



[ANDROID](#) |
 [JAVA](#) |
 [JVM LANGUAGES](#) |
 [SOFTWARE DEVELOPMENT](#) |
 [AGILE](#) |
 [CAREER](#) |
 [COMMUNICATIONS](#) |
 [DEVOPS](#) |
 [META JCG](#)

[Home](#) »
 [Java](#) »
 [Core Java](#) »
 Bridge Design Pattern

ABOUT ROHIT JOSHI



Rohit Joshi is a Senior Software Engineer from India. He is a Sun Certified Java Programmer and has worked on projects related to different domains. His expertise is in Core Java and J2EE technologies, but he also has good experience with front-end technologies like Javascript, JQuery, HTML5, and JQWidgets.

Bridge Design Pattern

Posted by: Rohit Joshi |
 in Core Java |
 September 30th, 2015

This article is part of our Academy Course titled Java Design Patterns.

In this course you will delve into a vast number of Design Patterns and see how those are implemented and utilized in Java. You will understand the reasons why patterns are so important and learn when and how to apply each one of them. Check it out here!

Want to be a Java Master ?

Subscribe to our newsletter and download Java Design Patterns [right now!](#)

In order to help you master the Java programming language, we have compiled a kick-ass guide with all the must-know Design Patterns for Java! Besides studying them online you may download the eBook in PDF format!

Email address:

Sign up

Table Of Contents

1. Introduction
2. What is Bridge Pattern
3. Solution to the Problem
4. Use of Bridge Pattern
5. Download the Source Code

1. Introduction

Sec Security System is a security and electronic company which produces and assembles products for cars. It delivers any car electronic or security system you want, from air bags to GPS tracking system, reverse parking system etc. Big car companies use its products in their cars. The company uses a well defined object oriented approach to keep track of their products using software which is developed and maintained by them only. They get the car, produce the system for it and assemble it into the car.

Recently, they got new orders from BigWheel (a car company) to produce central locking and gear lock system for their new xz model. To maintain this, they are creating a new software system. They started by creating a new abstract class CarProductSecurity, in which they kept some car specific methods and some of the features which they thought are common to all security products. Then they extended the class and created two different sub classes named them BigWheelXZCentralLocking, and BigWheelXZGearLocking. The class diagram looks like this:

NEWSLETTER

173,095 insiders are already enjoying weekly updates and complimentary whitepapers!

Join them now to gain [exclusive access](#) to the latest news in the Java world as well as insights about Android, Spring, Groovy and other related technologies!

Email address:

Sign up

RECENT JOBS

Java Engineer
Atlanta, Georgia

internship for content writer
london, United Kingdom

[VIEW ALL](#)

JOIN US



With **1,240,600** unique visitors and **500** authors placed among related sites at Google, we are constantly being looked out for and encouraged by you. So if you have unique and interesting content then you can check out our **JCG** partners program. You can be a **guest writer** for Java Code Geeks and showcase your writing skills!

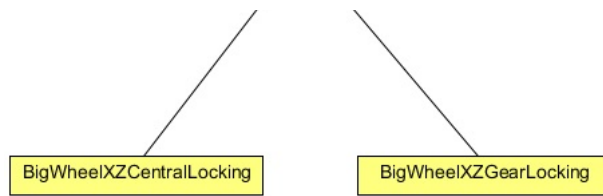


Figure 1

After a while, another car company Motoren asked them to produce a new system of central locking and gear lock for their Im model. Since, the same security system cannot be used in both models of different cars, the Sec Security System has produced the new system for them, and also has created to new classes MotorenLMCentralLocking, and MotorenLMGearLocking which also extend the CarProductSecurity class.

Now the new class diagram looks like this:

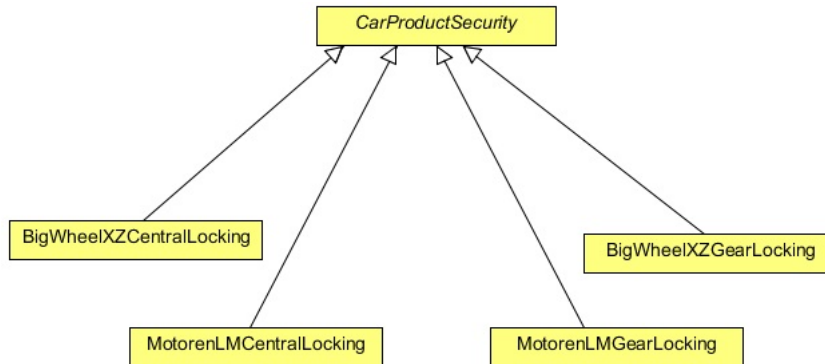


Figure 2

So far so good, but what happens if another car company demands another new system of central locking and gear lock? One needs to create another two new classes for it. This design will create one class per system, or worse, if the reverse parking system is produced for each of these two car companies, two more new classes will be created for each of them.

A design with too many subclasses is not flexible and is hard to maintain. An Inheritance also binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse the abstraction and implementation independently.

Please note that, the car and the product should vary independently in order to make the software system easy to extend and reusable.

The Bridge design pattern can resolve this problem, but before that, let us first have some details about the Bridge Pattern.

2. What is Bridge Pattern

The Bridge Pattern's intent is to decouple an abstraction from its implementation so that the two can vary independently. It puts the abstraction and implementation into two different class hierarchies so that both can be extended independently.

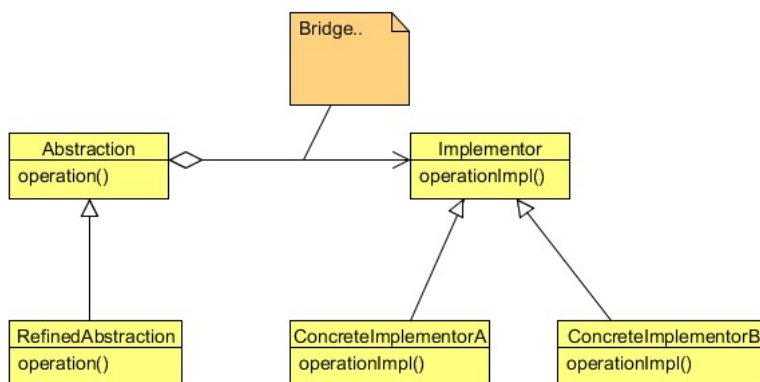


Figure 3

The components of the Bridge Pattern comprise of an abstraction, refined abstraction, an implementer, and concrete implementer.

An abstraction defines the abstraction's interface and also maintains a reference to an object of type implementer, and the link between the abstraction and the implementer is called a Bridge.

Refined Abstraction extends the interface defined by the abstraction.

The Bridge Pattern decouples the interface and the implementation. As a result, an implementation is not bound permanently to an interface. The implementation of an abstraction can be configured at run-time. It also eliminates compile-time dependencies on the implementation. Changing an implementation class doesn't required recompiling the abstraction class and its clients. The Client only needs to know about the abstraction and you can hide the implementation from them.

3. Solution to the Problem

Instead of creating a subclass for each product per car model in the above discussed problem, we can separate the design into two different hierarchies. One interface is for the product which will be used as an implementer and the other will be an abstraction of car type. The implementer will be implemented by the concrete implementers and provides an implementation for it. On the other side, the abstraction will be extended by more refined abstraction.

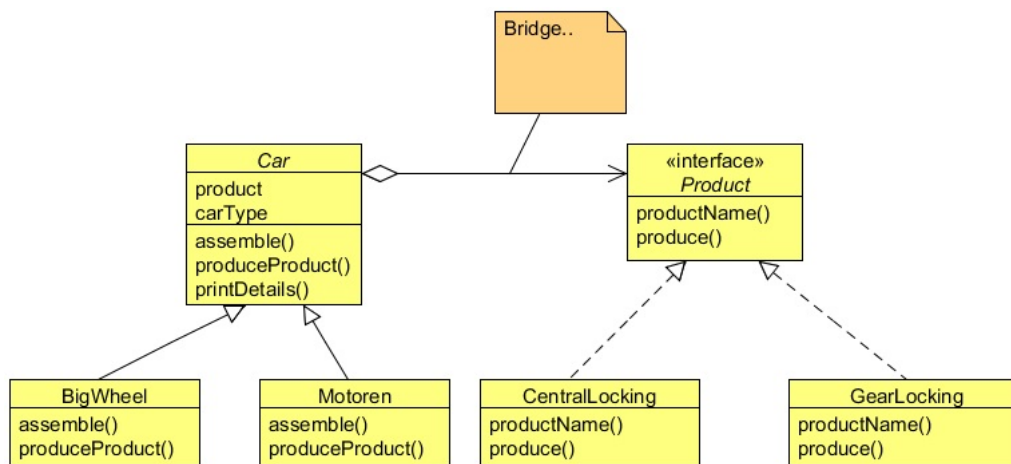


Figure 4

```

1 package com.javacodegeeks.patterns.bridgepattern;
2
3 public interface Product {
4
5     public String productName();
6     public void produce();
7 }
  
```

The implementer

Product

has a method

produce()

which will be used by the concrete implementers to provide concrete functionality to it. The method will produce the base model of the product which can be used with any car model after some modifications specific to that car model.

```

01 package com.javacodegeeks.patterns.bridgepattern;
02
03 public class CentralLocking implements Product{
04     private final String productName;
05
06     public CentralLocking(String productName){
07         this.productName = productName;
08     }
09
10
11     @Override
12     public String productName() {
13         return productName;
14     }
15
16     @Override
17     public void produce() {
18         System.out.println("Producing Central Locking System");
19     }
20 }
21
22
23 package com.javacodegeeks.patterns.bridgepattern;
24
25 public class GearLocking implements Product{
26
27     private final String productName;
28
29     public GearLocking(String productName){
30         this.productName = productName;
31     }
  
```

```

36     }
37
38     @Override
39     public void produce() {
40         System.out.println("Producing Gear Locking System");
41     }
42
43 }

```

The two different concrete implementers provide implementation to the

Product

implementer.

Now the abstraction, the

Car

class which holds a reference of a product type and provides two abstract methods

produceProduct()

and

assemble()

```

01 package com.javacodegeeks.patterns.bridgepattern;
02
03 public abstract class Car {
04
05     private final Product product;
06     private final String carType;
07
08     public Car(Product product,String carType){
09         this.product = product;
10         this.carType = carType;
11     }
12
13     public abstract void assemble();
14     public abstract void produceProduct();
15
16     public void printDetails(){
17         System.out.println("Car: "+carType+", Product:"+product.productName());
18     }
19 }

```

The subclasses of the

Car

will provide the concrete and specific implementation to the methods

assemble()

and

produceProduct()

```

01 package com.javacodegeeks.patterns.bridgepattern;
02
03 public class BigWheel extends Car{
04
05     private final Product product;
06     private final String carType;
07
08     public BigWheel(Product product, String carType) {
09         super(product, carType);
10         this.product = product;
11         this.carType = carType;
12     }
13
14     @Override
15     public void assemble() {
16         System.out.println("Assembling "+product.productName()+" for "+carType);
17     }
18
19     @Override
20     public void produceProduct() {
21         product.produce();
22         System.out.println("Modifying product "+product.productName()+" according to "+carType);
23     }
24 }
25
26
27 package com.javacodegeeks.patterns.bridgepattern;
28
29 public class Motoren extends Car{
30

```

```

35         super(product, carType);
36         this.product = product;
37         this.carType = carType;
38     }
39
40     @Override
41     public void assemble() {
42         System.out.println("Assembling "+product.productName()+" for "+carType);
43     }
44
45     @Override
46     public void produceProduct() {
47         product.produce();
48         System.out.println("Modifying product "+product.productName()+" according to "+carType);
49     }
50 }
51 }

```

Now, let's test the example.

```

01 package com.javacodegeeks.patterns.bridgepattern;
02
03 public class TestBridgePattern {
04
05     public static void main(String[] args) {
06         Product product = new CentralLocking("Central Locking System");
07         Product product2 = new GearLocking("Gear Locking System");
08         Car car = new BigWheel(product, "BigWheel xz model");
09         car.produceProduct();
10         car.assemble();
11         car.printDetails();
12
13         System.out.println();
14
15         car = new BigWheel(product2, "BigWheel xz model");
16         car.produceProduct();
17         car.assemble();
18         car.printDetails();
19
20         System.out.println("-----");
21
22         car = new Motoren(product, "Motoren lm model");
23         car.produceProduct();
24         car.assemble();
25         car.printDetails();
26
27         System.out.println();
28
29         car = new Motoren(product2, "Motoren lm model");
30         car.produceProduct();
31         car.assemble();
32         car.printDetails();
33     }
34 }
35
36 }

```

The above example will produce the following output:

```

01 Producing Central Locking System
02 Modifying product Central Locking System according to BigWheel xz model
03 Assembling Central Locking System for BigWheel xz model
04 Car: BigWheel xz model, Product:Central Locking System
05
06 Producing Gear Locking System
07 Modifying product Gear Locking System according to BigWheel xz model
08 Assembling Gear Locking System for BigWheel xz model
09 Car: BigWheel xz model, Product:Gear Locking System
10 -----
11 Producing Central Locking System
12 Modifying product Central Locking System according to Motoren lm model
13 Assembling Central Locking System for Motoren lm model
14 Car: Motoren lm model, Product:Central Locking System
15
16 Producing Gear Locking System
17 Modifying product Gear Locking System according to Motoren lm model
18 Assembling Gear Locking System for Motoren lm model
19 Car: Motoren lm model, Product:Gear Locking System

```

4. Use of Bridge Pattern

You should use the Bridge Pattern when:

1. You want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.
2. Both the abstractions and their implementations should be extensible by sub-classing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
3. Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
4. You want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client.

