## 装饰者模式

装饰者模式（Decorator Pattern）：动态地给一个对象增加一些额外的职责，就增加对象功能来说，装饰模式比生成子类实现更为灵活。装饰模式是一种对象结构型模式。

## 1. 介绍

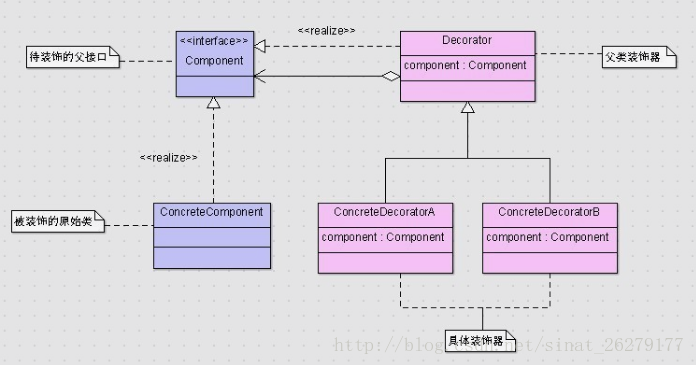
装饰者模式主要有Component、ConcreteComponent、Decorator和ConcreteDecorator组成。

****抽象组件角色(Component)****：定义一个对象接口，以规范准备接受附加责任的对象，即可以给这些对象动态地添加职责。 可以是接口也可以是抽象类，甚至是一个普通的父类（这个强烈不推荐，普通的类作为继承体系的超级父类不易于维护）。

****具体组件角色(ConcreteComponent)**** :被装饰者，定义一个将要被装饰增加功能的类。可以给这个类的对象添加一些职责。

****抽象装饰器(Decorator)****:维持一个指向构件Component对象的实例，并定义一个与抽象组件角色Component接口一致的接口。每个装饰者都有一个（包装一个）组件，即装饰者有一个实例变量以保存某个Component的引用，可以是接口或抽象类。

****具体装饰器角色（ConcreteDecorator)****:向组件添加职责。新行为是通过在旧行为面前或者后面做一些计算来添加的。有一个实例变量可以记录所装饰的事物。

类图如下   


## 2. 实现

## 2.1. 装饰模式的简化

如果只有一个ConcreteComponent类，那么可以考虑去掉抽象的Component类（接口），把Decorator作为一个ConcreteComponent子类。   
如果只有一个ConcreteDecorator类，那么就没必要建立一个单独的Decorator类，而可以把Decorator和ConcreteDecorator的责任合并成一个类。

装饰者模式有透明和半透明两种，这两种的区别在于装饰角色的接口与抽象构件角色的接口是否完全一致。透明的装饰模式也就是理想的装饰模式，要求具体构件角色、装饰角色的接口与抽象构件的接口完全一致。相反，如果装饰角色的接口与抽象构件角色接口不一致，也就是说装饰角色的接口比抽象构件角色的接口宽的话，装饰角色实际上已经成了一个适配器角色，这种装饰模式也是可以接受的，称为“半透明”的装饰模式

## 2.2. 透明装饰者模式

在透明装饰者模式中，要求客户端完全针对抽象编程，装饰模式的透明性要求客户端程序不应该将对象声明为具体构件类型或具体装饰类型，而应该全部声明为抽象构件类型。对于客户端而言，具体构件对象和具体封装对象没有任何区别：   
应该使用如下代码   
Component c，d   
c=new ConcreteComponent();   
d=new ConcreteDecorator(c)

而不应该使用如下代码   
ConcreteComponent c；   
c=new ConcreteComponent()   
或   
ConcreteDecorator d;   
d=new ConcreteDecorator(c)   
透明模式可以让客户端透明地使用装饰之前的对象和装饰之后的对象，无须关心它们的区别，此外还可以对一个已装饰过的对象进行多次装饰，得到更为复杂、功能更为强大的对象。在实现透明装饰模式时，要求具体装饰类的operation方法覆盖抽象装饰类的operation方法，除了调用原有对象的operation外还需要调用新增的addedBehavior()方法来增加新行为。

## 3. 半透明模式

透明装饰模式的设计难度较大，而且有时我们需要单独调用新增的业务方法。为了能够调用到新增方法，我们不得不用具体装饰类型来定义装饰之后的对象，而具体构件类型还是可以使用抽象构件类型来定义，这种装饰模式即为半透明装饰模式，也就是说，对于客户端而言，具体构件类型无须关心，是透明的；但是具体装饰类型必须指定，这是不透明的。如本节前面所提到的文件对象功能增加实例，为了能够调用到在Approver中新增方法approve()，客户端代码片段如下所示：   
……   
Document doc; //使用抽象构件类型定义   
doc = new PurchaseRequest();   
Approver newDoc; //使用具体装饰类型定义   
newDoc = new Approver(doc);   
……   
半透明装饰模式可以给系统带来更多的灵活性，设计相对简单，使用起来也非常方便；但是其最大的缺点在于不能实现对同一个对象的多次装饰，而且客户端需要有区别地对待装饰之前的对象和装饰之后的对象。在实现半透明的装饰模式时，我们只需在具体装饰类中增加一个独立的addedBehavior()方法来封装相应的业务处理，由于客户端使用具体装饰类型来定义装饰后的对象，因此可以单独调用addedBehavior()方法来扩展系统功能。

## 3. 优缺点

****优点****   
1、装饰者模式可以提供比继承更多的灵活性。   
2、可以通过一种动态的方式来扩展一个对象的功能，在运行时选择不同的装饰器，从而实现不同的行为。   
3、通过使用不同的具体装饰类以及这些装饰类的排列组合，可以创造出很多不同行为的组合。可以使用多个具体装饰类来装饰同一对象，得到功能更为强大的对象。   
4、具体构件类与具体装饰类可以独立变化，用户可以根据需要增加新的具体构件类和具体装饰类，在使用时再对其进行组合，原有代码无须改变，符合“开闭原则”。

****缺点****   
1、 会产生很多的小对象，增加了系统的复杂性。   
2、这种比继承更加灵活机动的特性，也同时意味着装饰模式比继承更加易于出错，排错也很困难，对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为烦琐。

## 4. 应用场景

(1) 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。

(2) 当不能采用继承的方式对系统进行扩展或者采用继承不利于系统扩展和维护时可以使用装饰模式。不能采用继承的情况主要有两类：第一类是系统中存在大量独立的扩展，为支持每一种扩展或者扩展之间的组合将产生大量的子类，使得子类数目呈爆炸性增长；第二类是因为类已定义为不能被继承（如Java语言中的final类）

## 5. 注意事项

(1) 尽量保持装饰类的接口与被装饰类的接口相同，这样，对于客户端而言，无论是装饰之前的对象还是装饰之后的对象都可以一致对待。这也就是说，在可能的情况下，我们应该尽量使用透明装饰模式。   
(2) 尽量保持具体构件类ConcreteComponent是一个“轻”类，也就是说不要把太多的行为放在具体构件类中，我们可以通过装饰类对其进行扩展。   
(3) 如果只有一个具体构件类，那么抽象装饰类可以作为该具体构件类的直接子类。

装饰模式降低了系统的耦合度，可以动态增加或删除对象的职责，并使得需要装饰的具体构件类和具体装饰类可以独立变化，以便增加新的具体构件类和具体装饰类。在软件开发中，装饰模式应用较为广泛，例如在JavaIO中的输入流和输出流的设计、javax.swing包中一些图形界面构件功能的增强等地方都运用了装饰模式。还比如jdk集合框架中的java.util.Collections#synchronizedList(List)。