# Builder Pattern

Preslav Gerchev 2487136  
Dimitar Vikentiev 2464020

# Introduction

In week 6 we had to develop a car building system using the Builder Pattern.  
The following document will reflect on the project and on the pattern itself- positive and negative consequences, reusability, maintainability, extensibility.

# The Pattern

The formal definition for the pattern is as following**:**

**“Separate the construction of a complex object from its representation so that the same construction process can create different representations.”**

# Reusability

The builder pattern strongly promotes reusability. The pattern has one constructor that can be reused throughout the whole application. Its only job is to create the object which is passed. The advantage here is that several builders can be passed to the constructor – it only knows that it will receive an object of a certain interface (the builder interface). That is the separation of the representation from the construction process – the constructor class doesn’t know anything about any of the builders.

# Extensibility

The builder pattern is also easy to extend – the only thing needed to be done is to simply create a new builder class and inherit the builder’s interface. The builder also promotes consistency – the construct knows that every builder will have concrete methods (those defined in the interface) and can use them.

It is also the builder’s responsibility to provide the client with the created object after the constructor has created it.

A disadvantage is that for each new product we need a new builder class – if we have 100 different types of cars we would need 100 different car builder classes.

# Maintainability

This is perhaps the strongest point of the builder pattern – the maintainability. The main reason for that is that the code for representation (the different builders) and the construction (the constructor or sometimes people to refer to it as the “director” class) are encapsulated and separated from each other.

That means if we were to change something in one of the builders it would:

1. Not affect the constructor – the constructor doesn’t know anything about the builder.
2. Not affect other builders – each builder is separate from the rest and independent.

The only problem would arise (but that is also with almost every other pattern) if we were to change something in the builder’s interface – then we would have to change every builder’s implementation.

Changing something in the constructor would not affect any of the builders – they are not even aware of the constructor class.

# Extra

## Relation with the Abstract Factory Pattern

The builder pattern is similar to the abstract factory in some way – it also takes care of creating objects. However the main difference between those is that the abstract factory takes care of creating a family of objects and you don’t have to create all. For example if we had a vehicle factory we could create a truck, a car, a motorcycle and a normal bike.

The builder pattern takes care of creating ***one***single object – in the example given above we would need 4 builders – truck builder, car builder, motorcycle builder and bike builder. The builder simply creates the different parts of that object – this can be well seen in our example (we have different car brand builders that take care of creating doors, tyres, frame, engine and color).

# A little bit more about Builders

Over the years different types of builders have arisen. Besides the formal builder pattern that is described above and that we used to create our application another type of builders can be seen.

An example code for that would be something like this :

*Car bmwCar=new CarBuilder().withDoors(4).withColor(Color.Red).withEngine(BmwEngine).create();*

Here the client is responsible for creating the car. The builder provides the ways to do that including the create() method which will return the created and modified car.

However this does not really differentiate from a regular Car construct with several parameters – we could simply do that:

*Car bmwCar=new Car(4,Color.Red,BmwEngine);*

And of course the main advantage of the builder pattern is violated – separating creation from representation.

# Summary

The builder pattern is a creational pattern that finds its usage in applications where the creation of an object is separated from its representation. In some ways it is similar to the Abstract Factory pattern and affords finer control over the construction process.  Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the control of the constructor (or also called director).