**Software Architectures**
**Lab. 1. Homework**

***Learn yourself the following design patterns: Factory Method, Builder, Decorator***

## Generalities about design patterns

**Def1**: Reusable solutions to commonly occurring problems.
**Def2**: OO design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.
Each pattern has:

- name
- problem – when to apply the problem
- solution – elements that make up the design, relationships, responsibilities and collaborations
- consequences – results and trade-offs of applying the pattern

**Reference:**
The course lectures: http://www.serbanati.com/poli/Lecture_Notes/ARCH/ARCH_Chap1.pdf
https://www.oodesign.com/
https://medium.com/@ronnieschaniel/object-oriented-design-patterns-explained-using-practical-examples-84807445b092
https://medium.com/@adeshsrivastava0/decorator-design-patterns-3d6ed6d5aba9
Please, read descriptions for Factory Method, Builder, and Decorator from references.

**Factory Method**

Factory Method Pattern says that just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate. Use a Factory Design Pattern :

- When a class doesn't know what sub-classes will be required to create
- When a class wants that its sub-classes specify the objects to be created.
- When the parent classes choose the creation of objects to its sub-classes.

Advantages:

- Factory Method Pattern allows the sub-classes to choose the type of objects to create.
- It promotes the loose-coupling by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

**Builder**

In general, we keep the object construction code such as instantiation and initialization within the object as a part of its constructor. This approach leads to the high coupling. It is suitable for small systems with lower object dependencies and lesser object representations or sub-classes.
When the roles and responsibilities get heavier for one object, the creation of that object becomes complex and messy. The object itself may get bulky, and low in modularity. At the same time object might need to take different forms to fulfill client requirements. Hence, it needs additional input parameters to get the required results. As a result, the number of constructor parameters increases. This may lead to the problem of 'Telescoping Constructor' pattern. It is an anti-pattern where the complexity caused due to the increased number of constructor parameters. Higher parameters lead problems like difficulty in selecting, remembering the correct order, and purpose of each parameter. In addition, it creates problems in selecting the suitable constructor for the required scenario.The Builder pattern avoids such situations.
Builder pattern is a creational design pattern. The builder pattern suggests an approach to remove the construction process from the object and hand over that process to a separate class. That separate class is 'Builder'. Custom implementations can have more than one builder class to create different representations.
The main intention of the builder pattern is to create complex objects by separating the construction process from its representation so that the same construction process can create different representations. This pattern enables the polymorphism by providing an effective solution for creating different forms of an object while using a consistent construction process: the process of constructing a complex object should be generic. That is different forms can use the same flow of steps to create the objects. In this case, a separate entity that called the 'Builder' do the object creation and the client can use the 'Builder' to create an object with any preference. Hence, the object can take different forms depending on the applied input parameters.

**Decorator**

When we subclass a parent class, there might be multiple combination of classes resulting in a class explosion. Although inheritance is a foundational pillar of Object-Oriented Programming, completely relying on it for structuring classes will lead to class explosion. So, it does not seem always an elegant solution and obviously is hard to maintain if the number of subclasses exceeds. What if there was a way to reduce the number of subclasses we create for every variation of an object? That would be a lot better solution and that is what the decorator design pattern has to offer. The approach is to make objects acquire responsibilities (feature, variation, combination, additional functionality) at run time.

Decorator pattern is a structural design pattern that can be used to add additional behavior to objects at runtime. The role of the Decorator pattern is to provide a way of attaching new state and behavior to an object dynamically. The object does not know it is being "decorated," which makes this a useful pattern for evolving systems. A key implementation point in the Decorator pattern is that decorators both inherit the original class and contain an instantiation of it.

Advantages:

1. Decorator pattern solves the class explosion problem. We did not create subclasses for every type of the base class objects.
2. We can easily extend a class without modifying other classes.
3. We can wrap the components with any number of decorators .

## Exercises

**Factory Method**

**Exercise 1**

Consider we want to implement a notification service (**Notification** ) using three different notifications types: email, SMS, and push. Let model the creation of a notification object of any type using a unique **Notification** class/interface having a method *notifyUser()*.

**End of exercise 1**

**Builder**

**Exercise 2**

Consider the case of a class representing the Nutrition Facts label that appears on packaged foods. These labels have a few required fields—serving size, servings per container, and calories per serving— and more than twenty optional fields—total fat, saturated fat, trans fat, cholesterol, sodium, and so on. Most products have nonzero values for only a few of these optional fields.

Use Builder design pattern to generate such labels for different kinds of Nutrition Facts.

**End of exercise 2**

**Decorator**

**Exercise 3**

Assume that the Photo class was written with *draw()* as a plain (concrete) method that simply displays a photo with its specified dimensions and that cannot be altered. We want to use this class for printing an album where the photos are diversified by framing them into borders of various formats (as color or weight). Design the UML class diagram that models the photo adornment.

**End of exercise 3**