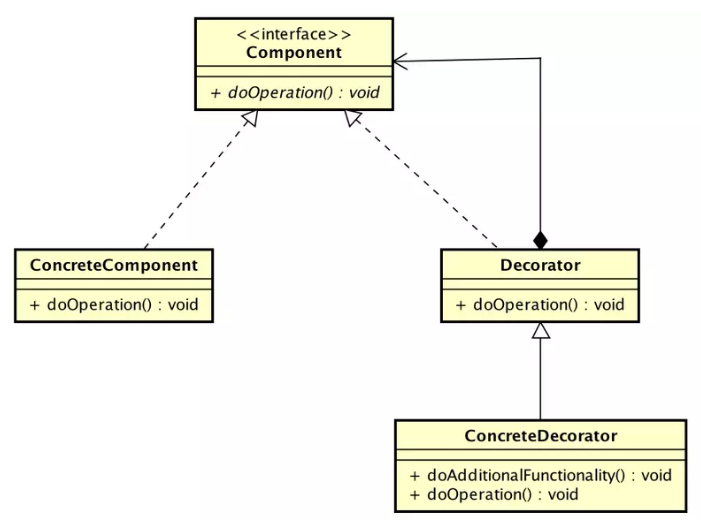
# **Decorator pattern**

Đôi khi chúng ta cần mở rộng một phương thức trong đối tượng, và cách thông thường là chúng ta sẽ kế thừa đối tượng đó. Việc này không phải sai, nhưng trong một vài trường hợp sẽ làm cho mã nguồn trở lên phức tạp hơn chúng ta mong muốn. Đó là lý do chính cho việc ra đời của mẫu thiết kế **Decorator**, là một cách để mở rộng các phương thức một cách linh động.

Mẫu thiết kế **Decorator** là một trong nhiều những mẫu thiết kế cấu trúc thường được sử dụng. Mẫu thiết kế này sẽ linh động thay đổi tính chất (functionality) đã có trong một đối tượng khi chương trình đang chạy (runtime) mà không ảnh hưởng đến các tình chất đã tồn tại của các đối tượng khác.

Vậy điều gì là điểm khác biệt giữa mở rộng phương thức theo cách linh động với mở rộng phương thức theo cách tĩnh? Về cơ bản, để mở rộng phương thức của đối tượng bằng cách kế thừa chúng ta cần phải triển khai code để mở rộng lớp có sẵn và viết lại (overide) các phương thức của lớp đó. Và những mở rộng đó sẽ được cài đặt ngay khi biên dịch chương trình. Trong một hình thức khác, chúng ta sẽ tạo ra một lớp mới đồng thời chúng ta sẽ áp dụng sự thay đổi trên tất các đối tượng thuộc lớp này. Mở rộng theo cách linh động là chúng ta sẽ cung cấp một cơ cấu mà cơ cấu này cho phép chúng ta thay đổi một đối tượng đã tồn tại nhưng không làm ảnh hưởng đến các đối tượng khác của cùng lớp đó.

Thêm nữa, có nhiều trường hợp mà việc sử dụng kế thừa sẽ mất nhiều công sức trong việc viết code. Mặt khác, mẫu thiết kế **Decorator** sẽ giảm cho bạn một lượng lớn thời gian viết code. Hãy xem xem xét ví dụ sau:

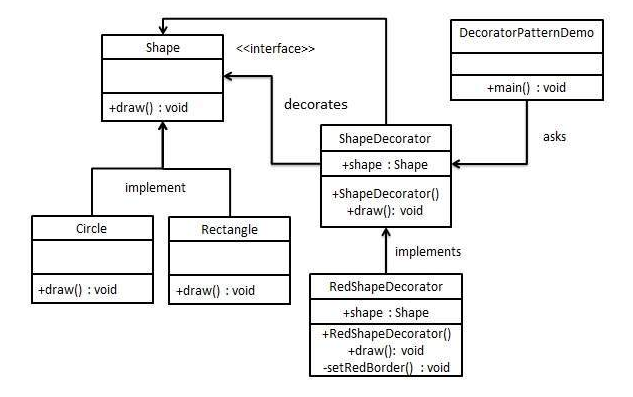


**Component**: giao diện (interface) chung để các đối tượng cần thêm chức năng trong quá trình chạy thì triển khai giao diện này.

**ConcreteComponent** : Một cài đặt cho giao diện Component mà nó định nghĩa một đối tượng cần thêm các chức năng trong quá trình chạy.

**Decorator** : một lớp trừu tượng dùng để duy trì một tham chiếu của đối tượng thành phần và đồng thời cài đặt các thành phần của giao diện.

**ConcreteDecorator** : Một cài đặt của Decorator, nó cài đặt thêm các thành phần vào đầu của các đối tượng thành phần.



Chúng ta sẽ tạo một giao diện Shape và các lớp cụ thể triển khai giao diện Shape. Sau đó, chúng ta sẽ tạo một lớp trang trí trừu tượng **ShapeDecorator** triển khai giao diện Shape và có đối tượng Shape làm biến thể hiện của nó.

**RedShapeDecorator** là lớp cụ thể triển khai **ShapeDecorator**.

**DecoratorPatternDemo**, lớp demo của chúng ta sẽ sử dụng **RedShapeDecorator** để trang trí các đối tượng Shape.

1.

Tạo interface Shape.java

Giao diện **Shape** là thành phần Component trong mẫu thiết kế Decorator, nó chứa phương thức **draw**, đây là phương thức dùng để tạo ra một **shape** phù hợp.

public interface Shape {

void draw();

}

2.

Tạo các lớp cụ thể triển khai cùng một giao diện: **Rectangle**.java và **Circle**.java. **Rectangle** và **Circle** là những cài đặt, triển khai của Shape. Chúng cung cấp cụ thể các thể hiện của lớp mà chúng ta cẩn mở rộng trong quá trình chương trình đang chạy.

**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");

}

}

3.

Tạo một abstract decorator class để implementing **Shape** interface.

**ShapeDecorator** là trung tâm của sơ đồ thiết kế trên. Nó giữ một thể hiện đã tồn tại của Shape như **Rectangle** hoặc **Circle** . Thuộc tính này sẽ được cài đặt thông qua phương thức cởi tạo, và nó được mở rộng trong khi chương trình chạy.

ShapeDecorator.java

public abstract class ShapeDecorator implements Shape {

protected Shape decoratedShape;

public ShapeDecorator(Shape decoratedShape){

this.decoratedShape = decoratedShape;

}

public void draw(){

decoratedShape.draw();

}

}

4.

**RedShapeDecorator** cài đặt các phương thức mở rộng, **RedShapeDecorator** sẽ thêm viền màu đỏ cho Shape. Tính năng mở rộng là được cài đặt trong phương thức **setRedBorder**().

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");

}

}

5.

Và cuối cùng chúng ta cần viết lớp để chạy các cài đặt ở bên trên là lớp

DecoratorPatternDemo.java

public class DecoratorPatternDemo {

public static void main(String[] args) {

Shape circle = new Circle();

Shape redCircle = new RedShapeDecorator(circle);

Shape rectangle = new Rectangle();

Shape redRectangle = new RedShapeDecorator(rectangle);

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

circle.draw();

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

redCircle.draw();

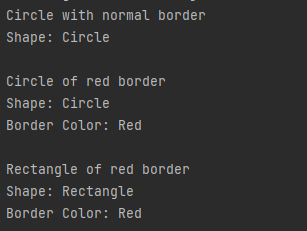
System.*out*.println("\nRectangle of red border");

redRectangle.draw();

}

}

6. Kết quả chạy chương trình



Source: <https://github.com/thaiphake46/Decorator-.git>