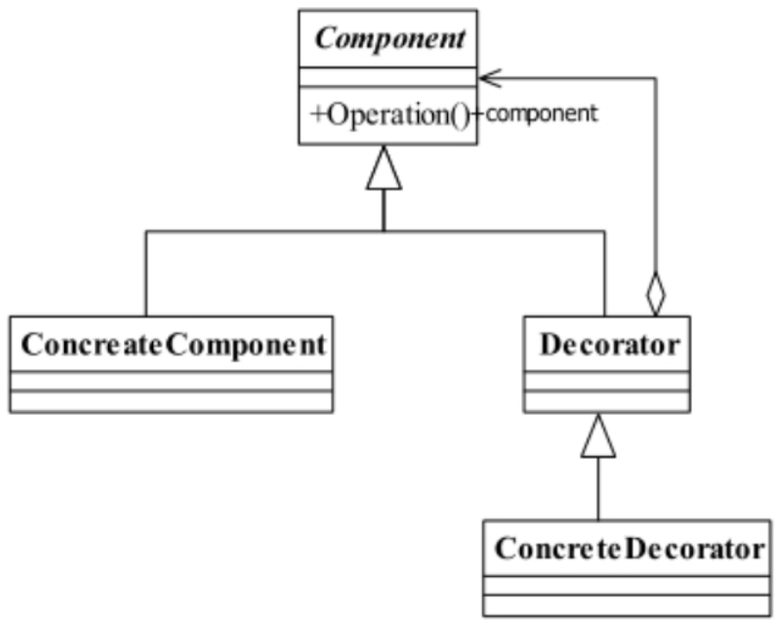
Decorator Pattern

Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.

优缺点用途

- 优点
  - 装饰类和被装饰类可以独立发展，而不会相互耦合
  - 是继承关系的一个替代方案
  - 可以动态地扩展一个实现类的功能
- 缺点
  - 装饰类层数量多的话，不好调试
- 使用
  - 需要扩展一个类的功能，或给一个类增加附加功能
  - 需要动态地给一个对象增加功能，这些功能可以再动态地撤销
  - 需要为一批的兄弟类进行改装或加装功能，当然是首选装饰模式

```
public abstract class Component{
    //抽象方法
    public abstract void operate();
}
```




```
public class ConcreteComponent extends Component{
    //具体实现
    @Override
    public void operate(){
        System.out.println("do Something");
    }
}
```

```
public abstract class Decorator extends Component{
    private Component component=null;
    //通过构造函数传递被修饰者
    public Decorator(Component _component){
        this.component=_component;
    }
    //委托给被修饰者执行
    @Override
    public void operate(){
        this.component.operate();
    }
}
```

```
public class Client{
    public static void main(String[] args){
        Component component=new ConcreteComponent();
        //第一次修饰
        component=new ConcreteDecorator1(component);
        //第二次修饰
        component=new ConcreteDecorator2(component);
        //修饰后运行
        component.operate();
    }
}
```

```
public class ConcreteDecorator1 extends Decorator{
    //定义被修饰者
    public ConcreteDecorator1(Component _component){
        super(_component);
    }
    //定义自己的修饰方法
    private void method1(){
        System.out.println("method1 修饰");
    }
    //重写父类的Operation方法
    public void operate(){
        this.method1();
        super.operate();
    }
}

public class ConcreteDecorator2 extends Decorator{
    //定义被修饰者
    public ConcreteDecorator2(Component _component){
        super(_component);
    }
    //定义自己的修饰方法
    private void method2(){
        System.out.println("method2修饰");
    }
    //重写父类的Operation方法
    public void operate(){
        super.operate();
        this.method2();
    }
}
```