Théotime Soulier-Verschaeve

TP#6: February 09 th – Design Patterns, Refactoring, Code Smells

Exercise 1 :

1.

A clean code is a code that is easy to read, to maintain and to understand. It increases the quality of the code and of the product.
The code must be readable by every programmer with meaningful variables names, functions, etc.
The code must not contain to much duplication to avoid waste of time when a change is necessary. It is also better to reduce the number of class and pass a maximum of tests.

2.

Over factoring is possible and the problem could for example be to reduce to much the number of classes and to not have enough classes to design correctly the project. A problem could also be to separate the project in too much files even when it is not necessary: it is not always a problem if a file is heavy if all the classes and functions are linked.

3.

Code smells are a list of bad codding habits such as the use of bloater, abused object-orientation, change preventers, dispensables or couplers.
It is important to bother in order to ensure maintainability and to allow the possibility to reuse and improve the code easily. This allows to have not too big functions that are difficult to read and to improve.

4.

Long methods: update quality is too long and treats all the items and cases at the same time. It is then difficult to read it and to improve it.
Switch statement: The function updateQuality is a long list of if statement making the code unreadable and difficult to improve and change.
Incomplete library class: Item class was not enough convenient to answer the problem. First the item class could have been improved to carry conjured items. Moreover, other classes could have been created to carry all the types of items.

5.

To overcome the long method smell, the good practice would have been to decompose the long updateQuality function into several smaller functions that carry only one kind of object and have a precise objective. This allows to make the code more readable and to reuse the function somewhere else.
The list of if statement could have been replaced by a clean switch case depending on the item and then each case could have called a function treating the quality update depending on the item.
The incomplete library class could have been improved by adding a class for each kind of item.

Théotime Soulier-Verschaeve

Exercise 2 :

1.
Design patterns are typical solutions to commonly occurring problems in software design. It is a general concept for solving particular problems. Pattern are different from algorithms because they don't define a clear set of actions that achieve the same goal. The use of the pattern can be different depending on the problem to solve.

1.a.
Design pattern should be used when the program is facing a common problem that can be solved by many different means, for example to create different objects with the same base or the same goal but with different characteristics and behaviour.

1.b.
The design pattern is not needed when the problem can be solved with algorithms with a precise goal.

2;
This is a good idea in this case because a pizza is a very customizable object and building design patterns are perfect for this. The main advantage is the possibility to add or remove easily topping options for the pizza. Changing each functions of the builder won't have a consequent impact on the whole project.

3.
Decorator design pattern can be used to :
As example, we could take the one of a car with many possible option. We could have a basic car with only the necessary things to make it work but we could also decide to add xenon headlights, a spoiler or a nitro.
To do so we would have implement:

- A ***Car*** interface:

This interface could have a signature of ***getCarDescription()*** returning a string.

- A ***ConcreteCar*** class implementing the ***Car*** interface.

This class would have a string *_descripition* as attribute and a method *getCarDescription()* returning a string like "Car".

- A ***OptionsDecorator*** abstract class implementing ***Car*** interface.

This abstract class would have an object of type Car from Car interface as attribute called *_car.*
It would contain a constructor initializing the attribute.
It would also contain a method *getCarDescription()* that return *_car.getCarDescription().*

- A class for each option: ***Spoiler, XenonHealights*** and ***Nitro*** implementing the ***OptionsDecorator*** class:

For each class, a constructor is needed taking a new Car *newCar* as argument and using base(*newCar*) to construct the mother class.
Theses class would also contain a method *getCarDescription()* returning *_car.getCarDescription()+"with spoiler"* for the spoiler class.

It is then possible to create a car with different options:
Car newCar = new Spoiler(new Nitro(new ConcreteCar()));

Because Spoiler, XenonHeadlights,Nitro classes implement OptionsDecorator class that implements Car interface, it is possible to call the constructors in each other's.

4.

Iterator behavioural design pattern for a library.