# Design Patterns - Bridge Pattern

设计模式-桥模式

Bridge is used when we need to **decouple （解耦）**an abstraction from its implementation so that the two can vary independently. This type of design pattern comes under structural pattern as this pattern decouples implementation class and abstract class by providing a bridge structure between them.

我们使用桥来解耦一个抽象以及该抽象的实现。使用桥之后抽象和实现可以相互独立的改变。这种类型的设计模式来源于结构型模式，它可以通过使用桥结构来解耦抽象类及其实现类。

This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other.

这种模式涉及一个接口，它扮演一个桥实现化者的角色，使得具体类的功能独立与接口。这两种类型的类可以在不影响对方的情况下改变自身结构。

We are demonstrating use of Bridge pattern via following example in which a circle can be drawn in different colors using same abstract class method but different bridge implementer classes.

我们通过下面的例子来演示桥模式的使用。画一个圆使用不同的颜色，相同的抽象类方法，不同的桥的具体实现者。

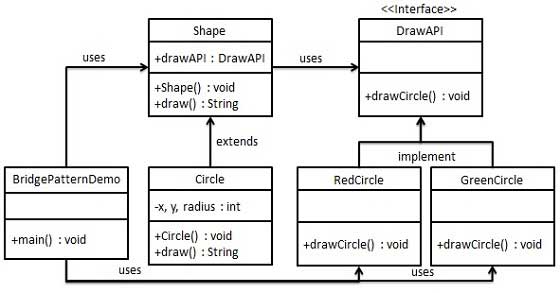
Implementation

实现

We have a *DrawAPI* interface which is acting as a bridge implementer and concrete classes *RedCircle*, *GreenCircle* implementing the *DrawAPI* interface.*Shape* is an abstract class and will use object of *DrawAPI*. *BridgePatternDemo*, our demo class will use *Shape* class to draw different colored circle.

我们有一个DrawAPI接口，它扮演一个桥的实现化者的角色，然后会有具体的类RedCircle和GreenCircle实现接口DrawAPI。抽象类Shape将持有DrawAPI对象。BridgePatternDemo，我们的demo类将使用Shape类画两个不同颜色的圆。

译者注：bridge implementer 这里翻译为桥的实现化者，它不同于具体的实现，如：继承，实现。这里的实现是指对桥这种概念的具体化，实现化。



Step 1

Create bridge implementer interface.

创建一个桥的实现化者接口DrawAPI

*DrawAPI.java*

public interface DrawAPI {

public void drawCircle(int radius, int x, int y);

}

Step 2

Create concrete bridge implementer classes implementing the *DrawAPI*interface.

创建具体的类实现DrawApI接口

*RedCircle.java*

public class RedCircle implements DrawAPI {

@Override

public void drawCircle(int radius, int x, int y) {

System.out.println("Drawing Circle[ color: red, radius: " + radius + ", x: " + x + ", " + y + "]");

}

}

*GreenCircle.java*

public class GreenCircle implements DrawAPI {

@Override

public void drawCircle(int radius, int x, int y) {

System.out.println("Drawing Circle[ color: green, radius: " + radius + ", x: " + x + ", " + y + "]");

}

}

Step 3

Create an abstract class *Shape* using the *DrawAPI* interface.

创建一个抽象类 Shape，该类持有一个DrawAPI接口的引用。

*Shape.java*

public abstract class Shape {

protected DrawAPI drawAPI;

protected Shape(DrawAPI drawAPI){

this.drawAPI = drawAPI;

}

public abstract void draw();

}

Step 4

Create concrete class implementing the *Shape* interface.

创建一个具体类实现抽象类Shape。

*Circle.java*

public class Circle extends Shape {

private int x, y, radius;

public Circle(int x, int y, int radius, DrawAPI drawAPI) {

super(drawAPI);

this.x = x;

this.y = y;

this.radius = radius;

}

public void draw() {

drawAPI.drawCircle(radius,x,y);

}

}

Step 5

Use the *Shape* and *DrawAPI* classes to draw different colored circles.

使用Shape和DrawAPI类画两个不同颜色的圆。

*BridgePatternDemo.java*

public class BridgePatternDemo {

public static void main(String[] args) {

Shape redCircle = new Circle(100,100, 10, new RedCircle());

Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

redCircle.draw();

greenCircle.draw();

}

}

Step 6

Verify the output.

校验输出。

Drawing Circle[ color: red, radius: 10, x: 100, 100]

Drawing Circle[ color: green, radius: 10, x: 100, 100]