedicate the team’s time to refactoring because management does not see the value of it.

**Failing to combat the strict coherence of components**

This is when the project resembles a monolith rather than the product of individual modules. In this case, any changes to one part of the project will affect others. Team development is made more difficult because it is difficult to isolate the work of individual members.

**Lack of tests**

The lack of immediate feedback encourages quick, but risky workarounds or kludges. In worst cases, these changes are implemented and deployed right into the production without any prior testing. The consequences can be catastrophic. For example, an innocent-looking hotfix might send a weird test email to thousands of customers or even worse, flush or corrupt an entire database.

**Lack of documentation**

This slows down the introduction of new people to the project and can grind development to a halt if key people leave the project.

**Lack of interaction between team members**

If the knowledge base is not distributed throughout the company, people will end up working with an outdated understanding of processes and information about the project. This situation can be exacerbated when junior developers are incorrectly trained by their mentors.

**Long-term simultaneous development in several branches**

This can lead to the accumulation of technical debt, which is then increased when changes are merged. The more changes made in isolation, the greater the total technical debt.

**Delayed refactoring**

The project’s requirements are constantly changing and at some point, it may become obvious that parts of the code are obsolete, have become cumbersome, and must be redesigned to meet new requirements.

On the other hand, the project’s programmers are writing new code every day that works with the obsolete parts. Therefore, the longer refactoring is delayed, the more dependent code will have to be reworked in the future.

**Lack of compliance monitoring**

This happens when everyone working on the project writes code as they see fit (i.e. the same way they wrote the last project).

**Incompetence**

This is when the developer just does not know how to write decent code.

**When to refactor**

**Rule of Three**

1. When you are doing something for the first time, just get it done.

2. When you are doing something similar for the second time, cringe at having to repeat but do the same thing anyway.

3. When you are doing something for the third time, start refactoring.

**When adding a feature**

Refactoring helps you understand other people’s code. If you have to deal with someone else’s dirty code, try to refactor it first. Clean code is much easier to grasp. You will improve it not only for yourself, but also for those who use it after you.

Refactoring makes it easier to add new features. It is much easier to make changes in clean code.

**When fixing a bug**

Bugs in code behave just like those in real life: they live in the darkest, dirtiest places in the code. Clean your code and the errors will practically discover themselves.

Managers appreciate proactive refactoring as it eliminates the need for special refactoring tasks later. Happy bosses make happy programmers!

**During a code review**

The code review may be the last chance to tidy up the code before it becomes available to the public.

It is best to perform such reviews in a pair with an author. This way you could fix simple problems quickly and gauge the time for fixing the more difficult ones.

**How to refactor**

Refactoring should be done as a series of small changes, each of which makes the existing code slightly better while still leaving the program in working order.

**Checklist of refactoring done right way**

The code should become cleaner.

If the code remains just as unclean after refactoring... well, I am sorry, but you have just wasted your time. Try to figure out why this happened.

It frequently happens when you move away from refactoring with small changes and mix a whole bunch of refactorings into one big change. So it is very easy to lose your mind, especially if you have a time limit.

But it can also happen when working with extremely sloppy code. Whatever you improve, the code as a whole remains a disaster.

In this case, it is worthwhile to think about completely rewriting parts of the code. But before that, you should have written tests and set aside a good chunk of time. Otherwise, you will end up with the kinds of results we talked about in the first paragraph.

**New functionality should not be created during refactoring.**

Do not mix refactoring and direct development of new features. Try to separate these processes at least within the confines of individual commits.

All existing tests must pass after refactoring.

There are two cases when tests can break down after refactoring:

You made an error during refactoring. This one is a no-brainer: go ahead and fix the error.

Your tests were too low-level. For example, you were testing private methods of classes.

In this case, the tests are to blame. You can either refactor the tests themselves or write an entirely new set of higher-level tests. A great way to avoid this kind of a situation is to write BDD-style tests.

**Code Smells**

**Bloaters**

Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

**Long Method**

* A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.

**Large Class**

* A class contains many fields/methods/lines of code.

**Primitive Obsession**

* Use of primitives instead of small objects for simple tasks (such as currency, ranges, special strings for phone numbers, etc.)
* Use of constants for coding information (such as a constant USER\_ADMIN\_ROLE = 1 for referring to users with administrator rights.)
* Use of string constants as field names for use in data arrays.

**Long Parameter List**

* More than three or four parameters for a method.

**Data Clumps**

* Sometimes different parts of the code contain identical groups of variables (such as parameters for connecting to a database). These clumps should be turned into their own classes.

**Object-Orientation Abusers**

All these smells are incomplete or incorrect application of object-oriented programming principles.

**Switch Statements**

* You have a complex switch operator or sequence of if statements.

**Temporary Field**

* Temporary fields get their values (and thus are needed by objects) only under certain circumstances. Outside of these circumstances, they are empty.

**Refused Bequest**

* If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter (away from the usual, expected, or correct condition or state). The unneeded methods may simply go unused or be redefined and give off exceptions.

**Alternative Classes with Different Interfaces**

* Two classes perform identical functions but have different method names.

**Change Preventers**

These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more complicated and expensive as a result.

**Divergent Change**

* You find yourself having to change many unrelated methods when you make changes to a class. For example, when adding a new product type you have to change the methods for finding, displaying, and ordering products.

**Shotgun Surgery**

* Making any modifications requires that you make many small changes to many different classes.

**Parallel Inheritance Hierarchies**

* Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class.

**Dispensables**

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

**Comments**

* A method is filled with explanatory comments.

**Duplicate Code**

* Two code fragments look almost identical.

**Lazy Class**

* Understanding and maintaining classes always costs time and money. So, if a class does not do enough to earn your attention, it should be deleted.

**Data Class**

* A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. These classes do not contain any additional functionality and cannot independently operate on the data that they own.

**Dead Code**

* A variable, parameter, field, method or class is no longer used (usually because it is obsolete).

**Speculative Generality**

* There is an unused class, method, field or parameter.

**Couplers**

All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

**Feature Envy**

* A method accesses the data of another object more than its own data.

**Inappropriate Intimacy**

* One class uses the internal fields and methods of another class.

**Message Chains**

* In code you see a series of calls resembling $a->b()->c()->d()

**Middle Man**

* If a class performs only one action, delegating work to another class, why does it exist at all?