Article

Builder Design Pattern With Java

By Aman Gupta on Updated date Oct 12, 2020

Introduction

In this series of learning different design patterns, now we came across a new pattern, i.e. the builder design pattern. As discussed in the last article, we saw what a design pattern is, and what it provides when implemented with the factory design pattern.

A design pattern provides a simple solution to common problems that are faced in day-to-day life by software developers. Three basic types of design patterns exist, which are listed below:

- Creational Pattern
- Structural Pattern
- Behavioral Pattern

Along with these, 23 design patterns are classified above in three patterns.

Thus, the pattern described here is like a blueprint that you can customize to solve a particular design problem and they only differ by complexity and scalability of the code.

In this article, we will learn about the builder design pattern. Let's get started.

Why the Builder Design Pattern?

To start with what is a builder design pattern, one should know why we need this design pattern and what's the drawback behind the factory pattern to introduce a new design pattern.

In simple language, when we have too many arguments to send in the constructor and is difficult to maintain the order of the values to be set or when we don't want to send all parameters in object initialization i.e parametric value as NULL.

Inserting too many attribute values through the constructor and maintaining the order of values respectively is a difficult task. To overcome the issue of ordering and insertion of the values, the builder design pattern has evolved.

Let's see this example:

```
1. package builderDesign;
2. public class CsharpCorner {
3.     private String name;
4.     private String role;
5.     private int reputationPoints;
6.     private int read;
7.     private int profileLikes;
8.     public Csharpcorner(String name, String role, int reputationPoints, int read, int profileLikes) {
9.         super();
10.        this.name = name;
11.        this.role = role;
12.        this.reputationPoints = reputationPoints;
13.        this.read = read;
14.        this.profileLikes = profileLikes;
15.    }
16. }
```

Now, what if we introduce new attributes in the class? One way is to create more constructor and another is to introduce setter methods and lose the immutability. By choosing either of both options, you lose something.

Here, the builder pattern will help in consuming additional attributes while retaining immutability. Thus, the builder pattern provides more control over the object creation problem.

Note
The public constructor is to include only required parameters, there's no need to add optional parameters in the constructor.

Builder Design Pattern

Builder Design Pattern is a part of a creational design pattern i.e it deals in the creation of an object and is an alternative approach for constructing complex objects.

The builder design pattern is used when want to construct different immutable objects using the same object. It provides more control over the object creation process before the learning builder pattern lets us understand why we need this design pattern.

Demo Implementation

Let's have a class Phone with 5 variables and we are also using a parameterized constructor to set the values. Along with this, we have a toString() method. When we print the object we get that method.

```
1. package BuilderDesign;
2. public class Phone {
3.     private String os;
4.     private String processor;
5.     private int battery;
6.     private double screensize;
7.     private String company;
8.     public Phone(String os, String processor, int battery, double screensize, String company) {
9.         super();
10.        this.os = os;
11.        this.processor = processor;
12.        this.battery = battery;
13.        this.screensize = screensize;
14.        this.company = company;
15.    }
16.    public String toString() {
17.        return "Phone [os=" + os + ", processor=" + processor + ", battery=" + battery + ", screensize=" + screensize + ", company=" + company + "]
18.    }
19. }
```

Now we will create PhoneBuilder class, which is responsible for creating a phone. Here, we have all member variables with their setter function. Instead of using "void" in the setter function, we will use the "PhoneBuilder" object that means for any method we set the value you will get the object of PhoneBuilder.

```
1. package BuilderDesign;
2. /* RULES
3.  * 1-> Create class and initialize variables
4.  * 2-> Create Setter functions of that variables with datatype of class name(here PhoneBuilder) and return this.
5.  * 3-> Create method similar to "toString" (here getPhone).
6.  * 4---> Create class(here Phone) like in Encapsulation(with constructor(parameterized) and toString() method).
7.  * 5---> Create a class used for testing the code(here Shop) and add details you want.
8.  */
9. public class PhoneBuilder {
10.    private String os;
11.    private String processor;
12.    private int battery;
13.    private double screensize;
14.    private String company;
15.    public PhoneBuilder setOs(String os) {
16.        this.os = os;
17.        return this;
18.    }
19.    public PhoneBuilder setProcessor(String processor) {
20.        this.processor = processor;
21.        return this;
22.    }
23.    public PhoneBuilder setBattery(int battery) {
24.        this.battery = battery;
25.        return this;
26.    }
27.    public PhoneBuilder setScreensize(double screensize) {
28.        this.screensize = screensize;
29.        return this;
30.    }
31.    public PhoneBuilder setCompany(String company) {
32.        this.company = company;
33.        return this;
34.    }
35.    public Phone getPhone() {
36.        return new Phone(os, processor, battery, screensize, company);
37.    }
38. }
```

Create the class Shop and set the values you want only & others will take the default value accordion to the datatype.

```
1. package BuilderDesign;
2. public class Shop {
3.     public static void main(String[] args) {
4.         Phone p1 = new PhoneBuilder().setCompany("Samsung").setBattery(6000).setOs("Android10").getPhone();
```

```
        System.out.println(p1);
        Phone p2 = new PhoneBuilder().setCompany("Htc").setOs("Android10").setScreensize(15.6).getPhone();
        System.out.println(p2);
        Phone p3 = new PhoneBuilder().setBattery(7000).setCompany("Samsung").setProcessor("Intel").getPhone();
        System.out.println(p3);
    }
}
```

OUTPUT

Phone [os=Android10, processor=null, battery=6000, screensize=0.0, company=Samsung]
Phone [os=Android10, processor=null, battery=0, screensize=15.6, company=Htc]
Phone [os=null, processor=Intel, battery=7000, screensize=0.0, company=Samsung]

From this above demo example, you will have an idea i.e what builder pattern is and what is the usage with the option to add only the required values and other values are default NULL.

Realtime case problem Implementation

Here is the real-time problem of any restaurant to calculate the cost of your meal. We have a few options, mentioned below:

- Interface- Item, Packing
- Abstract class- Burger, ColdDrink
- Burger- VegBurger, ChickenBurger
- Cold drink- Pepsi, Coke
- Packing class- Bottle, Wrapper
- and other classes

First, we have two interfaces, Item, and Packing:

```
package exampleBuilder;
public interface Item {
    public String name();
    public Packing packing();
    public float price();
}
```

```
package exampleBuilder;
public interface Packing {
    public String pack();
}
```

Then we have two abstract classes for types of food items that implement the interface item.

```
package exampleBuilder;
public abstract class Burger implements Item {
    @Override
    public Packing packing() {
        return new Wrapper();
    }
    @Override
    public abstract float price();
}
```

```
package exampleBuilder;
public abstract class ColdDrink implements Item {
    @Override
    public Packing packing() {
        return new Bottle();
    }
    @Override
    public abstract float price();
}
```

For home delivery of food items, we need to wrap the food. For that purpose, we need a Wrapper class for Burger and Bottle class for Colddrink, and they are defined as,
package exampleBuilder;

```
public class Bottle implements Packing {
    @Override
    public String pack() {
        return "Bottle";
    }
}
```

```
package exampleBuilder;
public class Wrapper implements Packing {
    @Override
    public String pack() {
```

```
        return "Wrapper";
    }
}
```

Then, a Burger class is divided into two different types of the burger as Veg and Non-Veg, which extends an abstract class, Burger.

```
package exampleBuilder;
public class VegBurger extends Burger {
    @Override
    public String name() {
        return "VegBurger";
    }
    @Override
    public float price() {
        return 50.25 f;
    }
}
```

```
package exampleBuilder;
public class ChickenBurger extends Burger {
    @Override
    public String name() {
        return "ChickenBurger";
    }
    @Override
    public float price() {
        return 80.50 f;
    }
}
```

Similarly, a class Colddrink is further divided into two different classes which extends an abstract class, ColdDrink.

```
package exampleBuilder;
public class Coke extends ColdDrink {
    @Override
    public String name() {
        return "Coke";
    }
    @Override
    public float price() {
        return 40.0 f;
    }
}
```

```
package exampleBuilder;
public class Pepsi extends ColdDrink {
    @Override
    public String name() {
        return "Pepsi";
    }
    @Override
    public float price() {
        return 35.0 f;
    }
}
```

After all these food items are known, we have to get a meal, shown by:

```
package exampleBuilder;
import java.util.ArrayList;
import java.util.List;
public class Meal {
    List < Item > list = new ArrayList < Item > ();
    public void addItem(Item item) {
        list.add(item);
    }
    public float getCost() {
        float cost = 0.0 f;
        for (Item item: list) {
            cost += item.price();
        }
        return cost;
    }
    public void showMeal() {
        for (Item i: list) {
            System.out.print("Item Name " + i.name());
            System.out.print("\t Item Packing " + i.packing().pack());
            System.out.println("\t Item Price " + i.price());
        }
    }
}
```

After the meal is ready, we need a MealBuilder class similar to a waiter taking your food order whether veg or non-veg meal. This is shown by:

```
1. package exampleBuilder;
2. public class MealBuilder {
3.     public Meal vegMeal() {
4.         Meal meal = new Meal();
5.         meal.addItem(new VegBurger());
6.         meal.addItem(new Coke());
7.         return meal;
8.     }
9.     public Meal nonvegMeal() {
10.        Meal meal = new Meal();
11.        meal.addItem(new ChickenBurger());
12.        meal.addItem(new Pepsi());
13.        return meal;
14.    }
15. }
```

At last, we just place our order and wait for the delicious food. Here the process is the same, except for the service time that the restaurant workers need to make your food ready. This is neglected here:

```
1. package exampleBuilder;
2. public class PlaceOrder {
3.     public static void main(String[] args) {
4.         MealBuilder mealbuilder = new MealBuilder();
5.         Meal veg = mealbuilder.vegMeal();
6.         veg.showMeal();
7.         System.out.println("Meal Cost: " + veg.getCost());
8.         Meal nonveg = mealbuilder.nonvegMeal();
9.         nonveg.showMeal();
10.        System.out.println("Meal Cost: " + nonveg.getCost());
11.    }
12. }
```

OUTPUT

```
Item Name VegBurger Item Packing Wrapper Item Price 50.25
Item Name Coke Item Packing Bottle Item Price 40.0
Meal Cost: 90.25
Item Name ChickenBurger Item Packing Wrapper Item Price 80.5
Item Name Pepsi Item Packing Bottle Item Price 35.0
Meal Cost: 115.5
```

Note
We don’t use any setter method, so its state can not be changed once it has been built thus provides desired immutability. Alternatively, if we have a setter method one can change the value from outside.

Existing Implementations in JDK

- StringBuffer
- StringBuilder, and
- ByteBuffer

Advantages

- Flexible Design.
- Readable code(easily).
- Reduces the number of a parameter in the constructor thus no need to pass NULL for optional parameters.
- An object is instantiated in a complete state.
- Build immutable objects.

Disadvantage

A drawback with the builder design pattern that I came to know is that it increases the lines of code.

Summary

Thus, coming towards the end of this article, we learned how to create a builder design pattern.

What did we learn?

- What is the design pattern?
- Why the builder design pattern?
- Builder design pattern.
- Demo implementation.
- Realtime case implementation.
- Existing implementations in JDK.

- Advantages & disadvantages of a builder design pattern.

Thank you for using C# Corner