Article

Bridge Design Pattern With Java

By **Aman Gupta** on **Updated date Nov 23, 2020**

Introduction

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

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 most famous and commonly used design pattern i.e bridge design pattern. Let's get started.

Why the Bridge Design Pattern?

To start with, what is a bridge design pattern? One should know why we need this design pattern, what are the drawbacks behind the other patterns to need to introduce a new design pattern, and why we need this new pattern in our code.

In the native language, this pattern comes into use to separate abstraction from its implementation so that it can be modified independently, i.e. changing in their implementation should not have an impact on the entire program (recompiled).

We choose the Bridge Design Pattern when:

- We need to avoid a permanent binding between an abstraction and its implementation i.e to avoid tight coupling between interface/abstract class and other classes.
- Both abstraction and their implementation should be extensible by subclassing.
- When changing in their implementation should not have an impact on the customer/client i.e their code should not have to be recompiled.
- When we need to share an implementation among multiple objects, and
- When we need to hide the implementation of an abstraction completely from clients.

To overcome this problem, we need to find a new design pattern.

What is the Bridge Design Pattern?

Bridge Design Pattern is a part of a structural design pattern, as the name suggests it act as an intermediary between two components.

As per the “Gang of Four” definition, the Bridge Design Pattern decouples an abstraction from its implementation so that the two can vary independently which means, this pattern is used to separate abstraction from its implementation so that can be modified independently.

More specifically this pattern involves an interface that acts as a bridge between the abstraction classes and implementer classes. With the bridge pattern, both types of classes can be modified without affecting each other.

Implementation

One thing that comes to mind when we listen to the word “bridge” i.e pillars, and the same applies to this design pattern, i.e. we have an implementer and that implementer has concrete implementers for the given interface that, act as pillars (concrete implementer) to the bridge (implementer).

Let’s assume that our requirement is to process the payment of the customer who had either opted for an option of Credit/Debit card or NetBanking or Pay on the Delivery during its purchase process.

Typically, we are provided with NetBanking, Credit/Debit card, and Pay on the Delivery process as payment choices to do the payments.

Step 1

Construct an implementer i.e “paymentSystem” which has a method to process the payments:

```
1. package BridgeDesign;
2. //Interface
3. public interface paymentSystem {
4.     //Method
5.     void ProcessPayment();
6.     void ProcessPayment(String string);
7. }
```

Step 2

In order to process the payment, there are many payment gateways like SBIpaymentSystem, PNBpaymentSystem, IDBIpaymentSystem, to construct classes for the same.

SBIpaymentSystem class

```
1. package BridgeDesign;
2. public class SBIpaymentSystem implements paymentSystem {
3.     @Override
4.     public void ProcessPayment(String string) {
5.         // TODO Auto-generated method stub
6.         System.out.println("Using SBI payment gateway for " + string);
```

```
    }
    @Override
    public void ProcessPayment() {
        // TODO Auto-generated method stub
        System.out.println("User will Pay on Delivery");
    }
}
```

PNBpaymentSystem class

```
package BridgeDesign;
public class PNBpaymentSystem implements paymentSystem {
    @Override
    public void ProcessPayment(String string) {
        // TODO Auto-generated method stub
        System.out.println("Using PNB payment gateway for " + string);
    }
    @Override
    public void ProcessPayment() {
        // TODO Auto-generated method stub
        System.out.println("User will Pay on Delivery");
    }
}
```

Similarly, for other banks too we will be constructing the same classes. To simplify it, we are just returning the System.out.println line.

Now, we constructed the implementer interface as well as concrete implementers for the given interface.

Step 3

Construct a new abstract class, i.e. payment which had a method to make payment.

```
package BridgeDesign;
//Abstraction
public abstract class Payment {
    paymentSystem payment; //instance
    public abstract void makePayment(); //method responsible to makePayment
}
```

Step 4

Construct those refined abstractions to implement this makePayment() method in the Payment abstract class.

For credit card:

```
package BridgeDesign;
//Refined Abstraction
public class CreditCardPayment extends Payment {
    @Override
    public void makePayment() {
        //payment object provides independency
        payment.ProcessPayment("Credit Card Payment");
    }
}
```

For debit card:

```
1. package BridgeDesign;
2. //Refined Abstraction
3. public class DebitCardPayment extends Payment {
4.     @Override
5.     public void makePayment() {
6.         //payment object provides independency
7.         payment.ProcessPayment("Debit Card Payment");
8.     }
9. }
```

For net banking:

```
1. package BridgeDesign;
2. //Refined Abstraction
3. public class NetBanking extends Payment {
4.     @Override
5.     public void makePayment() {
6.         //payment object provides independency
7.         payment.ProcessPayment("Net Banking");
8.     }
9. }
```

Note
Notice that we are able to bridge the interface and decouple the abstraction implementations.

Step 5

Now switch to the main class:

```
1. package BridgeDesign;
2. public class PaymentProcessor {
3.     public static void main(String[] args) {
4.         //Using PNB payment gateway for Credit Card Payment
5.         Payment order1 = new CreditCardPayment();
6.         order1.payment = new PNBpaymentSystem();
7.         order1.makePayment();
8.         //Using PNB payment gateway for Debit Card Payment
9.         Payment order12 = new DebitCardPayment();
10.        order12.payment = new PNBpaymentSystem();
11.        order12.makePayment();
12.        //Using PNB payment gateway for Net Banking
13.        Payment order11 = new NetBanking();
14.        order11.payment = new PNBpaymentSystem();
15.        order11.makePayment();
16.        //User will Pay on Delivery
17.        Payment order111 = new PayOnDelivery();
18.        order111.payment = new PNBpaymentSystem(); //Pay on Delivery is irrespective of banks
19.        order111.makePayment();
20.        //Using same payment gateway with a different bank
21.        //Using SBI payment gateway for Credit Card Payment
22.        Payment order2 = new CreditCardPayment();
23.        order2.payment = new SBIpaymentSystem();
24.        order2.makePayment();
25.        //Using SBI payment gateway for Debit Card Payment
26.        Payment order21 = new DebitCardPayment();
27.        order21.payment = new SBIpaymentSystem();
28.        order21.makePayment();
29.        //Using SBI payment gateway for Net Banking
```

```
30.         Payment order22 = new NetBanking();
31.         order22.payment = new SBIpaymentSystem();
32.         order22.makePayment();
33.         //User will Pay on Delivery
34.         Payment order222 = new PayOnDelivery();
35.         order222.payment = new SBIpaymentSystem(); //Pay on Delivery is irrespective of banks
36.         order222.makePayment();
37.     }
38. }
```

Output

Using PNB payment gateway for Credit Card Payment
Using PNB payment gateway for Debit Card Payment
Using PNB payment gateway for Net Banking
User will Pay on Delivery
Using SBI payment gateway for Credit Card Payment
Using SBI payment gateway for Debit Card Payment
Using SBI payment gateway for Net Banking
User will Pay on Delivery

Similarly, we can access the payment process with different banks based upon availability.

I hope it will make some sense to understand what the Bridge Pattern is. With this, we successfully implemented the Bridge Design Pattern.

Summary

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

What did we learn?

- What is a Design Pattern?
- Why the Bridge Design Pattern?
- What is a Bridge Design Pattern?
- Demo implementation