TP6

BERTRAND Maxime

Exercice 1 :

1. What is clean code? What is the role played by refactoring one's code?

- Clean code is obvious for other programmers.
- Clean code doesn't contain duplication.
- Clean code contains a minimal number of classes and other moving parts.
- Clean code passes all tests.
- Clean code is easier and cheaper to maintain.

Refactoring is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design.

2. Do you think you can "over-refactor" and do too much? How?

It's possible because it takes time for those who review/approve the code and it may result in manual regression testing.

3. What is a code smell? Why should you bother?

Code smells are not bugs or errors. Instead, these are absolute violations of the fundamentals of developing software that decrease the quality of code. We should bother because having code smells does not certainly mean that the software won't work, it would still give an output, but it may slow down processing, increased risk of failure and errors while making the program vulnerable to bugs in the future. Smelly code contributes to poor code quality and hence increasing the technical debt.

4. Can you identify a few code smells in the original GuildedRose?

For instance, the set of “items” could be improved in order to be more readable cause in the original GuildedRose, it’s complicated and not instinctive when we look at this part of the code.

5. What are some refactoring techniques you could have used in the GuildedRose?

We could do some changes as:

- The original had no tests. Since this is a refactoring kata, I feel the tests are important and provide a complete test suite.
- The original used a hard coded set of "items", presumably for testing the code. If we add a test suite, the hard coded values are not of much use. We should also change the interface to accept a list of items as a parameter rather than a hard coded constant.

Exercice 2:

1. In simple terms, and a few sentences at most, what’s a design pattern?

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

   a. When should you use one?

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architect’s familiar with the patterns.

   b. When shouldn’t you?

Where Design Patterns fail is when the team doesn't understand them, or when they are overused so much that they stop making the design clearer and instead make it harder to understand what is really going on.

2. Write a small program that build a pizza with various toppings using the builder pattern. Why is this a good idea? Think about scalability for instance

The definition of builder pattern is that it separates object construction from its representation. I would go with the Builder pattern because the task is building a pizza. The Builder pattern is used to build something which afterword exists as is (and which should not be altered at any time).

3. Find an original idea to implement a decorator pattern (not the one from the website). Can you think of any limitations? No need to code here.

The decorator pattern would be used for example if we have a class which represents a rectangle. The Rectangle class has only one method which calculates the perimeter. And you want to add a method to the Rectangle to calculate the area. The problem is that you cannot edit the class Rectangle so you have to create a new class.