TutorialsPoint

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.

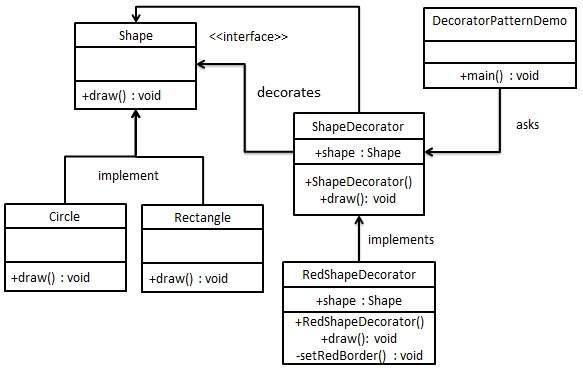
We are demonstrating the use of decorator pattern via following example in which we will decorate a shape with some color without alter shape class.

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.



Step 1

Create an interface.

*Shape.java*

public interface Shape {

void draw();

}

Step 2

Create concrete classes implementing the same interface.

*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.

*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.

*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.

*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.

Circle with normal border

Shape: Circle

Circle of red border

Shape: Circle

Border Color: Red

Rectangle of red border

Shape: Rectangle

Border Color: Red

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////DIFFERENT SOURCE///////////////////////////////////////////////////

StackOverflow

**Decorator pattern achieves a single objective of dynamically adding responsibilities to any object.**

Consider a case of a pizza shop. In the pizza shop they will sell few pizza varieties and they will also provide toppings in the menu. Now imagine a situation wherein if the pizza shop has to provide prices for each combination of pizza and topping. Even if there are four basic pizzas and 8 different toppings, the application would go crazy maintaining all these concrete combination of pizzas and toppings.

Here comes the decorator pattern.

As per the decorator pattern, you will implement toppings as decorators and pizzas will be decorated by those toppings' decorators. Practically each customer would want toppings of his desire and final bill-amount will be composed of the base pizzas and additionally ordered toppings. Each topping decorator would know about the pizzas that it is decorating and it's price. GetPrice() method of Topping object would return cumulative price of both pizza and the topping.

# EDIT

Here's a code-example of explanation above.

public abstract class BasePizza

{

protected double myPrice;

public virtual double GetPrice()

{

return this.myPrice;

}

}

public abstract class ToppingsDecorator : BasePizza

{

protected BasePizza pizza;

public ToppingsDecorator(BasePizza pizzaToDecorate)

{

this.pizza = pizzaToDecorate;

}

public override double GetPrice()

{

return (this.pizza.GetPrice() + this.myPrice);

}

}

class Program

{

[STAThread]

static void Main()

{

//Client-code

Margherita pizza = new Margherita();

Console.WriteLine("Plain Margherita: " + pizza.GetPrice().ToString());

ExtraCheeseTopping moreCheese = new ExtraCheeseTopping(pizza);

ExtraCheeseTopping someMoreCheese = new ExtraCheeseTopping(moreCheese);

Console.WriteLine("Plain Margherita with double extra cheese: " + someMoreCheese.GetPrice().ToString());

MushroomTopping moreMushroom = new MushroomTopping(someMoreCheese);

Console.WriteLine("Plain Margherita with double extra cheese with mushroom: " + moreMushroom.GetPrice().ToString());

JalapenoTopping moreJalapeno = new JalapenoTopping(moreMushroom);

Console.WriteLine("Plain Margherita with double extra cheese with mushroom with Jalapeno: " + moreJalapeno.GetPrice().ToString());

Console.ReadLine();

}

}

public class Margherita : BasePizza

{

public Margherita()

{

this.myPrice = 6.99;

}

}

public class Gourmet : BasePizza

{

public Gourmet()

{

this.myPrice = 7.49;

}

}

public class ExtraCheeseTopping : ToppingsDecorator

{

public ExtraCheeseTopping(BasePizza pizzaToDecorate)

: base(pizzaToDecorate)

{

this.myPrice = 0.99;

}

}

public class MushroomTopping : ToppingsDecorator

{

public MushroomTopping(BasePizza pizzaToDecorate)

: base(pizzaToDecorate)

{

this.myPrice = 1.49;

}

}

public class JalapenoTopping : ToppingsDecorator

{

public JalapenoTopping(BasePizza pizzaToDecorate)

: base(pizzaToDecorate)

{

this.myPrice = 1.49;

}

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////DIFFERENT SOURCE///////////////////////////////////////////////////

ProgramCreek

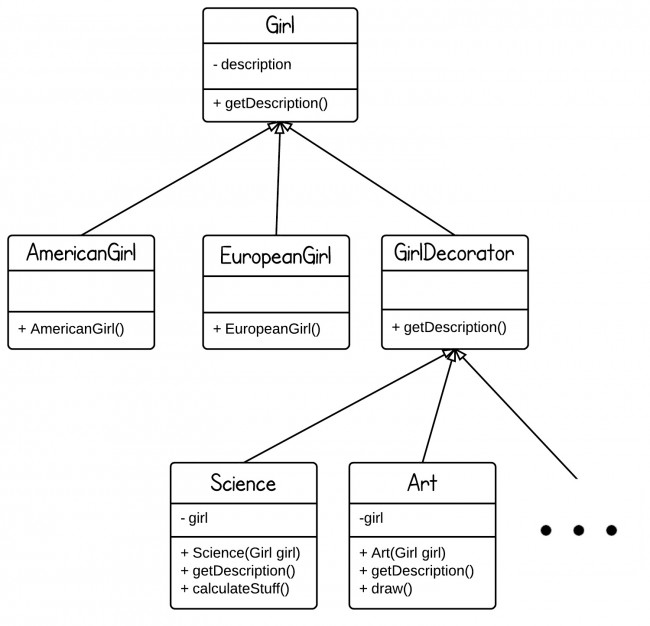
Decorator pattern adds additional features to an existing object dynamically. In this post, I will use a simple example - decorate your girlfriend - to illustrate how decorator pattern works.

1. Decorator Pattern Story

Let's assume you are looking for a girlfriend. There are girls from different countries such as America, China, Japan, France, etc. They may have different personalities and hobbies. In a dating web like eharmony.com, if each type of girl is an individual Java class, there would be thousands of classes. That is a serious problem called *class explosion*. Moreover, this design is not extensible. Whenever there is a new girl type, a new class needs to be created.

Let's change the design, and let each hobby/personality becomes a decorator which can be dynamically applied to a girl.

2. Class Diagram

  
Girl is the abstract class at the top level, we have girls from different countries. With a GirlDecorator class, we can decorator each girl with any feature by adding a new decorator.

3. Decorator pattern Java code

Girl.java

|  |
| --- |
| **public** **abstract** **class** Girl {  String description = "no particular";    **public** String getDescription(){  **return** description;  }  } |

AmericanGirl.java

|  |
| --- |
| **public** **class** AmericanGirl **extends** Girl {  **public** AmericanGirl(){  description = "+American";  }  } |

EuropeanGirl.java

|  |
| --- |
| **public** **class** EuropeanGirl **extends** Girl {  **public** EuropeanGirl() {  description = "+European";  }  } |

GirlDecorator.java

|  |
| --- |
| **public** **abstract** **class** GirlDecorator **extends** Girl {  **public** **abstract** String getDescription();  } |

Science.java

|  |
| --- |
| **public** **class** Science **extends** GirlDecorator {    **private** Girl girl;    **public** Science(Girl g) {  girl = g;  }    @Override  **public** String getDescription() {  **return** girl.getDescription() + "+Like Science";  }    **public** **void** caltulateStuff() {  System.out.println("scientific calculation!");  }  } |

We can add more method like "Dance()" to each decorator without any limitations.

Art.java

|  |
| --- |
| **public** **class** Art **extends** GirlDecorator {    **private** Girl girl;    **public** Art(Girl g) {  girl = g;  }    @Override  **public** String getDescription() {  **return** girl.getDescription() + "+Like Art";  }    **public** **void** draw() {  System.out.println("draw pictures!");  }  } |

Main.java

|  |
| --- |
| **package** designpatterns.decorator;    **public** **class** Main {    **public** **static** **void** main(String[] args) {  Girl g1 = **new** AmericanGirl();  System.out.println(g1.getDescription());    Science g2 = **new** Science(g1);  System.out.println(g2.getDescription());    Art g3 = **new** Art(g2);  System.out.println(g3.getDescription());  }  } |

Output:

+American  
+American+Like Science  
+American+Like Science+Like Art

We can also do something like this:

|  |
| --- |
| Girl g = **new** Science(**new** Art(**new** AmericanGirl())); |

4. Decorator Pattern Used in Java Stand Library

A typical usage of Decorator pattern is Java IO classes.

Here is a simple example - BufferedReader decorates InputStreamReader.

|  |
| --- |
| BufferedReader input = **new** BufferedReader(**new** InputStreamReader(System.in));  *//System.in is an InputStream object* |

InputStreamReader(InputStream in) - bridge from byte streams to character streams. InputSteamReader reads bytes and translates them into characters using the specified character encoding.

BufferedReader(Reader in) - read text from a character stream and buffer characters in order to provide efficient reading methods(e.g., readLine())

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////DIFFERENT SOURCE///////////////////////////////////////////////////