Title: **Good and bad programming practices**

Author: **João Miguel Abrantes Dias**

Date: **20/05/2020**

INDEX

1. Introductory Note

The objective of this essay is to explore other concepts that were not lectured in the PDS class. The class mainly focused on design patterns, but there are a lot of good practices that were not included in the curricular plan. With that being said, a lot of bad practices are made every day by people who are new to programming. Some of those mistakes are common, so, we are going to go into detail on some of them further ahead. So as to avoid these mistakes, we have refactoring, which is a very helpful technique that we are going to take a look at as well.

2. Code Smells

We are going to start by analyzing the bad practices, also known as **code smells**. In the computer science field, a code smell is any type of characteristic in the code of a program that indicates that a deeper problem exists. They usually indicate some sort of violation of fundamental design principles and negatively impact the design quality. There are several types of **code smells**, but they usually are inside one of the following categories.

2.1 Bloaters

Code, methods and/or classes that have increased in size to the point where it is very hard to work with. Usually these go unnoticed for quite some time and the accumulation starts to occur over time. This accumulation can occur in method where we keep adding features to it until it gets too complex; it can occur in its own parameter list, as we saw in "*TopicA"* about how, by using the Builder pattern, we can avoid that in our constructors. Just like in the methods, it can happen in the class itself where we add a complex feature instead of just creating a separate class for it. It can also happen when creating a lot of primitive fields instead of creating a new class.

Solution: The best solution for this is to implement the Extract Method/Class/Interface. This way we can simplify each component, making it easier to read, implement and avoid complexity.

2.2 Object-Orientation Abusers

Characterized by the misusing object-oriented programming principles. Let us see some examples: The programmer created a class which function was already able to be fulfilled by another one. The subclass uses little to no methods inherited from its parents.

Solution: The use of complex switch operators or a long sequence of if statements usually means that a class using the Extract method could have been created instead. The same solution can also be applied to temporary fields.

2.3 Change Preventers

The program classes are not vey decoupled, so a change in one place of the code means that changes need to be made in other parts of the code too. For example: having to change unrelated methods of a class, just because one of the methods was changed. Making any sort of modification will lead to the need to make very small changes to different classes. When a subclass is created, another subclass needs to be created for a different class as well.

Solution: These problems mostly originate from poor programming skills. The best solution for these is to simple increase decoupling between classes by splitting up their behavior using Extract class, and moving methods and fields between classes, avoiding classes being attached to methods that are barely used by them.

2.4 Dispensables

These are parts of the code that are considered useless. They do not add any meaningful value to the program itself, and if deleted, the code would look cleaner. These can range from any sort of chunks of code that look the same to unused classes, methods, or fields to a data class that functions as a simple container for data and cannot operate as a standalone class on the data they own.

Solution: All of these can be avoided by always making sure your code is refactored, to make sure all sorts of unused and duplicated code is eliminated. We will dive deeper into how refactoring works further in this essay.

2.5 Couplers

These smells are characterized by contributing excessively to the increase of coupling between classes. They can be found in classes that delegate almost all their work to other classes, rendering it useless to exist at all, when a method accesses more data from another object than its own, or when a class makes use of the internal fields and methods of another class.

Solution: Implementing the Move method. This means that if a method from class B is more used in class A than in class B, we can move it to class A and replace the content of the B method to a reference to the new method created. In cases where a middleman exists, removing it can be the best option.

3. Refactoring

We hear people talking about refactoring all the time but, what is code refactoring? In simple words, **code refactoring** is the process of restructuring already existing and working code without changing its external behavior. Refactoring is intended to improve the structure, design, and implementation of the software without changing its functionality. Agile teams maintain and extend their code a lot from iteration to iteration, and, if it were not for the continuous refactoring they do, it would be impossible, after a couple of iterations, to continue to extend that code. This is because, during development and iterations, classes and method start to gain unwanted **dependencies**, start to grow bigger and some get left out, so it's inevitable that after a while the code becomes impossible to work on, especially if a lot of different people were working on it. The best way to look at it is to compare our program with a kitchen. If every time we cooked we didn't clean and store our used ingredients, after a couple of meals cooked, it would be impossible to cook in that kitchen, ingredients would be spread everywhere, some would have even been mixed with others, and even to cook the simplest of meals it would require a lot of work just to find the specific ingredients we needed. Add more people cooking in that kitchen and it would be total chaos. Our ingredients and tools are like our classes and methods, if we do not clean them up and store them like we should, it is very hard to work with them in the future.

So, when should we refactor? The best time to do it is right before adding any sort of update or new feature to our existing code.

3.1 How to refactor?

The best way to do it is in small steps and, like referenced above, before adding any sort of new functionality or feature to the code. There are a lot of techniques used for refactoring, let us look at some of the more popular ones.

3.2 Red-Green-Refactor

The **Red-Green-Refactor** is probably the most famous refactoring technique in agile development. Characterized by a "test-first" approach to design and implementation, it builds the foundation for all forms of refactoring. Applying this method breaks refactoring into three different steps:

Red Section – Where we consider what needs to be done and we make the tests for that functionality.

Green Section – The code developed was enough to pass the test design to test the new functionalities.

Refactor Section – Where the code optimization and restructuring occur without adding new functionalities.

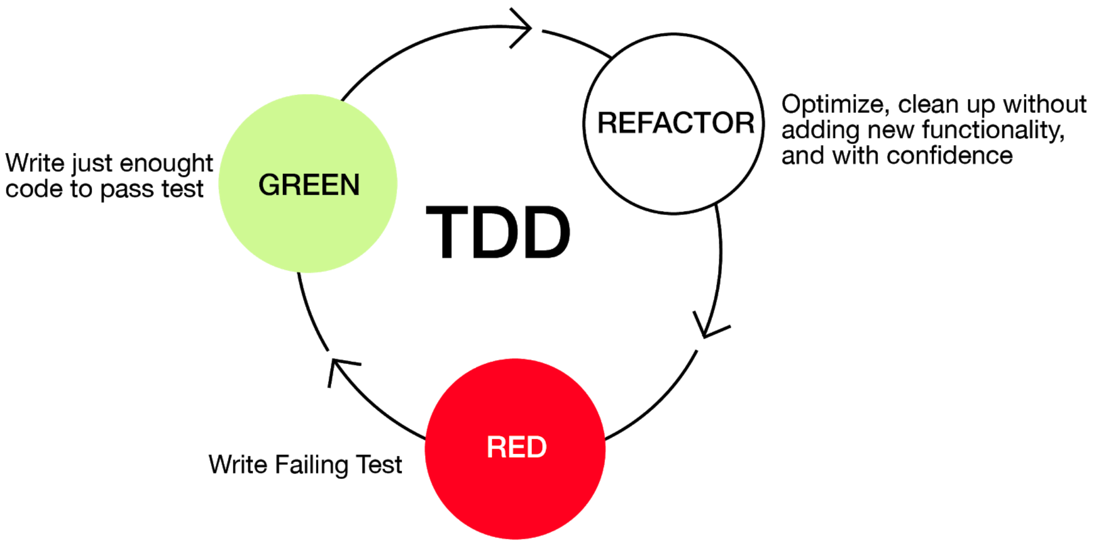
REFACTOR
Optimize, clean up without adding new functionality, and with confidence
Write just enought code to pass test
GREEN
TDD
RED
Write Failing Test

Figure 1 - Red-Green Refactoring

3.3 *Branching by Abstraction Refactoring*

This method is used when there is a large amount of refactoring to be made. The idea is to build an abstraction layer that wraps the part of the system that is going to be refactored and the counterpart that is going to replace it. One example is the **Pull-UP/Push-Down method**. The Pull-Up method pulls the code into a superclass in order to eliminate code duplication. Push-Down takes it from a superclass and moves it down into subclasses.

3.4 *Composing Method*

This method involves streamlining the code in order to avoid duplicate code. It's done through a lot of processes, including extraction, where we break code into smaller chunks, in order to find the extract fragmentation that is then moved to a new method and replaced with a call to this new method; and inline, where we find all calls of the method and replace them with the content of the method.

3.5 *Moving Features Between Objects*

This method shows how to safely create classes and move functionality between them and hides the details from public access. The Move Methods referenced in the code smells solutions are often achieved using this type of refactoring.

3.6 *User Interface Refactoring*

Simple interface changes that do not affect functionality but improve user experience, such as increasing color contrast, font size, buttons positioning, etc.

4. Final thoughts

Refactoring is a sharp weapon for developers, and it’s an essential tool for new programmers to learn how to use, in order to to avoid code smells like the ones referenced above. Refactoring improves the design of the software, makes it easier to understand, helps finding bugs and helps programming faster. Despite all of this, long and deep refactoring can be a bad thing when deadlines are near, or company resources are scarce, so we should always have that in mind.

5. References

https://code.tutsplus.com/tutorials/top-15-best-practices-for-writing-super-readable-code--net-8118
https://apiumhub.com/tech-blog-barcelona/code-smells/
https://code.tutsplus.com/tutorials/top-15-best-practices-for-writing-super-readable-code--net-8118
https://www.altexsoft.com/blog/engineering/code-refactoring-best-practices-when-and-when-not-to-do-it/
https://www.cio.com/article/2448952/10-bad-coding-practices-that-wreck-software-development-projects.html
https://refactoring.guru/refactoring
https://sourcemaking.com/refactoring/smells