

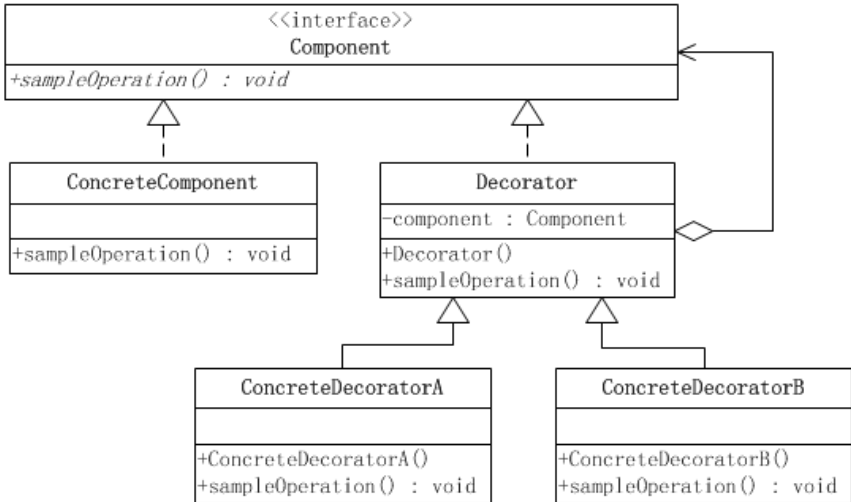
在闫宏博士的《JAVA与模式》一书中开头是这样描述装饰（Decorator）模式的：

装饰模式又名包装(Wrapper)模式。装饰模式以对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。

装饰模式的结构

装饰模式以对客户透明的方式动态地给一个对象附加上更多的责任。换言之，客户端并不会觉得对象在装饰前和装饰后有什么不同。装饰模式可以在不使用创造更多子类的情况下，将对象的功能加以扩展。

装饰模式的类图如下：



在装饰模式中的角色有：

- **抽象构件(Component)角色**：给出一个抽象接口，以规范准备接收附加责任的对象。
- **具体构件(ConcreteComponent)角色**：定义一个将要接收附加责任的类。
- **装饰(Decorator)角色**：持有一个构件(Component)对象的实例，并定义一个与抽象构件接口一致的接口。
- **具体装饰(ConcreteDecorator)角色**：负责给构件对象“贴上”附加的责任。

源代码

抽象构件角色

```
public interface Component {

    public void sampleOperation();

}
```

具体构件角色

```
public class ConcreteComponent implements Component {

    @Override
    public void sampleOperation() {
        // 写相关的业务代码
    }

}
```

装饰角色

```
public class Decorator implements Component{
    private Component component;

    public Decorator(Component component){
        this.component = component;
    }

    @Override
    public void sampleOperation() {
        // 委派给构件
        component.sampleOperation();
    }
}
```

具体装饰角色

```
public class ConcreteDecoratorA extends Decorator {

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

    @Override
    public void sampleOperation() {
        super.sampleOperation();
        // 写相关的业务代码
    }
}
```

```
public class ConcreteDecoratorB extends Decorator {

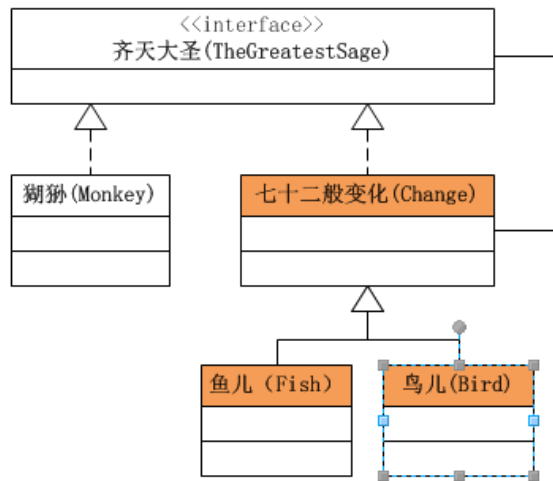
    public ConcreteDecoratorB(Component component) {
        super(component);
    }

    @Override
    public void sampleOperation() {
        super.sampleOperation();
        // 写相关的业务代码
    }
}
```

齐天大圣的例子

孙悟空有七十二般变化，他的每一种变化都给他带来一种附加的本领。他变成鱼儿时，就可以到水里游泳；他变成鸟儿时，就可以在天上飞行。

本例中，Component的角色便由鼎鼎大名的齐天大圣扮演；ConcreteComponent的角色属于大圣的本尊，就是猴孙本人；Decorator的角色由大圣的七十二变扮演。而ConcreteDecorator的角色便是鱼儿、鸟儿等七十二般变化。



源代码

抽象构件角色“齐天大圣”接口定义了一个move()方法，这是所有的具体构件类和装饰类必须实现的。

```
//大圣的尊号
public interface TheGreatestSage {

    public void move();
}
```

具体构件角色“大圣本尊”猴类

```
public class Monkey implements TheGreatestSage {

    @Override
    public void move() {
        //代码
        System.out.println("Monkey Move");
    }

}
```

抽象装饰角色“七十二变”

```
public class Change implements TheGreatestSage {
    private TheGreatestSage sage;

    public Change(TheGreatestSage sage){
        this.sage = sage;
    }
    @Override
    public void move() {
        // 代码
        sage.move();
    }

}
```

具体装饰角色“鱼儿”

```
public class Fish extends Change {
```

```

public Fish(TheGreatestSage sage) {
    super(sage);
}

@Override
public void move() {
    // 代码
    System.out.println("Fish Move");
}
}

```

具体装饰角色“鸟儿”

```

public class Bird extends Change {

    public Bird(TheGreatestSage sage) {
        super(sage);
    }

    @Override
    public void move() {
        // 代码
        System.out.println("Bird Move");
    }
}

```

客户端类

```

public class Client {

    public static void main(String[] args) {
        TheGreatestSage sage = new Monkey();
        // 第一种写法
        TheGreatestSage bird = new Bird(sage);
        TheGreatestSage fish = new Fish(bird);
        // 第二种写法
        //TheGreatestSage fish = new Fish(new Bird(sage));
        fish.move();
    }
}

```

“大圣本尊”是ConcreteComponent类，而“鸟儿”、“鱼儿”是装饰类。要装饰的是“大圣本尊”，也即“猢狲”实例。

上面的例子中，系统把大圣从一只猢狲装饰成了一只鸟儿（把鸟儿的功能加到了猢狲身上），然后又把鸟儿装饰成了一条鱼儿（把鱼儿的功能加到了猢狲+鸟儿身上，得到了猢狲+鸟儿+鱼儿）。

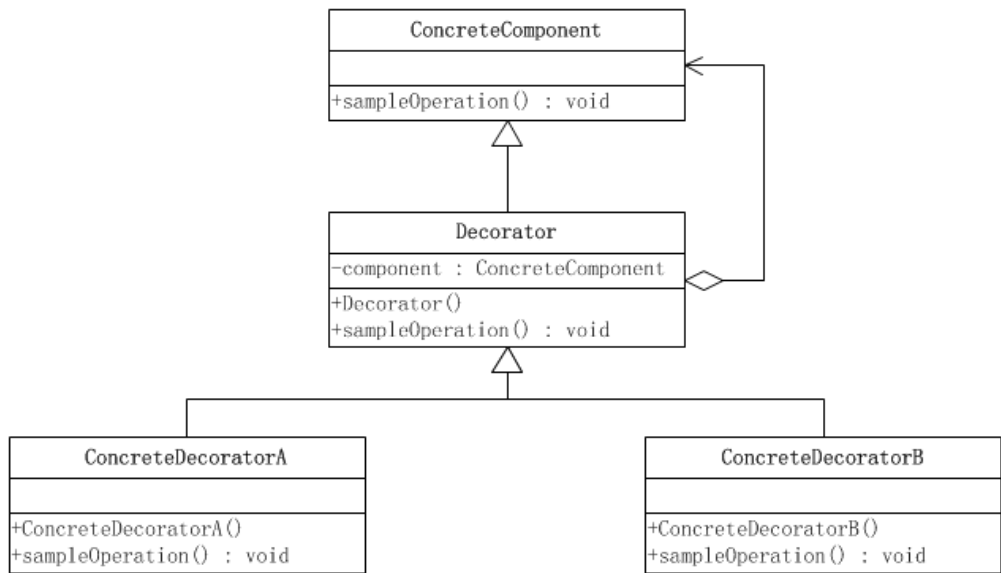


如上图所示，大圣的变化首先将鸟儿的功能附加到了猢狲身上，然后又将鱼儿的功能附加到猢狲+鸟儿身上。

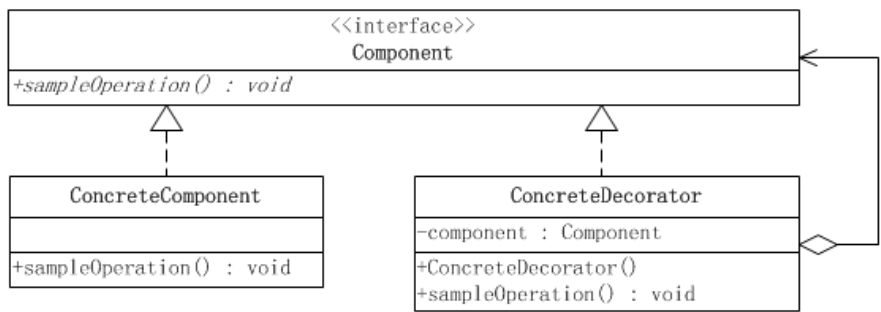
装饰模式的简化

大多数情况下，装饰模式的实现都要比上面给出的示意性例子要简单。

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。如下图所示：



如果只有一个ConcreteDecorator类，那么就没有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。甚至在只有两个ConcreteDecorator类的情况下，都可以这样做。如下图所示：



透明性的要求

装饰模式对客户端的透明性要求程序不要声明一个ConcreteComponent类型的变量，而应当声明一个Component类型的变量。

用孙悟空的例子来说，必须永远把孙悟空的所有变化都当成孙悟空来对待，而如果把老孙变成的鱼儿当成鱼儿，而不是老孙，那就被老孙骗了，而这时不应当发生的。下面的做法是对的：

```
TheGreatestSage sage = new Monkey();
TheGreatestSage bird = new Bird(sage);
```

而下面的做法是不对的：

```
Monkey sage = new Monkey();
Bird bird = new Bird(sage);
```

半透明的装饰模式

然而，纯粹的装饰模式很难找到。装饰模式的用意是在不改变接口的前提下，增强所考虑的类的性能。在增强性能的时候，往往需要建立新的公开的方法。即便是在孙大圣的系统里，也需要新的方法。比如齐天大圣类并没有飞行的能力，而鸟儿有。这就意味着鸟儿应当有一个新的fly()方法。再比如，齐天大圣类并没有游泳的能力，而鱼儿有，这就意味着在鱼儿类里应当有一个新的swim()方法。

这就导致了大多数的装饰模式的实现都是“半透明”的，而不是完全透明的。换言之，允许装饰模式改变接口，增加新的方法。这意味着客户端可以声明ConcreteDecorator类型的变量，从而可以调用ConcreteDecorator类中才有的方法：

```
TheGreatestSage sage = new Monkey();
Bird bird = new Bird(sage);
bird.fly();
```

半透明的装饰模式是介于装饰模式和适配器模式之间的。适配器模式的用意是改变所考虑的类的接口，也可以通过改写一个或几个方法，或增加新的方法来增强或改变所考虑的类的功能。大多数的装饰模式实际上是半透明的装饰模式，这样的装饰模式也称做半装饰、半适配器模式。

装饰模式的优点

(1) 装饰模式与继承关系的目都是要扩展对象的功能，但是装饰模式可以提供比继承更多的灵活性。装饰模式允许系统动态决定“贴上”一个需要的“装饰”，或者除掉一个不需要的“装饰”。继承关系则不同，继承关系是静态的，它在系统运行前就决定了。

(2) 通过使用不同的具体装饰类以及这些装饰类的排列组合，设计师可以创造出很多不同行为的组合。

装饰模式的缺点

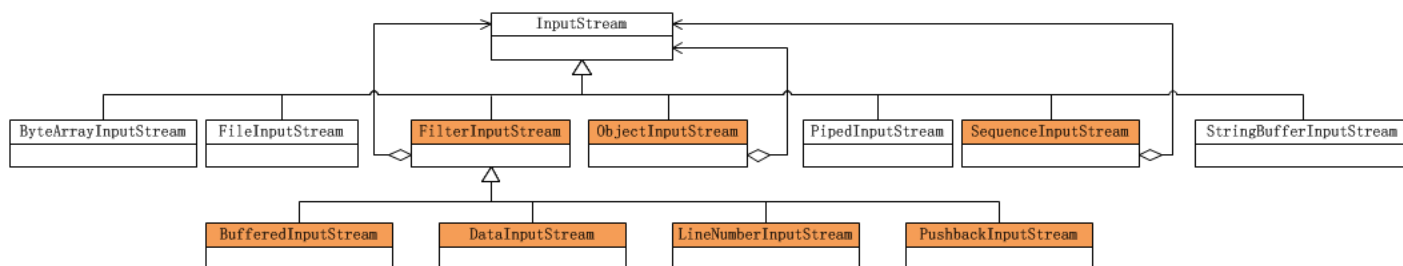
由于使用装饰模式，可以比使用继承关系需要较少数目的类。使用较少的类，当然使设计比较易于进行。但是，在另一方面，使用装饰模式会产生比使用继承关系更多的对象。更多的对象会使得查错变得困难，特别是这些对象看上去都很相像。

设计模式在JAVA I/O库中的应用

装饰模式在Java语言中的最著名应用莫过于Java I/O标准库的设计了。

由于Java I/O库需要很多性能的各种组合，如果这些性能都是用继承的方法实现的，那么每一种组合都需要一个类，这样就会造成大量性能重复的类出现。而如果采用装饰模式，那么类的数目就会大大减少，性能的重复也可以减至最少。因此装饰模式是Java I/O库的基本模式。

Java I/O库的对象结构图如下，由于Java I/O的对象众多，因此只画出InputStream的部分。



根据上图可以看出：

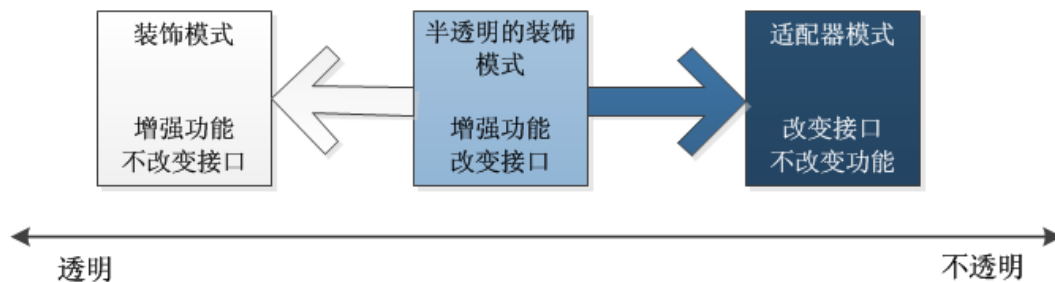
- **抽象构件(Component)角色**：由InputStream扮演。这是一个抽象类，为各种子类型提供统一的接口。
- **具体构件(ConcreteComponent)角色**：由ByteArrayInputStream、FileInputStream、PipedInputStream、StringBufferInputStream等类扮演。它们实现了抽象构件角色所规定的接口。
- **抽象装饰(Decorator)角色**：由FilterInputStream扮演。它实现了InputStream所规定的接口。
- **具体装饰(ConcreteDecorator)角色**：由几个类扮演，分别是BufferedInputStream、DataInputStream以及两个不常用到的类LineNumberInputStream、PushbackInputStream。

半透明的装饰模式

装饰模式和适配器模式都是“包装模式(Wrapper Pattern)”，它们都是通过封装其他对象达到设计的目的，但是它们的形态有很大区别。

理想的装饰模式在对被装饰对象进行功能增强的同时，要求具体构件角色、装饰角色的接口与抽象构件角色的接口完全一致。而适配器模式则不然，一般而言，适配器模式并不要求对源对象的功能进行增强，但是会改变源对象的接口，以便和目标接口相符合。

装饰模式有透明和半透明两种，这两种的区别就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角色的接口完全一致。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，如下图所示。




在适配器模式里面，适配器类的接口通常会与目标类的接口重叠，但往往并不完全相同。换言之，适配器类的接口会被比被装饰的目标类接口宽。

显然，半透明的装饰模式实际上就是处于适配器模式与装饰模式之间的灰色地带。如果将装饰模式与适配器模式合并成为一个“包装模式”的话，那么半透明的装饰模式倒可以成为这种合并后的“包装模式”的代表。

InputStream类型中的装饰模式

InputStream类型中的装饰模式是半透明的。为了说明这一点，不妨看一看装饰模式的抽象构件角色的InputStream的源代码。这个抽象类声明了九个方法，并给出了其中八个的实现，另外一个抽象方法，需要由子类实现。



```
public abstract class InputStream implements Closeable {

    public abstract int read() throws IOException;

    public int read(byte b[]) throws IOException {}

    public int read(byte b[], int off, int len) throws IOException {}

    public long skip(long n) throws IOException {}

    public int available() throws IOException {}


    public void close() throws IOException {}

    public synchronized void mark(int readlimit) {}


    public synchronized void reset() throws IOException {}

    public boolean markSupported() {}

}
```



下面是作为装饰模式的抽象装饰角色FilterInputStream类的源代码。可以看出，FilterInputStream的接口与InputStream的接口是完全一致的。也就是说，直到这一步，还是与装饰模式相符合的。



```
public class FilterInputStream extends InputStream {
    protected FilterInputStream(InputStream in) {}

    public int read() throws IOException {}

    public int read(byte b[]) throws IOException {}

    public int read(byte b[], int off, int len) throws IOException {}

    public long skip(long n) throws IOException {}

    public int available() throws IOException {}


    public void close() throws IOException {}

    public synchronized void mark(int readlimit) {}


    public synchronized void reset() throws IOException {}

    public boolean markSupported() {}

}
```



下面是具体装饰角色PushbackInputStream的源代码。



```
public class PushbackInputStream extends FilterInputStream {
    private void ensureOpen() throws IOException {}

    public PushbackInputStream(InputStream in, int size) {}

    public PushbackInputStream(InputStream in) {}

    public int read() throws IOException {}

}
```

```

public int read(byte[] b, int off, int len) throws IOException {}

public void unread(int b) throws IOException {}

public void unread(byte[] b, int off, int len) throws IOException {}

public void unread(byte[] b) throws IOException {}

public int available() throws IOException {}

public long skip(long n) throws IOException {}

public boolean markSupported() {}

public synchronized void mark(int readlimit) {}

public synchronized void reset() throws IOException {}

public synchronized void close() throws IOException {}
}

```



查看源码，你会发现，这个装饰类提供了额外的方法unread()，这就意味着PushbackInputStream是一个半透明的装饰类。换言之，它破坏了理想的装饰模式的要求。如果客户端持有一个类型为InputStream对象的引用in的话，那么如果in的真实类型是 PushbackInputStream的话，只要客户端不需要使用unread()方法，那么客户端一般没有问题。但是如果客户端必须使用这个方法，就必须进行向下类型转换。将in的类型转换为PushbackInputStream之后才可能调用这个方法。但是，这个类型转换意味着客户端必须知道它拿到的引用是指向一个类型为PushbackInputStream的对象。这就破坏了使用装饰模式的原始用意。

现实世界与理论总归是有一段差距的。纯粹的装饰模式在真实的系统中很难找到。一般所遇到的，都是这种半透明的装饰模式。

下面是使用I/O流读取文件内容的简单操作示例。



```

public class IOTest {

    public static void main(String[] args) throws IOException {
        // 流式读取文件
        DataInputStream dis = null;
        try{
            dis = new DataInputStream(
                new BufferedInputStream(
                    new FileInputStream("test.txt")
                )
            );
            //读取文件内容
            byte[] bs = new byte[dis.available()];
            dis.read(bs);
            String content = new String(bs);
            System.out.println(content);
        }finally{
            dis.close();
        }
    }
}

```



观察上面的代码，会发现最里层是一个FileInputStream对象，然后把它传递给一个BufferedInputStream对象，经过BufferedInputStream处理，再把处理后的对象传递给了DataInputStream对象进行处理，这个过程其实就是装饰器的组装过程，FileInputStream对象相当于原始的被装饰的对象，而BufferedInputStream对象和DataInputStream对象则相当于装饰器。