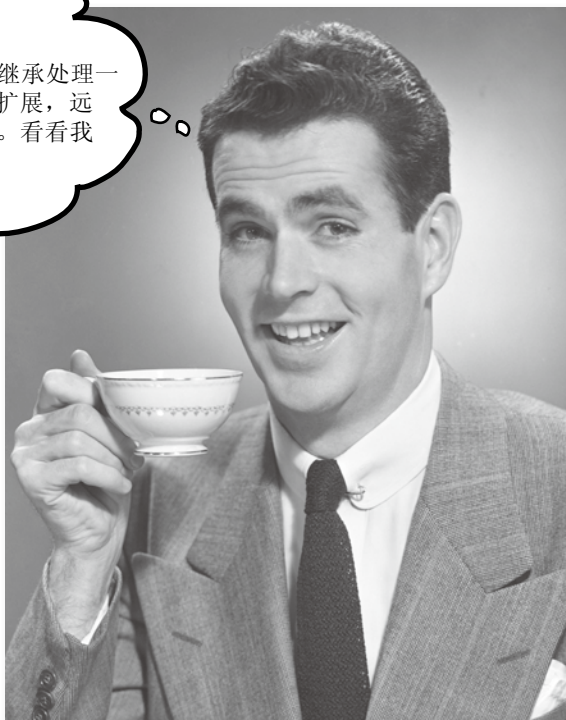


### 3 装饰者模式

## 装饰对象

我曾经以为男子汉应该用继承处理一切。后来我领教到运行时扩展，远比编译时期的继承威力大。看看我现在光采的样子！



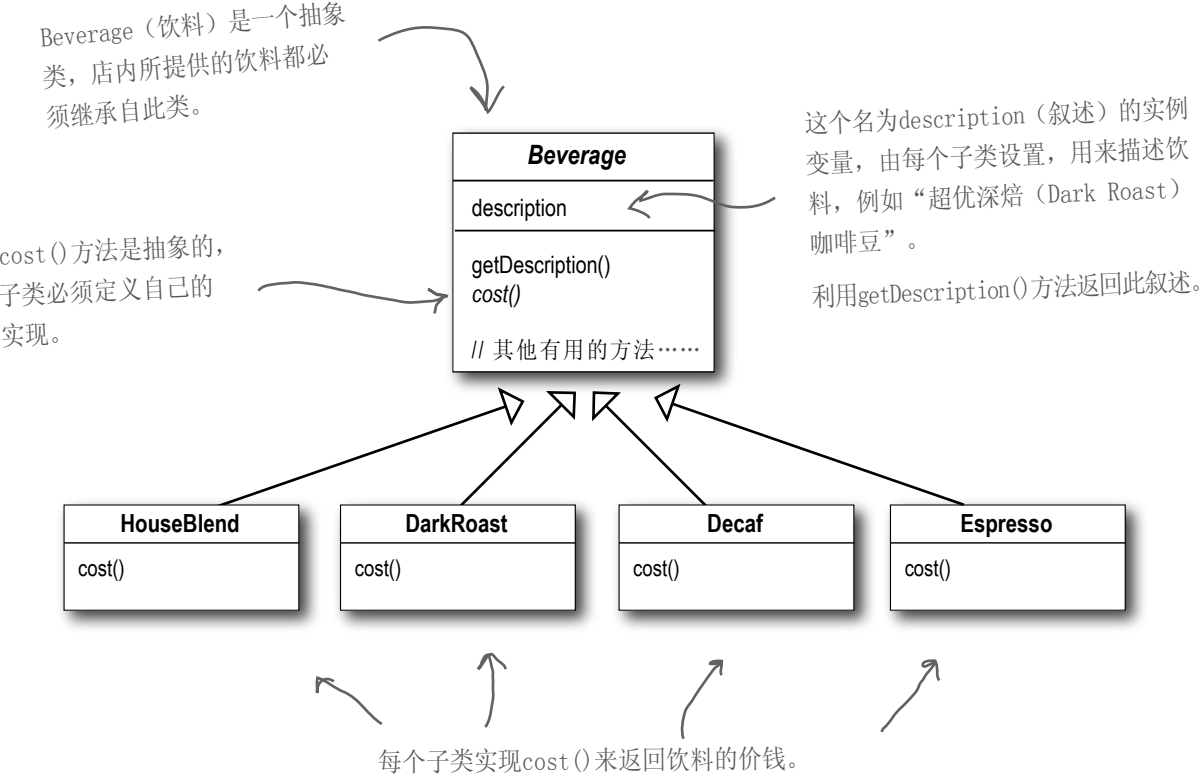
本章可以称为“给爱用继承的人一个全新的设计眼界”。我们即将再度探讨典型的继承滥用问题。你将在本章学到如何使用对象组合的方式，做到在运行时装饰类。为什么呢？一旦你熟悉了装饰的技巧，你将能够在不修改任何底层代码的情况下，给你的（或别人的）对象赋予新的职责。

# 欢迎来到星巴兹咖啡

星巴兹（Starbuzz）是以扩张速度最快而闻名的咖啡连锁店。如果你在街角看到它的店，在对面街上肯定还会看到另一家。

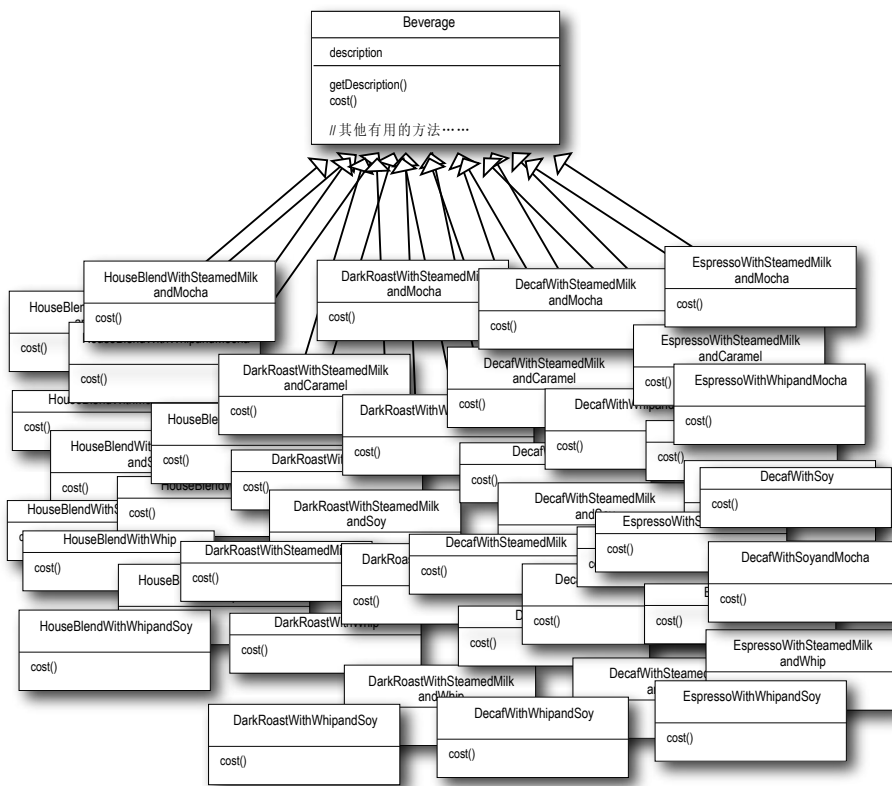
因为扩张速度实在太快了，他们准备更新订单系统，以合乎他们的饮料供应要求。

他们原先的类设计是这样的……



购买咖啡时，也可以要求在其中加入各种调料，例如：蒸奶（Steamed Milk）、豆浆（Soy）、摩卡（Mocha，也就是巧克力风味）或覆盖奶泡。星巴兹会根据所加入的调料收取不同的费用。所以订单系统必须考虑到这些调料部分。

这是他们的第一个尝试……



每个cost()方法将计算出咖啡加上订单上各种调料的价钱。



很明显，星巴兹为自己制造了一个维护恶梦。如果牛奶的价钱上扬，怎么办？新增一种焦糖调料风味时，怎么办？

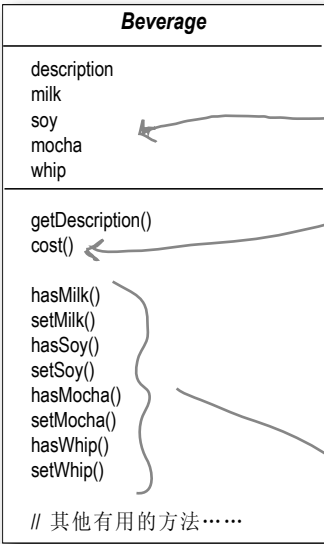
造成这种维护上的困难，究竟违反了我们之前提过的哪种设计原则？

提示：违反了两个原则，而且很严重！

笨透了！干嘛设计这么多类呀？利用实例变量和继承，就可以追踪这些调料呀！



好吧！就来试试看。先从Beverage基类下手，加上实例变量代表是否加上调料（牛奶、豆浆、摩卡、奶泡……）



各种调料的新的布尔值

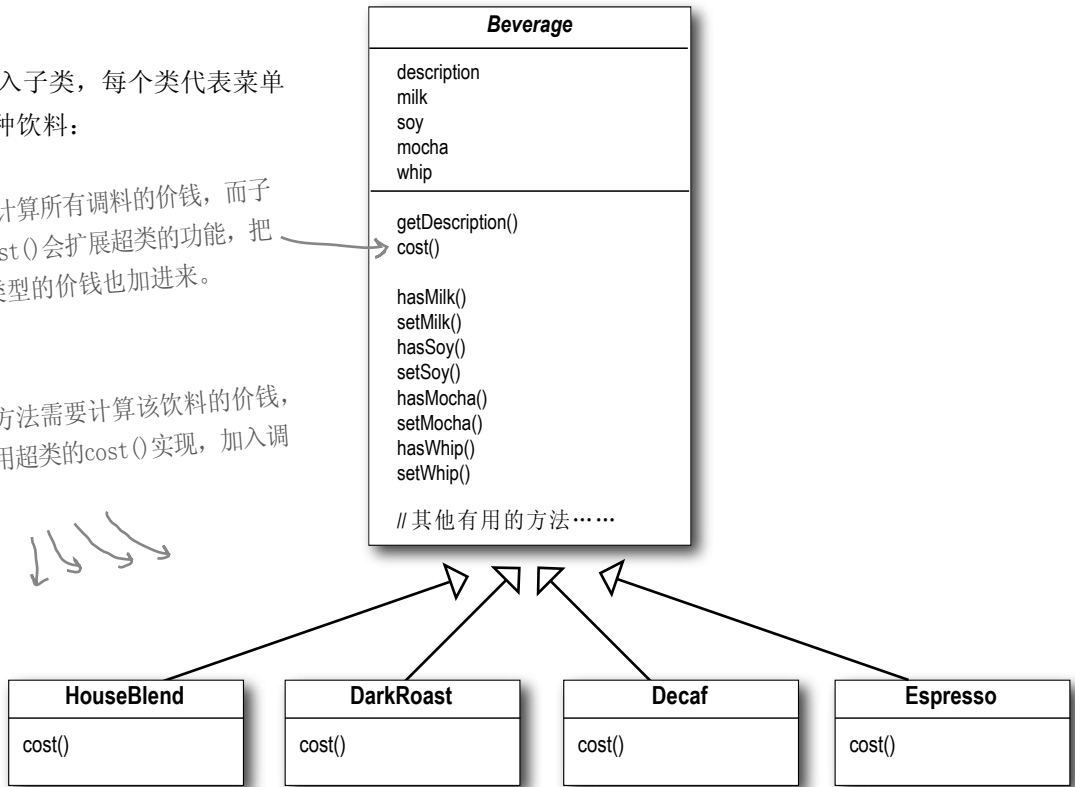
现在，Beverage类中的cost()不再是一个抽象方法，我们提供了cost()的实现，让它计算要加入各种饮料的调料价钱。子类仍将覆盖cost()，但是会调用超类的cost()，计算出基本饮料加上调料的价钱。

这些方法取得和设置调料的布尔值。

现在加入子类，每个类代表菜单上的一种饮料：

超类cost()将计算所有调料的价钱，而子类覆盖过的cost()会扩展超类的功能，把指定的饮料类型的价钱也加进来。

每个cost()方法需要计算该饮料的价钱，然后通过调用超类的cost()实现，加入调料的价钱。



## Sharpen your pencil

请为下面类的cost()方法书写代码（用伪Java代码即可）。

```

public class Beverage {
    public double cost() {
        }
    }

```

```

public class DarkRoast extends Beverage {
    public DarkRoast() {
        description = "Most Excellent Dark Roast";
    }
    public double cost() {
        }
    }

```



看吧！一共只需要五个类，这正是我们要的做法。



我不确定耶！通过思考设计将来可能需要的变化，我可以看出来这种方法有一些潜在的问题。

## Sharpen your pencil

当哪些需求或因素改变时会影响这个设计？

调料价钱的变化会使我们更改现有代码。

一旦出现新的调料，我们就需要加上新的方法，并改变超类中的`cost()`方法。

以后可能会开发出新饮料。对这些饮料而言（例如：冰茶），某些调料可能并不适合，但是在这个设计方式中，`Tea`（茶）子类仍将继承那些不适合的方法，例如：`hasWhip()`（加奶泡）。

万一顾客想要双倍摩卡咖啡，怎么办？

轮到你了：

这是很糟糕的！我们在第1章就得到了这个教训。



### 大师与门徒……

大师：我说蚱蜢呀！距离我们上次见面已经有些时日，你对于继承的冥想，可有精进？

门徒：是的，大师。尽管继承威力强大，但是我体会到它并不总是能够实现最有弹性和最好维护的设计。

大师：啊！是的，看来你已经有所长进。那么，告诉我，我的门徒，不通过继承又能如何达到复用呢？

门徒：大师，我已经了解到利用组合（composition）和委托（delegation）可以在运行时具有继承行为的效果。

大师：好，好，继续……

门徒：利用继承设计子类的行为，是在编译时静态决定的，而且所有的子类都会继承到相同的行为。然而，如果能够利用组合的做法扩展对象的行为，就可以在运行时动态地进行扩展。

大师：很好，蚱蜢，你已经开始看到组合的威力了。

门徒：是的，我可以利用此技巧把多个新职责，甚至是设计超类时还没有想到的职责加在对象上。而且，可以不用修改原来的代码。

大师：利用组合维护代码，你认为效果如何？

门徒：这正是我要说的。通过动态地组合对象，可以写新的代码添加新功能，而无须修改现有代码。既然没有改变现有代码，那么引进bug或产生意外副作用的机会将大幅度减少。

大师：非常好。蚱蜢，今天的谈话就到这里。希望你能在这个主题上更深入……牢记，代码应该如同晚霞中的莲花一样地关闭（免于改变），如同晨曦中的莲花一样地开放（能够扩展）。

# 开放-关闭原则

此刻，蚱蜢面临最重要的设计原则之一：



设计原则

类应该对扩展开放，对修改关闭。



请进，现在“开放”中。欢迎用任何你想要的行为来扩展我们的类。如果你的需要或需求有所改变（我们知道这一定会发生的），那就来吧！动手扩展吧！



抱歉，现在是“关闭”状态。没错。我们花了许多时间得到了正确的代码，还解决了所有的bug，所以不能让你修改现有代码。我们必须关闭代码以防止被修改。如果你不喜欢，可以找经理谈。

我们的目标是允许类容易扩展，在不修改现有代码的情况下，就可搭配新的行为。如能实现这样的目标，有什么好处呢？这样的设计具有弹性可以应对改变，可以接受新的功能来应对改变的需求。



## there are no Dumb Questions

**问：** 对扩展开放，对修改关闭？听起来很矛盾。设计如何兼顾两者？

**答：** 这是一个很好的问题。乍听之下，的确感到矛盾，毕竟，越难修改的事物，就越难以扩展，不是吗？

但是，有一些聪明的OO技巧，允许系统在不修改代码的情况下，进行功能扩展。想想观察者模式（在第2章）……通过加入新的观察者，我们可以在任何时候扩展Subject（主题），而且不需向主题中添加代码。以后，你还会陆续看到更多的扩展行为的其他OO设计技巧。

**问：** 好吧！我了解观察者（Observable），但是该如何将某件东西设计成可以扩展，又禁止修改？

**答：** 许多模式是长期经验的实证，可通过提供扩展的方法来保护代码免于被修改。在本章，将看到使用装饰者模式的一个好例子，完全遵循开放-关闭原则。

**问：** 我如何让设计的每个部分都遵循开放-关闭原则？

**答：** 通常，你办不到。要让OO设计同时具备开放性和关闭性，又不修改现有的代码，需要花费许多时间和努力。一般来说，我们实在没有闲工夫把设计的每个部分都这么设计（而且，就算做得到，也可能只是一种浪费）。遵循开放-关闭原则，通常会引入新的抽象层次，增加代码的复杂度。你需要把注意力集中在设计中最有可能改变的地方，然后应用开放-关闭原则。

**问：** 我怎么知道，哪些地方的改变是更重要呢？

**答：** 这牵涉到设计OO系统的经验，和对你工作领域的了解。多看一些其他的例子可以帮你学习如何辨别设计中的变化区。

虽然似乎有点矛盾，但是的确有一些技术可以允许在不直接修改代码的情况下对其进行扩展。

在选择需要被扩展的代码部分时要小心。每个地方都采用开放-关闭原则，是一种浪费，也没必要，还会导致代码变得复杂且难以理解。

## 认识装饰者模式

好了，我们已经了解利用继承无法完全解决问题，在星巴兹遇到的问题有：类数量爆炸、设计死板，以及基类加入的新功能并不适用于所有的子类。

所以，在这里要采用不一样的做法：我们要以饮料为主体，然后在运行时以调料来“装饰”（decorate）饮料。比方说，如果顾客想要摩卡和奶泡深焙咖啡，那么，要做的是：

- ❶ 拿一个深焙咖啡（DarkRoast）对象
- ❷ 以摩卡（Mocha）对象装饰它
- ❸ 以奶泡（Whip）对象装饰它
- ❹ 调用cost()方法，并依赖委托（delegate）将调料的价格加上去。

好了！但是如何“装饰”一个对象，而“委托”又要如何与此搭配使用呢？给一个暗示：把装饰者对象当成“包装者”。让我们看看这是如何工作的……

够了！你们这些“面向对象设计俱乐部”的家伙。快来解决真正的问题吧！还记得我们吗？星巴兹咖啡？你认为这些设计原则有实质的帮助吗？



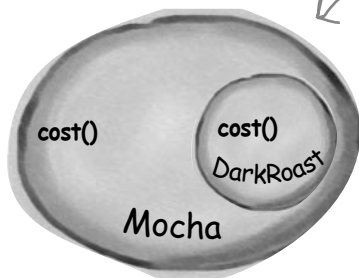
# 以装饰者构造饮料订单

## 1 以DarkRoast对象开始



别忘了，DarkRoast继承自Beverage，且有一个用来计算饮料价钱的cost()方法。

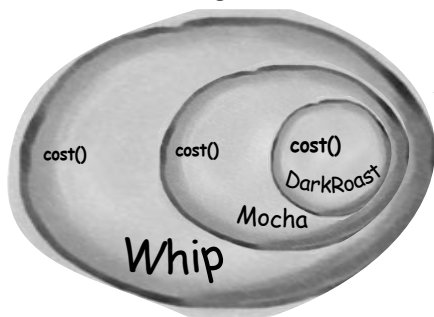
## 2 顾客想要摩卡（Mocha），所以建立一个Mocha对象，并用它将DarkRoast对象包（wrap）起来。



Mocha对象是一个装饰者，它的类型“反映”了它所装饰的对象（本例中，就是Beverage）。所谓的“反映”，指的就是两者类型一致。

所以Mocha也有一个cost()方法。通过多态，也可以把Mocha所包裹的任何Beverage当成是Beverage（因为Mocha是Beverage的子类型）。

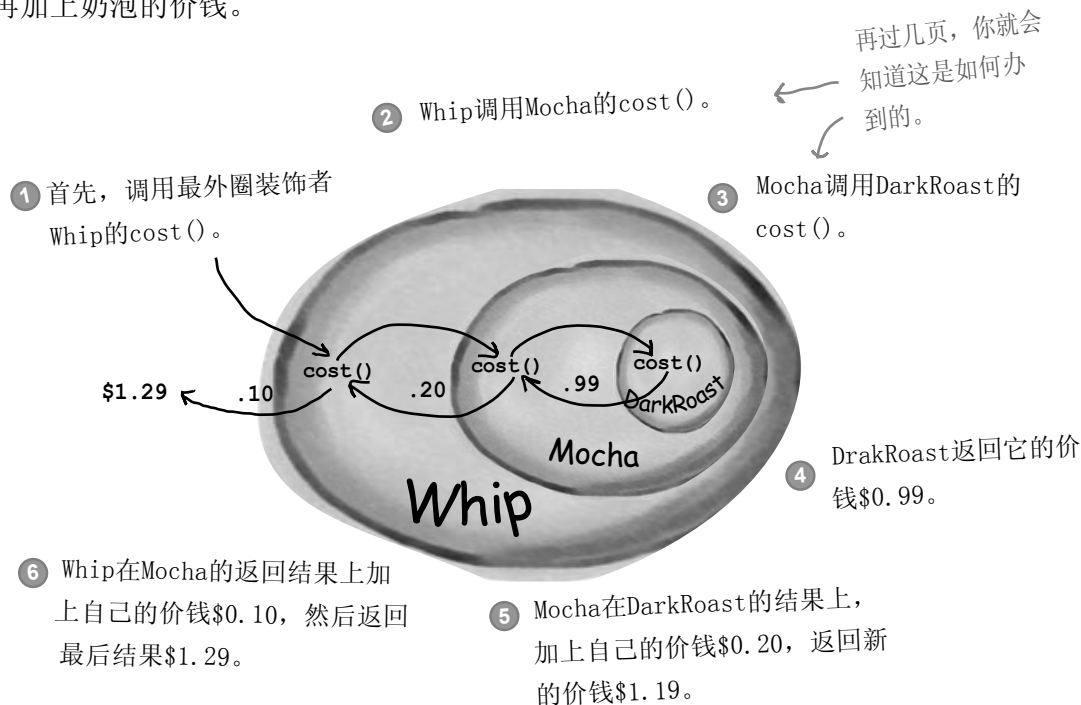
## 3 顾客也想要奶泡（Whip），所以需要建立一个Whip装饰者，并用它将Mocha对象包起来。别忘了，DarkRoast继承自Beverage，且有一个cost()方法，用来计算饮料价钱。



Whip是一个装饰者，所以它也反映了DarkRoast类型，并包括一个cost()方法。

所以，被Mocha和Whip包起来的DarkRoast对象仍然是一个Beverage，仍然可以具有DarkRoast的一切行为，包括调用它的cost()方法。

- 4 现在，该是为顾客算钱的时候了。通过调用最外圈装饰者（Whip）的cost()就可以办得到。Whip的cost()会先委托它装饰的对象（也就是Mocha）计算出价钱，然后再加上奶泡的价钱。



好了，这是目前所知道的一切……

- 装饰者和被装饰对象有相同的超类型。
  - 你可以用一个或多个装饰者包装一个对象。
  - 既然装饰者和被装饰对象有相同的超类型，所以在任何需要原始对象（被包装的）的场合，可以用装饰过的对象代替它。
  - 装饰者可以在所委托被装饰者的行为之前与/或之后，加上自己的行为，以达到特定的目的。
  - 对象可以在任何时候被装饰，所以可以在运行时动态地、不限量地用你喜欢的装饰者来装饰对象。
- 关键点！

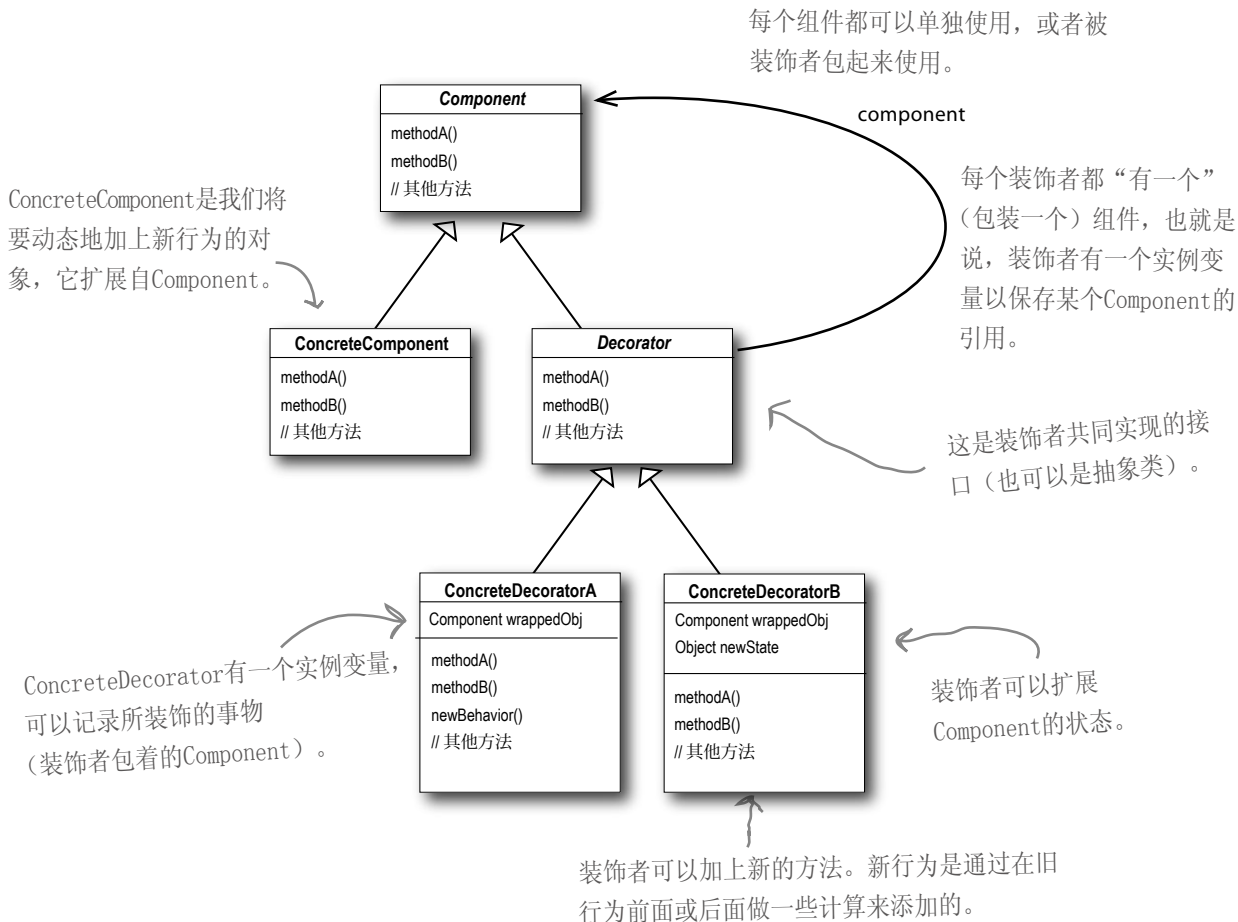
现在，就来看看装饰者模式的定义，并写一些代码，了解它到底是怎么工作的。

# 定义装饰者模式

让我们先来看看装饰者模式的说明：

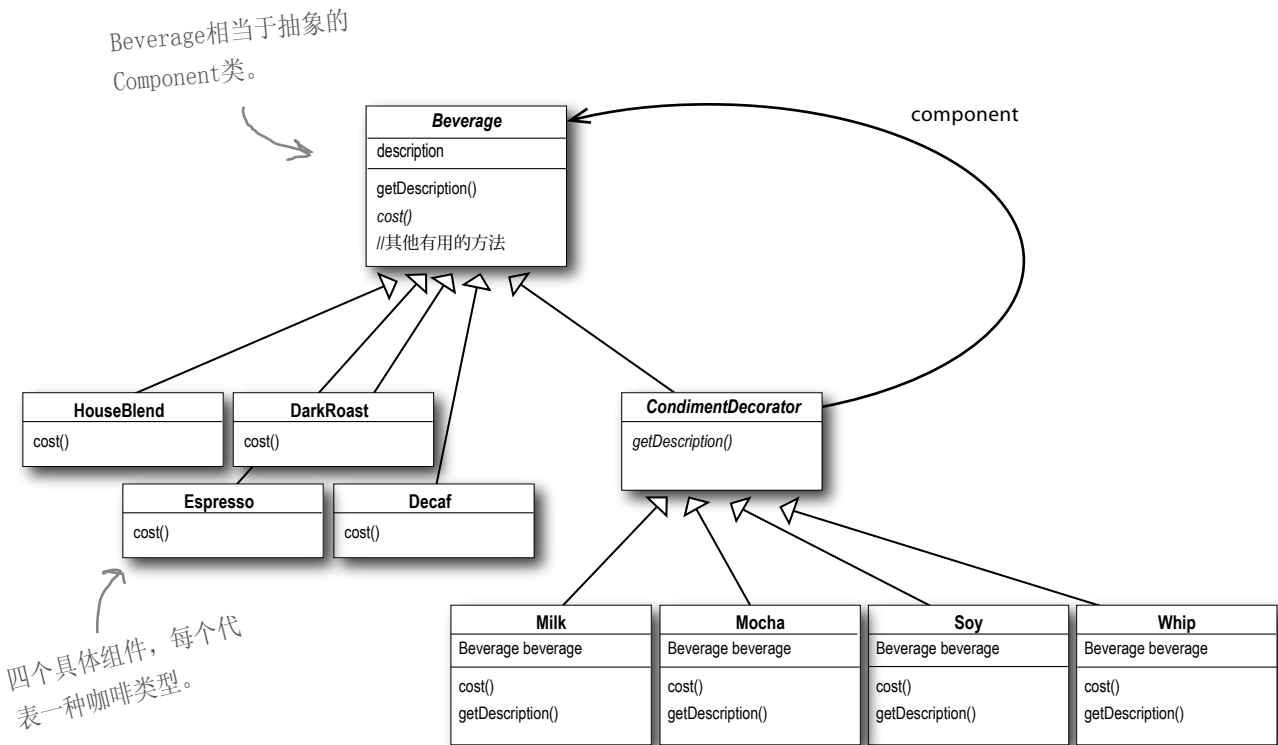
装饰者模式动态地将责任附加到对象上。  
若要扩展功能，装饰者提供了比继承更有弹性的替代方案。

虽然这说明了装饰者模式的“角色”，但是没说明怎么在我们的实现中实际“应用”它。我们来看看类图，会有些帮助（下一页，我们会将此结构套用在饮料问题上）。



# 装饰我们的饮料

好吧！让星巴兹饮料也能符合此框架……



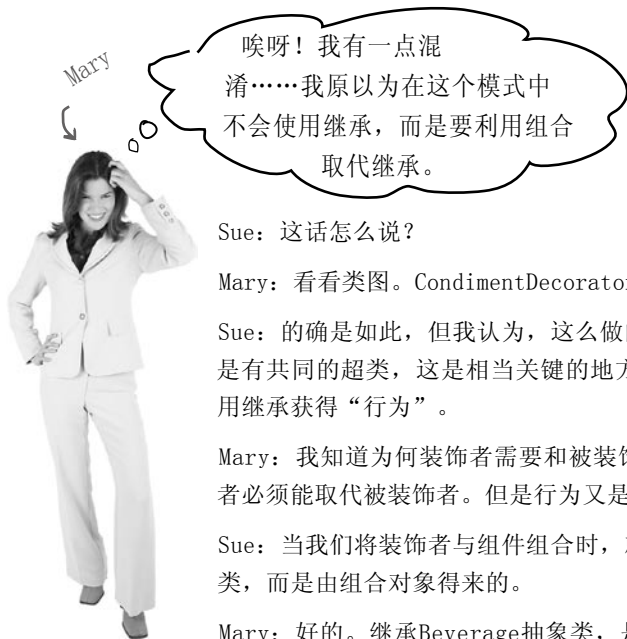
这是调料装饰者。请注意，它们除了必须实现cost()之外，还必须实现getDescription()。稍后我们会解释为什么……



在往下看之前，想想如何实现咖啡和调料的cost()方法。也思考一下如何实现调料的getDescription()方法。

# 办公室隔间对话

在继承和组合之间，观念有一些混淆。



Sue: 这话怎么说？

Mary: 看看类图。CondimentDecorator扩展自Beverage类，这用到了继承，不是吗？

Sue: 的确是如此，但我认为，这么做的重点在于，装饰者和被装饰者必须是一样的类型，也就是有共同的超类，这是相当关键的地方。在这里，我们利用继承达到“类型匹配”，而不是利用继承获得“行为”。

Mary: 我知道为何装饰者需要和被装饰者（亦即被包装的组件）有相同的“接口”，因为装饰者必须能取代被装饰者。但是行为又是从哪里来的？

Sue: 当我们将装饰者与组件组合时，就是在加入新的行为。所得到的新行为，并不是继承自超类，而是由组合对象得来的。

Mary: 好的。继承Beverage抽象类，是为了有正确的类型，而不是继承它的行为。行为来自装饰者和基础组件，或与其他装饰者之间的组合关系。

Sue: 正是如此。

Mary: 哦！我明白了。而且因为使用对象组合，可以把所有饮料和调料更有弹性地加以混和与匹配，非常方便。

Sue: 是的。如果依赖继承，那么类的行为只能在编译时静态决定。换句话说，行如果不是来自超类，就是子类覆盖后的版本。反之，利用组合，可以把装饰者混合着用……而且是在“运行时”。

Mary: 而且，如我所理解的，我们可以在任何时候，实现新的装饰者增加新的行为。如果依赖继承，每当需要新行为时，还得修改现有的代码。

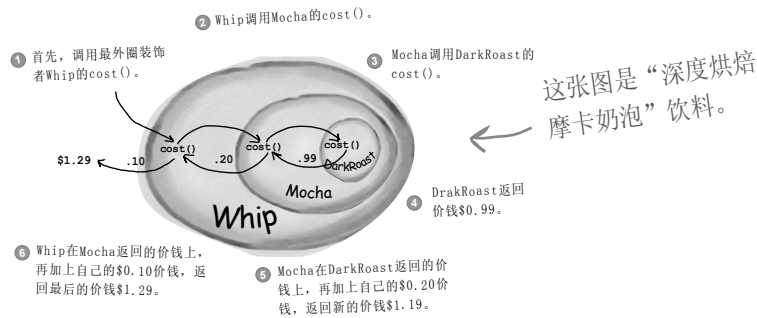
Sue: 的确如此。

Mary: 我还剩下一个问题，如果我们需要继承的是component类型，为什么不Beverage类设计成一个接口，而是设计成一个抽象类呢？

Sue: 关于这个嘛，还记得吗？当初我们从星巴兹拿到这个程序时，Beverage“经”是一个抽象类了。通常装饰者模式是采用抽象类，但是在Java中可以使用接口。尽管如此，通常我们都努力避免修改现有的代码，所以，如果抽象类运作得好好的，还是别去修改它。

# 新咖啡师傅特训

如果有一张单子点的是：“双倍摩卡豆浆奶泡拿铁咖啡”，请使用菜单得到正确的价钱并画一个图来表达你的设计，采用和几页前一样的格式。



OK，我要一杯“双倍摩卡豆浆奶泡拿铁咖啡”。

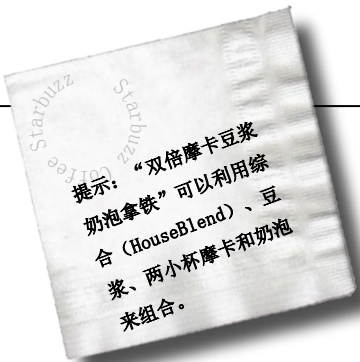


Sharpen your pencil

把图画在这里

## 星巴兹咖啡

咖啡	
综合	.89
深焙	.99
低咖啡因	1.05
浓缩	1.99
配料	
牛奶	.10
摩卡	.20
豆浆	.15
奶泡	.10





## 写下星巴兹的代码



该是把设计变成真正的代码的时候了！

先从Beverage类下手，这不需要改变星巴兹原始的设计。如下所示：

```
public abstract class Beverage {  
    String description = "Unknown Beverage";  
  
    public String getDescription() {  
        return description;  
    }  
  
    public abstract double cost();  
}
```

Beverage是一个抽象类，有两个方法：getDescription()及cost()。

getDescription()已经在此实现了，但是cost()必须在子类中实现。

Beverage很简单。让我们也来实现Condiment（调料）抽象类，也就是装饰者类吧：

```
public abstract class CondimentDecorator extends Beverage {  
    public abstract String getDescription();  
}
```

首先，必须让Condiment Decorator能够取代Beverage，所以将Condiment Decorator扩展自 Beverage 类。

所有的调料装饰者都必须重新实现getDescription()方法。稍后我们会解释为什么……

# 写饮料的代码

现在，已经有了基类，让我们开始实现一些饮料吧！先从浓缩咖啡（Espresso）开始。别忘了，我们需要为具体的饮料设置描述，而且还必须实现cost()方法。

```
public class Espresso extends Beverage {  
  
    public Espresso() {  
        description = "Espresso";  
    }  
  
    public double cost() {  
        return 1.99;  
    }  
}
```

首先，让Espresso扩展自Beverage类，因为Espresso是一种饮料。

为了要设置饮料的描述，我们写了一个构造器。记住，description实例变量继承自Beverage。

最后，需要计算Espresso的价钱，现在不需要管调料的价钱，直接把Espresso的价格\$1.99返回即可。

```
public class HouseBlend extends Beverage {  
    public HouseBlend() {  
        description = "House Blend Coffee";  
    }  
  
    public double cost() {  
        return .89;  
    }  
}
```

这是另一种饮料，做法和Espresso一样，只是把Espresso名称改为“HouseBlend Coffee”，并返回正确的价钱\$0.89。

你可以自行建立另外两种饮料类（DarkRoast和Decaf），做法都一样。

星巴兹咖啡	
咖啡	.89
综合	.99
深焙	1.05
低咖啡因	1.99
浓缩	
配料	.10
牛奶	.20
摩卡	.15
豆浆	.10
奶泡	

## 写调料代码

如果你回头去看看装饰者模式的类图，将发现我们已经完成了抽象组件（Beverage），有了具体组件（HouseBlend），也有了抽象装饰者（CondimentDecorator）。现在，我们就来实现具体装饰者。先从摩卡下手：

摩卡是一个装饰者，所以让它扩展自CondimentDecorator。

别忘了，CondimentDecorator扩展自Beverage。

要让Mocha能够引用一个Beverage，做法如下：

- (1) 用一个实例变量记录饮料，也就是被装饰者。
- (2) 想办法让被装饰者（饮料）被记录到实例变量中。这里的做法是：把饮料当作构造器的参数，再由构造器将此饮料记录在实例变量中。

```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }

    public double cost() {
        return .20 + beverage.cost();
    }
}
```

我们希望叙述不只是描述饮料（例如“DarkRoast”），而是完整地连调料都描述出来（例如“DarkRoast, Mocha”）。所以首先利用委托的做法，得到一个叙述，然后在其后加上附加的叙述（例如“Mocha”）。

要计算带Mocha饮料的价钱。首先把调用委托给被装饰对象，以计算价钱，然后再加上Mocha的价钱，得到最后结果。

在下一页，我们会实际实例化一个饮料对象，然后用各种调料（装饰者）包装它。但是，在这么做之前，首先……



写下Soy和Whip调料的代码，并完成编译。你需要它们，否则将无法进行下一页的程序。

# 供应咖啡

恭喜你，是时候舒服地坐下来，点一些咖啡，看看你利用装饰者模式设计出的灵活系统是多么神奇了。

这是用来下订单的一些测试代码★：

```
public class StarbuzzCoffee {  
  
    public static void main(String args[]) {  
        Beverage beverage = new Espresso();  
        System.out.println(beverage.getDescription()  
            + " $" + beverage.cost());  
  
        Beverage beverage2 = new DarkRoast();  
        beverage2 = new Mocha(beverage2);  
        beverage2 = new Mocha(beverage2);  
        beverage2 = new Whip(beverage2);  
        System.out.println(beverage2.getDescription()  
            + " $" + beverage2.cost());  
  
        Beverage beverage3 = new HouseBlend();  
        beverage3 = new Soy(beverage3);  
        beverage3 = new Mocha(beverage3);  
        beverage3 = new Whip(beverage3);  
        System.out.println(beverage3.getDescription()  
            + " $" + beverage3.cost());  
    }  
}
```



订一杯Espresso，不需要调料，打印出它的描述与价钱。

制造出一个DarkRoast对象。

用Mocha装饰它。

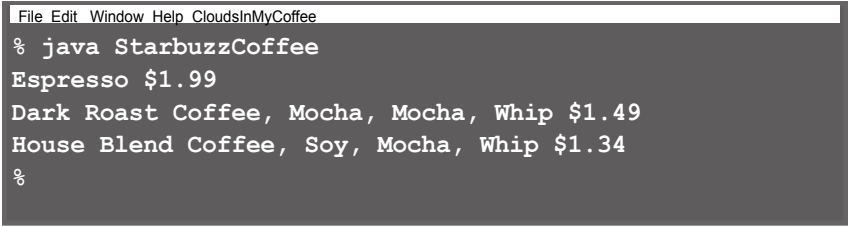
用第二个Mocha装饰它。

用Whip装饰它。

最后，再来一杯调料为豆浆、摩卡、奶泡的HouseBlend咖啡。

★当我们介绍到“工厂”和“生成器”设计模式时，将有更好的方式建立被装饰者对象。注意，关于“生成器模式”请参考本书附录。

现在，来看看实验结果：



```
File Edit Window Help CloudsInMyCoffee  
% java StarbuzzCoffee  
Espresso $1.99  
Dark Roast Coffee, Mocha, Mocha, Whip $1.49  
House Blend Coffee, Soy, Mocha, Whip $1.34  
%
```

## there are no Dumb Questions

**问：** 如果我将代码针对特定种类的具体组件（例如HouseBlend），做一些特殊的事（例如，打折），我担心这样的设计是否恰当。因为一旦用装饰者包装HouseBlend，就会造成类型改变。

**答：** 的确是这样。如果你把代码写成依赖于具体的组件类型，那么装饰者就会导致程序出问题。只有在针对抽象组件类型编程时，才不会因为装饰者而受到影响。但是，如果的确针对特定的具体组件编程，就应该重新思考你的应用架构，以及装饰者是否适合。

**问：** 对于使用到饮料的某些客户来说，会不会容易不使用最外圈的装饰者呢？比方说，如果我有深焙咖啡，以摩卡、豆浆、奶泡来装饰，

引用到豆浆而不是奶泡，代码会好写一些，这意味着订单里没有奶泡了。

**答：** 你当然可以争辩说，使用装饰者模式，你必须管理更多的对象，所以犯下你所说的编码错误的机会会增加。但是，装饰者通常是用其他类似于工厂或生成器这样的模式创建的。一旦我们讲到这两个模式，你就会明白具体的组件及其装饰者的创建过程，它们会“封装得很好”，所以不会有这种问题。

**问：** 装饰者知道这一连串装饰链条中其他装饰者的存在吗？比方说，我想要让getDescription()列出“Whip, Double Mocha”而不是“Mocha, Whip, Mocha”，这需要最外圈的装饰者知道有哪些装饰者牵涉其中了。

**答：** 装饰者该做的事，就是增加行为到被包装对象上。当需要窥视装饰者链中的每一个装饰者时，这就超出他们的天赋了。但是，并不是做不到。可以写一个CondimentPrettyPrint装饰者，解析出最后的描述字符串，然后把“Mocha, Whip, Mocha”变成“Whip, Double Mocha”。如果能把getDescription()的返回值变成ArrayList类型，让每个调料名称独立开来，那么CondimentPrettyPrint方法会更容易编写。

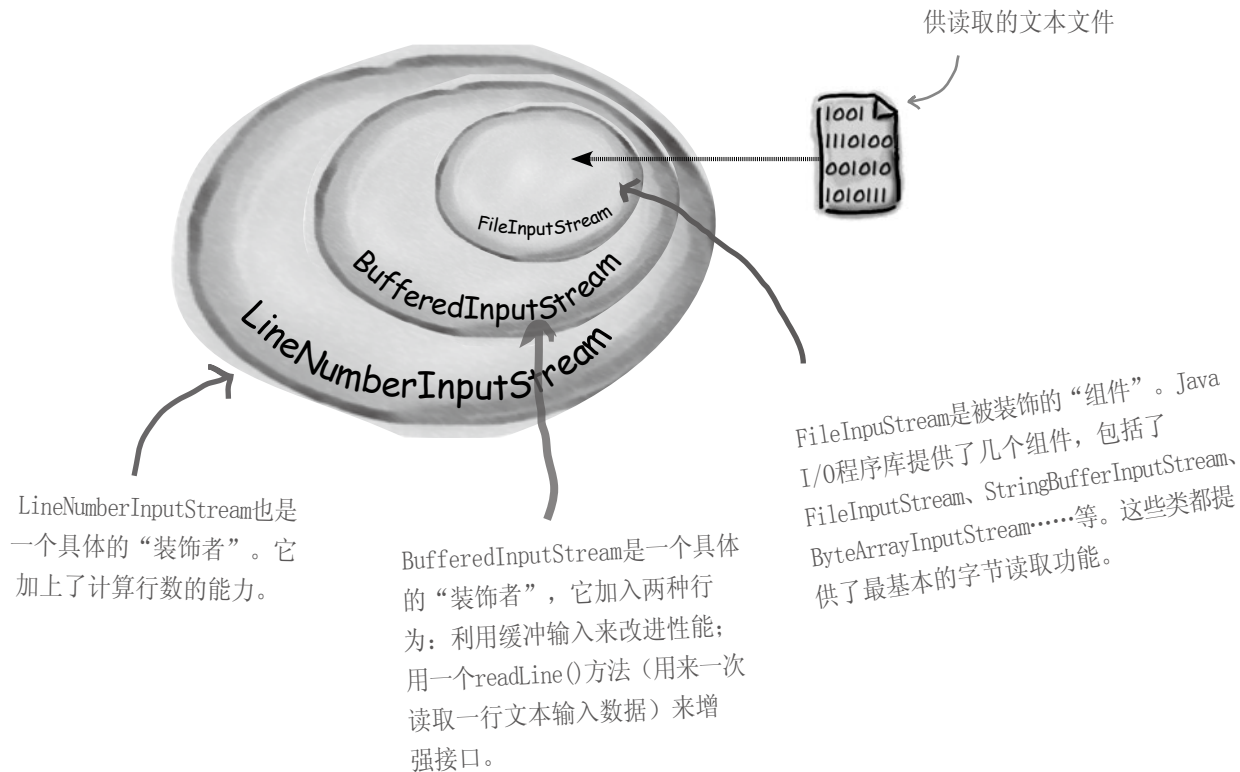


我们在星巴兹的朋友决定开始在菜单上加上咖啡的容量大小，供顾客可以选择小杯（tall）、中杯（grande）、大杯（venti）。星巴兹认为这是任何咖啡都必须具备的，所以在Beverage类中加上了getSize()与setSize()。他们也希望调料根据咖啡容量收费，例如：小中大杯的咖啡加上豆浆，分别加收0.10、0.15、0.20美金。

如何改变装饰者类应对这样的需求？

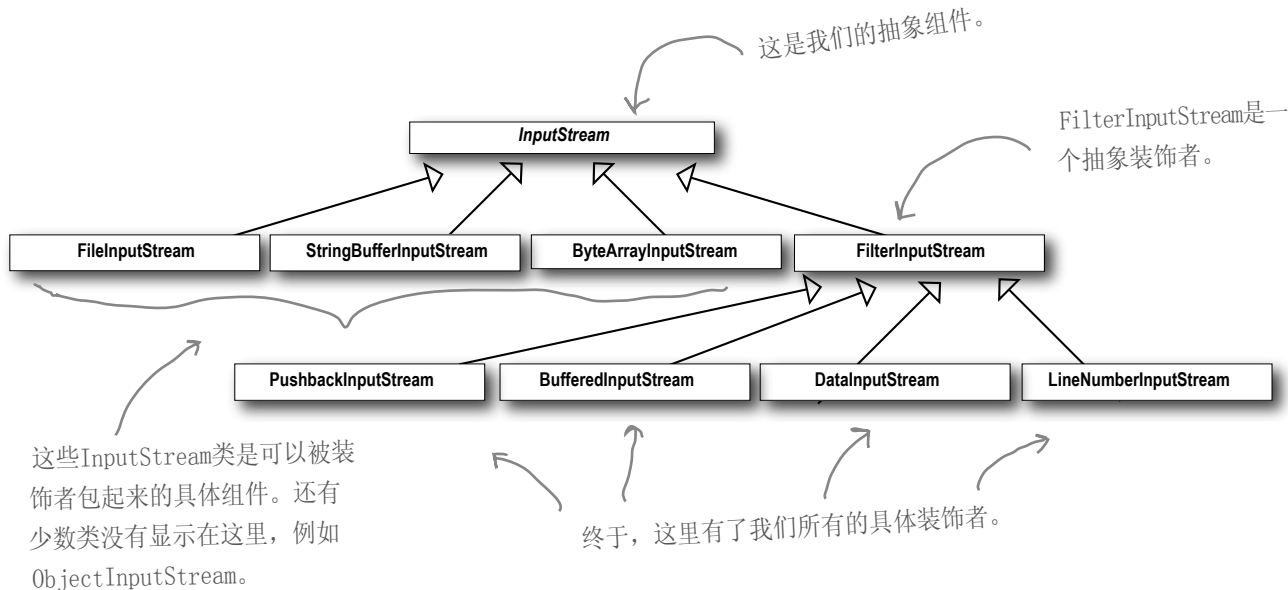
## 真实世界的装饰者：Java I/O

java.io包内的类太多了，简直是……“排山倒海”。你第一次（还有第二次和第三次）看到这些API发出“哇”的惊叹时，放心，你不是唯一受到惊吓的人。现在，你已经知道装饰者模式，这些I/O的相关类对你来说应该更有意义了，因为其中许多类都是装饰者。下面是一个典型的对象集合，用装饰者来将功能结合起来，以读取文件数据：



BufferedInputStream及LineNumberInputStream都扩展自FilterInputStream，而FilterInputStream是一个抽象的装饰类。

# 装饰java.io类



你可以发现，和星巴兹的设计相比，java.io其实没有多大的差异。我们把java.io API范围缩小，让你容易查看它的文件，并组合各种“输入”流装饰者来符合你的用途。

你会发现“输出”流的设计方式也是一样的。你可能还会发现Reader/Writer流（作为基于字符数据的输入输出）和输入流/输出流的类相当类似（虽然有一些小差异和不一致之处，但是相当雷同，所以你应该可以了解这些类）。

但是Java I/O也引出装饰者模式的一个“缺点”：利用装饰者模式，常常造成设计中有大量的小类，数量实在太多，可能会造成使用此API程序员的困扰。但是，现在你已经了解了装饰者的工作原理，以后当使用别人的大量装饰的API时，就可以很容易地辨别出他们的装饰者类是如何组织的，以方便使用包装方式取得想要的行为。

## 编写自己的Java I/O装饰者

你已经知道装饰者模式，也看过Java I/O类图，应该已经准备好编写自己的输入装饰者了。

这个想法怎么样：编写一个装饰者，把输入流内的所有大写字符转成小写。举例：当读取“I know the Decorator Pattern therefore I RULE!”，装饰者会将它转成“i know the decorator pattern therefore i rule!”

没问题，我只要扩展FilterInputStream类，并覆盖read()方法就行了！

不要忘了导入java.io……  
(这里省略了)

首先，扩展FilterInputStream，这是所有InputStream的抽象装饰者。

```
public class LowerCaseInputStream extends FilterInputStream {
    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = super.read();
        return (c == -1 ? c : Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int offset, int len) throws IOException {
        int result = super.read(b, offset, len);
        for (int i = offset; i < offset+result; i++) {
            b[i] = (byte)Character.toLowerCase((char)b[i]);
        }
        return result;
    }
}
```

别忘了：我们在代码中没有列出package与import语句。如果想取得完整的源代码，可以到第xxxv页中列出的wickedlysmart网站URL下载。

现在，必须实现两个read()方法，一个针对字节，一个针对字节数组，把每个是大写字符的字节（每个代表一个字符）转成小写。





## 测试你的新Java I/O装饰者

写个小程序，来测试刚写好的I/O 装饰者：

```
public class InputTest {
    public static void main(String[] args) throws IOException {
        int c;
        try {
            InputStream in =
                new LowerCaseInputStream(
                    new BufferedInputStream(
                        new FileInputStream("test.txt")));

            while((c = in.read()) >= 0) {
                System.out.print((char)c);
            }

            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

设置FileInputStream, 先用  
BufferedInputStream装饰它, 再用我  
们崭新的LowerCaseInputStream过  
滤器装饰它。

只用流来读取字符，一直到文件  
尾端。每读一个字符，就马上将  
它显示出来。

I know the Decorator Pattern therefore I RULE!

test.txt file

你需要做出这个  
文件。

运行看看：

```
File Edit Window Help DecoratorsRule
% java InputTest
i know the decorator pattern therefore i rule!
%
```



## 模式访谈

本周访问：  
装饰者的告白

HeadFirst：欢迎装饰者模式，听说你最近情绪有点差？

装饰者：是的，我知道大家都认为我是一个有魅力的设计模式，但是，你知道吗？我也有自己的困扰，就和大家一样。

HeadFirst：愿意让我们分担一些你的困扰吗？

装饰者：当然可以。你知道我有能力为设计注入弹性，这是毋庸置疑的，但我也有“黑暗面”。有时候我会在设计中加入大量的小类，这偶尔会导致别人不容易了解我的设计方式。

HeadFirst：你能够举个例子吗？

装饰者：以Java I/O库来说，人们第一次接触到这个库时，往往无法轻易地理解它。但是如果他们能认识到这些类都是用来包装InputStream的，一切都会变得简单多了。

HeadFirst：听起来并不严重。你还是一个很好的模式，只需要一点点的教育，让大家知道怎么用，问题就解决了。

装饰者：恐怕不只这些，我还有类型问题。有些时候，人们在客户代码中依赖某种特殊类型，然后忽然导入装饰者，却又没有周详地考虑一切。现在，我的一个优点是，你通常可以透明地插入装饰者，客户程序甚至不需知道它是在和装饰者打交道。但是，如我刚刚所说的，有些代码会依赖特定的类型，而这样的代码一导入装饰者，嘭！出状况了！

HeadFirst：这个嘛，我相信每个人都必须了解到，在插入装饰者时，必须要小心谨慎。我不认为这是你的错！

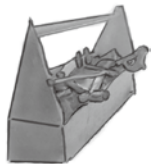
装饰者：我知道，我也试着不这么想。我还有一个问题，就是采用装饰者在实例化组件时，将增加代码的复杂度。一旦使用装饰者模式，不只需要实例化组件，还要把此组件包装进装饰者中，天晓得有几个。

HeadFirst：我下周会访谈工厂（Factory）模式和生成器（Builder）模式，我听说他们对这个问题的帮助。

装饰者：那倒是真的。我应该常和这些家伙聊聊。

HeadFirst：我们都认为你是一个好的模式，适合用来建立有弹性的设计，维持开放—关闭原则。你要开心一点，别负面思考。

装饰者：我尽量吧，谢谢你！



## 设计箱内的工具

本章已经接近尾声，你的工具箱内又多了一个新的原则和一个新的模式。

### 00 基础

### 00 原则

封装变化

多用组合，少用继承

针对接口编程，不针对实现编程

为交互对象之间的松耦合设计而努力

对扩展开放，对修改关闭。

抽象  
封装  
多型  
继承

现在有了开放-关闭原则引导我们。我们会努力地设计系统，好让关闭的部分和新扩展的部分隔离。

### 00 模式

“策略  
起来  
演算  
式。

观察者模式——在对象之间定义一对

装饰者模式——动态地将责任附加到对象上。想要扩展功能，装饰者提供有别于继承的另一种选择。

这是第一个符合开放-关闭原则的模式。真的是第一个吗？有没有其他曾经用过的遵循此原则的模式？

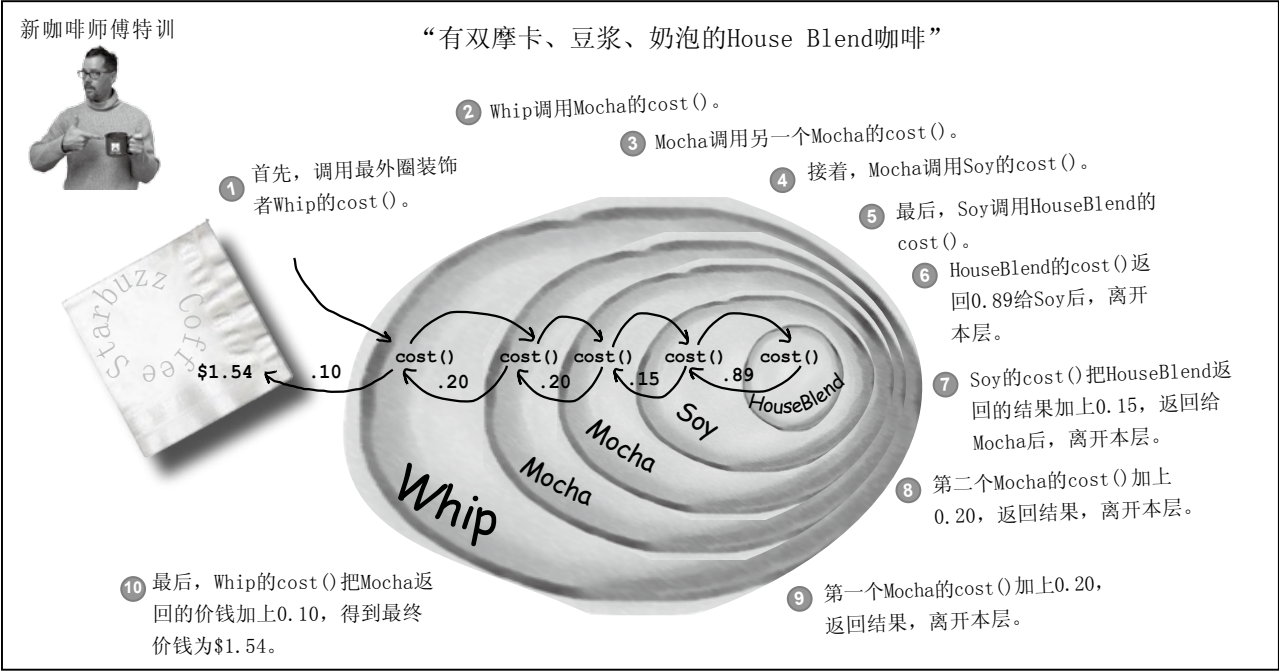
## 要点



- ， 继承属于扩展形式之一，但不见得是达到弹性设计的最佳方式。
- ， 在我们的设计中，应该允许行为可以被扩展，而无须修改现有的代码。
- ， 组合和委托可用于在运行时动态地加上新的行为。
- ， 除了继承，装饰者模式也可以让我们扩展行为。
- ， 装饰者模式意味着一群装饰者类，这些类用来包装具体组件。
- ， 装饰者类反映出被装饰的组件类型（事实上，他们具有相同的类型，都经过接口或继承实现）。
- ， 装饰者可以在被装饰者的行为前面与/或后面加上自己的行为，甚至将被装饰者的行为整个取代掉，而达到特定的目的。
- ， 你可以用无数个装饰者包装一个组件。
- ， 装饰者一般对组件的客户是透明的，除非客户程序依赖于组件的具体类型。
- ， 装饰者会导致设计中出现许多小对象，如果过度使用，会让程序变得很复杂。

# 习题解答

```
public class Beverage {  
    //为milkCost、soyCost、mochaCost  
    //和whipCost声明实例变量。  
    //为milk、soy、mocha和whip  
    //声明getter与setter方法。  
  
    public double cost() {  
        float condimentCost = 0.0;  
        if (hasMilk()) {  
            condimentCost += milkCost;  
        }  
        if (hasSoy()) {  
            condimentCost += soyCost;  
        }  
        if (hasMocha()) {  
            condimentCost += mochaCost;  
        }  
        if (hasWhip()) {  
            condimentCost += whipCost;  
        }  
        return condimentCost;  
    }  
}  
  
public class DarkRoast extends Beverage {  
    public DarkRoast() {  
        description = "Most Excellent Dark Roast";  
    }  
    public double cost() {  
        return 1.99 + super.cost();  
    }  
}
```



# 习题解答

我们在星巴兹的朋友决定开始在菜单上加上咖啡的容量大小，供顾客可以选择小杯（tall）、中杯（grande）、大杯（venti）。星巴兹认为这是任何咖啡都必须具备的，所以在Beverage类中加上了getSize()与setSize()。他们也希望调料根据咖啡容量收费，例如：小中大杯的咖啡加上豆浆，分别加收0.10、0.15、0.20美金。

如何改变装饰者类应对这样的需求？

```
public class Soy extends CondimentDecorator {
    Beverage beverage;

    public Soy(Beverage beverage) {
        this.beverage = beverage;
    }

    public int getSize() {
        return beverage.getSize();
    }

    public String getDescription() {
        return beverage.getDescription() + ", Soy";
    }

    public double cost() {
        double cost = beverage.cost();
        if (getSize() == Beverage.TALL) {
            cost += .10;
        } else if (getSize() == Beverage.GRANDE) {
            cost += .15;
        } else if (getSize() == Beverage.VENTI) {
            cost += .20;
        }
        return cost;
    }
}
```

现在要把getSize()传播到被包装的饮料。因为所有的调料装饰者都会用到这个方法，所以也应该把它移到抽象类中。

在这里取得容量大小（全都传播到具体的饮料），然后加上适当的价钱。