Design Patterns - Decorator Pattern

设计模式-装饰器模式

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

We are demonstrating the use of decorator pattern via following example in which we will decorate a shape with some color without alter shape class.

装饰器模式允许使用者将新功能添加到现有的对象而不需要改变它的数据结构。这种类型的设计模式来源于结构型模式，该设计模式将会去包装一个现有的类。

这种设计模式会常见一个装饰器类，它包装了原始类，并且在不改变原始类的方法的基础之上添加额外的新的功能。

我们将通过下面的例子来展示如何使用装饰器模式，在接下来的例子中我们将用颜色来装饰图形，但不需要修改图形的类。

Implementation

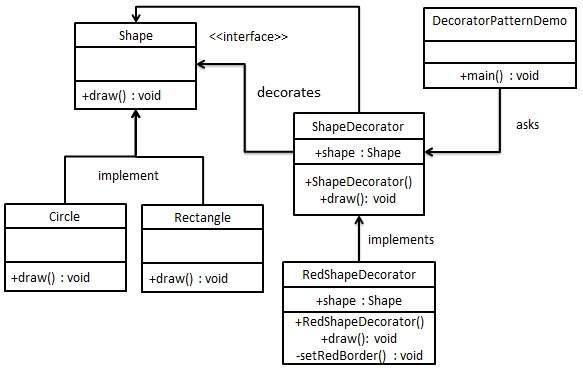
We're going to create a *Shape* interface and concrete classes implementing the*Shape* interface. We will then create an abstract decorator class *ShapeDecorator*implementing the *Shape* interface and having *Shape* object as its instance variable.

*RedShapeDecorator* is concrete class implementing *ShapeDecorator*.

*DecoratorPatternDemo*, our demo class will use *RedShapeDecorator* to decorate*Shape* objects.

我们将创建一个Shape接口和实现该接口的具体类。然后在创建一个抽象的ShaperDecorator类，该类也实现了Shape接口，并且持有一个Shape类的对象。

RedShapeDecorator 作为具体类实现了ShapeDecorator。DecoratorPatternDemo，作为我们的demo类将会去使用RedShapeDecorator去装饰Shape对象。



Step 1

Create an interface.

创建接口

*Shape.java*

public interface Shape {

void draw();

}

Step 2

Create concrete classes implementing the same interface.

创建具体的类来实现Shape接口

*Rectangle.java*

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Shape: Rectangle");

}

}

*Circle.java*

public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Shape: Circle");

}

}

Step 3

Create abstract decorator class implementing the *Shape* interface.

创建抽象的装饰器类实现Shape接口。

*ShapeDecorator.java*

public abstract class ShapeDecorator implements Shape {

protected Shape decoratedShape;

public ShapeDecorator(Shape decoratedShape){

this.decoratedShape = decoratedShape;

}

public void draw(){

decoratedShape.draw();

}

}

Step 4

Create concrete decorator class extending the *ShapeDecorator* class.

创建具体的装饰器类，该类继承了ShaperDecoratoe类。

*RedShapeDecorator.java*

public class RedShapeDecorator extends ShapeDecorator {

public RedShapeDecorator(Shape decoratedShape) {

super(decoratedShape);

}

@Override

public void draw() {

decoratedShape.draw();

setRedBorder(decoratedShape);

}

private void setRedBorder(Shape decoratedShape){

System.out.println("Border Color: Red");

}

}

Step 5

Use the *RedShapeDecorator* to decorate *Shape* objects.

使用RedShapeDecorator装饰Shape对象。

*DecoratorPatternDemo.java*

public class DecoratorPatternDemo {

public static void main(String[] args) {

Shape circle = new Circle();

Shape redCircle = new RedShapeDecorator(new Circle());

Shape redRectangle = new RedShapeDecorator(new Rectangle());

System.out.println("Circle with normal border");

circle.draw();

System.out.println("\nCircle of red border");

redCircle.draw();

System.out.println("\nRectangle of red border");

redRectangle.draw();

}

}

Step 6

Verify the output.

检验输出

Circle with normal border

Shape: Circle

Circle of red border

Shape: Circle

Border Color: Red

Rectangle of red border

Shape: Rectangle

Border Color: Red