装饰模式

1.装饰模式

装饰模式（Decorator Pattern）也称为包装模式（Wrapper Pattern），结构型设计模式之一，其使用一种对客户端透明的方式来动态地扩展对象的功能，同时它也是继承关系的一种替代方案之一。在现实生活中你也可以看见很多装饰模式的例子，或者可以大胆地说出装饰模式无处不在，就拿人来说，人需要各式各样的衣着，不管你穿着怎样，但是，对于个人的本质来说是不变的，充其量只是在外面披上一层遮盖物而已，这就是装饰模式，装饰物也许各不相同但是装饰的对象本质是不变的。

2.装饰模式的定义

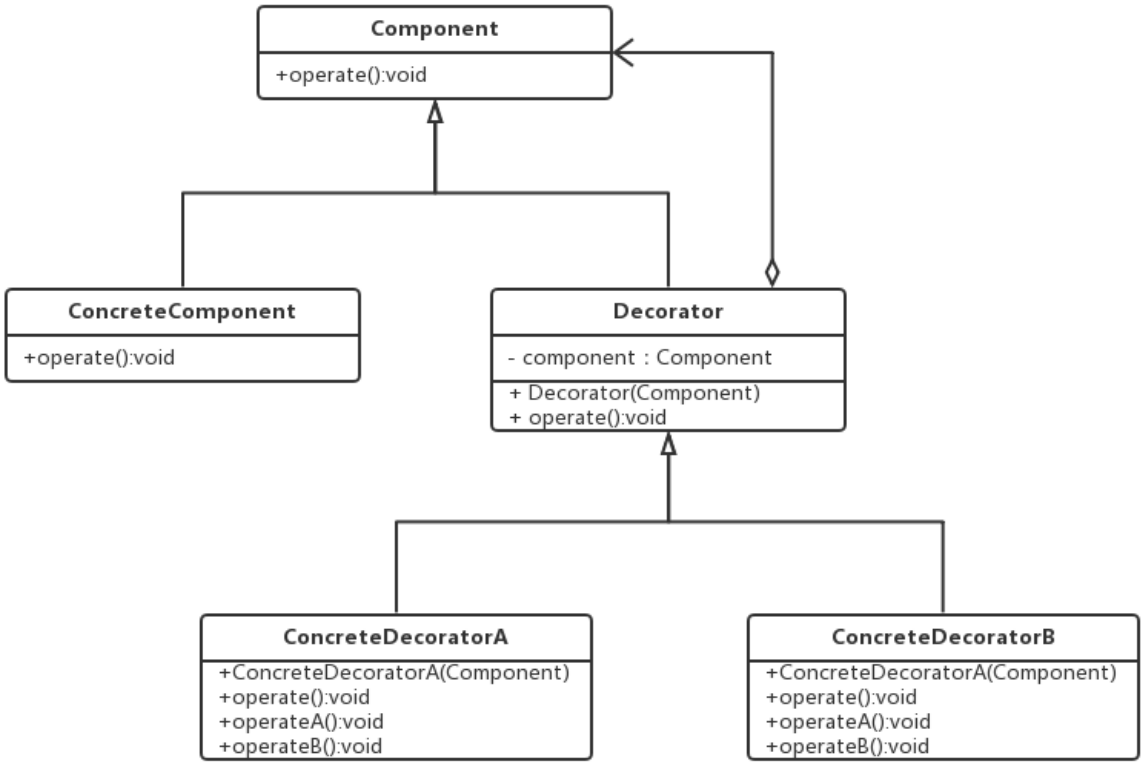
动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比生成子类更为灵活。

3.装饰模式的使用场景

需要透明且动态的扩展类的功能时。

4.装饰模式的UML类图

UML类图如图所示：



角色介绍：

- Component:抽象组件。
  可以是一个接口或抽象类，其充当的就是被装饰的原始对象。
- ConcreteComponent:组件具体实现类。
  该类是COmponent类的基本实现，也是我们装饰的具体对象。
- Decorator：抽象装饰者。
  顾明思议，其承担的职责就是为了装饰我们的组件对象，其内部一定要有一个指向组件对象的引用。在大多数情况下，该类为抽象类，需要根据不同的装饰逻辑实现不同的具体子类。当然，如果装饰逻辑单一，只有一个的情况下我们可以忽略该类直接作为具体的装饰者。
- ConcreteDecoratorA：装饰者具体实现类。
  只是对抽象装饰者作出具体的实现。

根据类图可以得出通用模式代码

抽象组件类

```
package decorator_common;
//抽象组件类
public abstract class Component {
        /*
         * 抽象的方法，这个随你做 同样地你也可以增加更多的抽象方法
         */
        public abstract void operate();
}

```

组件具体实现类

```
package decorator_common;

//组件具体实现类
public class ConcreteComponent extends Component {

	@Override
	public void operate() {
		// 具体逻辑，这个随你做
	}

}

```

抽象装饰者

```
package decorator_common;

//抽象装饰者
public abstract class Decorator extends Component {
	private Component component;// 持有一个Component对象的引用

	/*
	 * 必要的构造方法，需要一个Component类型的对象
	 */
	public Decorator(Component component) {
		this.component = component;
	}

	@Override
	public void operate() {
		component.operate();
	}

}

```

装饰者实现类

```
package decorator_common;

//装饰者具体实现类
public class ConcreteDecoratorA extends Decorator {

	protected ConcreteDecoratorA(Component component) {
		super(component);
	}

	@Override
	public void operate() {
		// 装饰方法A和B即可在父类方法前调用也可在之后调用
		operatorA();
```

```
                super.operate();
                operatorB();
        }

        public void operatorA() {
                // 装饰方法逻辑...
        }

        public void operatorB() {
                // 装饰方法逻辑...
        }
}

```

5.模式的简单实现

文章开头我们所讲到的那个例子，人总是要穿衣服的，我们将人定义为一个抽象类，将其穿衣的行为定义为一个抽象方法。

```
//抽象组件类
public abstract class Person {
        /*
         * 抽象的方法，这个随你做 同样地你也可以增加更多的抽象方法
         */
        public abstract void dressed();
}
```

该类其实就是上面我们所提及的抽象组件类，也就是我们需要装饰的原始对象，那么具体装饰谁呢？我们需要一个具体的实现类。

```
//组件具体实现类
public class Boy extends Person {

        @Override
        public void dressed() {
                //Boy类dressed方法的基本逻辑
                System.out.println("穿了内衣内裤");
        }
}
```

Boy类继承于Person类，该类仅对Person中的dressed方法作了具体的实现，而Boy类则是我们所要装饰的具体对象，现在需要一个装饰者来装饰我们得这个Boy对象，这里定义一个PersonCloth类表示人所穿着的衣服。

```
//抽象装饰者
public abstract class PersonCloth extends Person {
        private Person mPerson;// 持有一个Person对象的引用

        /*
         * 必要的构造方法，需要一个Component类型的对象
```

```
         */
        public PersonCloth(Person mPerson) {
                this.mPerson = mPerson;
        }

        @Override
        public void dressed() {
                mPerson.dressed();
        }

}

```

在PersonCloth类中我们保持了一个对Person类的引用，可以方便地调用具体的被装饰对象中的方法，这就是为什么我们可以在不破坏原类层次结构的情况下增加一些功能，我们只需要在被装饰对象的相应方法前或后增加相应的功能逻辑即可。在装饰物只有一个的情况下，可不比声明一个抽象类作为装饰者抽象的提取，仅需定一个普通的类表示装饰者即可，这里为了表明示例我们定义两种衣服类型，一个类ExpensiveCloth表示高档衣服。

```
//装饰者具体实现类
public class ExpensiveCloth extends PersonCloth {

        protected ExpensiveCloth(Person person) {
                super(person);
        }

        @Override
        public void dressed() {
                super.dressed();
                dressShirt();
                dressLeather();
        }

        /*
         * 穿短袖
         */
        public void dressShirt() {
                System.out.println("穿件短袖");
        }

        /*
         * 穿皮衣
         */
        public void dressLeather() {
                // 装饰方法逻辑...
                System.out.println("穿件皮衣");
        }

}
```

逻辑依旧很简单，另一个类CheapCloth则表示便宜的衣服。

```
//装饰者具体实现类
public class CheapCloth extends PersonCloth {
```

```
3
4           protected CheapCloth(Person person) {
5                   super(person);
6           }
7
8           @Override
9           public void dressed() {
10                  super.dressed();
11                  dressShorts();
12          }
13
14          /*
15           * 穿短裤
16           */
17          public void dressShorts() {
18                  System.out.println("穿件短裤");
19          }
20
21
22  }
```

这两个类本质上并没有区别，两者都是为原本Boy类中的dressed方法提供功能扩展，不过这种扩展并非是直接修改原有的方法逻辑或结构，更恰当地说，仅仅是在另一个中将原有的方法和新逻辑进行分装整合而已。最后我们来看看客户类中的调用。

```
1   package decorator_common;
2
3   public class Client {
4
5           public static void main(String[] args) {
6                   //首先我们要有一个Person男孩
7                   Person person=new Boy();
8                   //然后为他穿上便宜的衣服
9                   PersonCloth clothCheap=new CheapCloth(person);
10                  clothCheap.dressed();
11
12                  PersonCloth clothExpensive=new ExpensiveCloth(person);
13                  clothExpensive.dressed();
14          }
15  }
```

输出：

```
1   穿了内衣内裤
2   穿件短裤
3   穿了内衣内裤
4   穿件短袖
5   穿件皮衣
6
```

6.总结

装饰模式和我们前面讲到的代理模式有点类似，有时甚至容易混淆，但不如说会把代理当成装饰，而常常会是将装饰看作代理，所以大家一定要多留心，装饰模式是以客户端透明的方式扩展对象的功能，是继承关系的一个替代方案；而代理模式则是给一个对象提供一个代理对象，并有代理对象来控制对原有对象的引用。装饰模式应该为所装饰的对象增强功能；代理模式对代理的对象施加控制，但不对对象本身的功能进行增强。