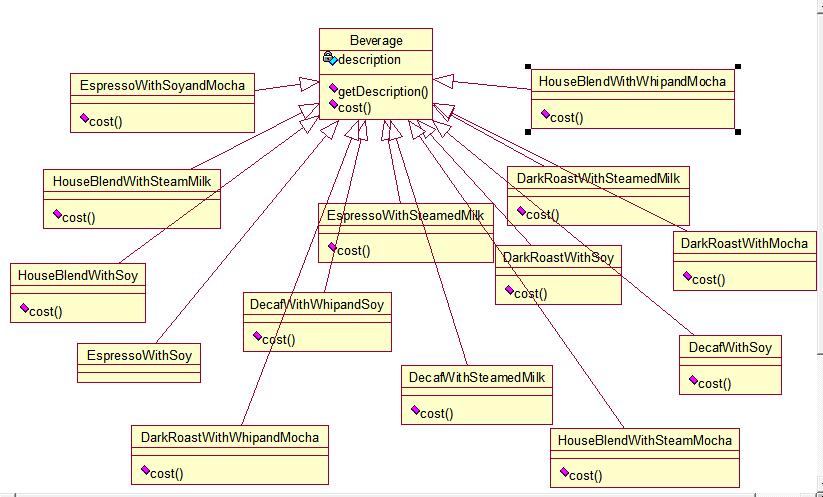
**装饰者模式：**动态的将责任附加到对象上。想要扩展功能，装饰者提供有别于继承的另一种选择。

**举个例子：星巴兹咖啡（不是星巴克）**

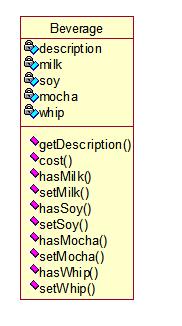
星巴兹扩张速度太快了，他们准备更新订单系统，以合乎他们的饮料供应要求。他们的要求是这样子的：购买咖啡时，可以要求在其中加入各种调料，例如：蒸奶（Steamed Milk）、豆浆(Soy)、摩卡（Mocha，也就是巧克力风味）或者覆盖奶泡。星巴兹会根据所加入的调料不同收取不同的费用。所以订单系统必须考虑到这些调料的部分。

　　先来进行第一次尝试：

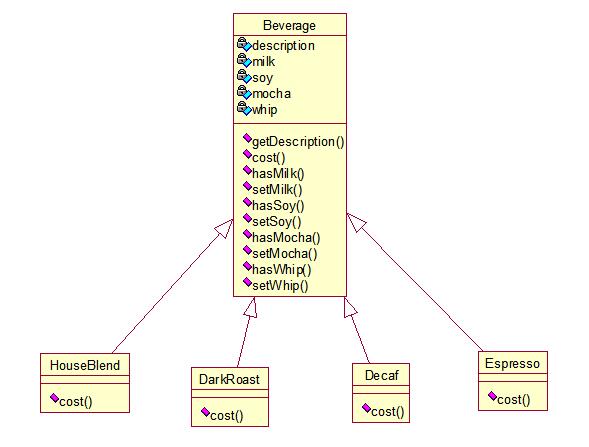


　　哇塞！这简直是“类爆炸”！很明显，按照上面做出来的的系统同样创造了一个维护噩梦。如果调料Milk价格上涨，怎么办？新增加一种调料时，就会诞生多少种新的咖啡？造成这样维护上的困难，违背了我们的设计原则：多用组合，少用继承；要针对接口编程，而不是针对实现编程。

　　那我们就换一种方式，先从Beverage基类入手，加上实例变量代表是否加上调料（牛奶、豆浆、摩卡、奶泡、辣椒......）



　　现在加入子类，每个类代表一种饮料。



　　认真想一下上面的方法还是有一些潜在的问题。当哪些需求或因素改变时会影响这个设计？

　　1.当调料价格的改变会使我们更改现有代码。

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

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

　　4.万一顾客想要双倍牛奶咖啡，怎么办?

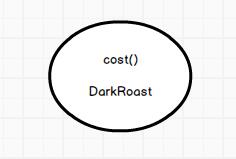
**在此我们将介绍最重要的设计原则之一**

**类应该对扩展开放，对修改关闭（开放封闭原则）**

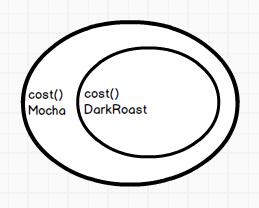
现在我们介绍今天的主角：**装扮者模式**

**以装饰者构造饮料订单**

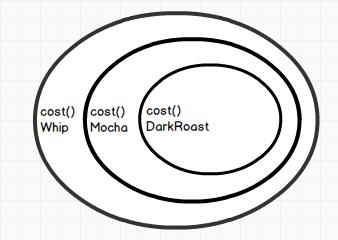
**1.以DarkRoast对象开始**

****

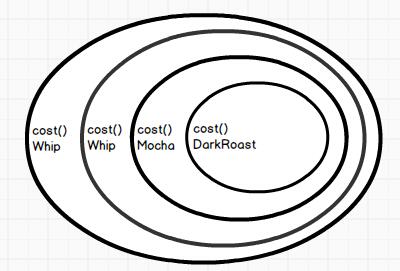
**2.顾客想要摩卡(Mocha)，所以建立一个Mocha对象，并用它将DarkRoast对象包装(wrap)起来。**

****

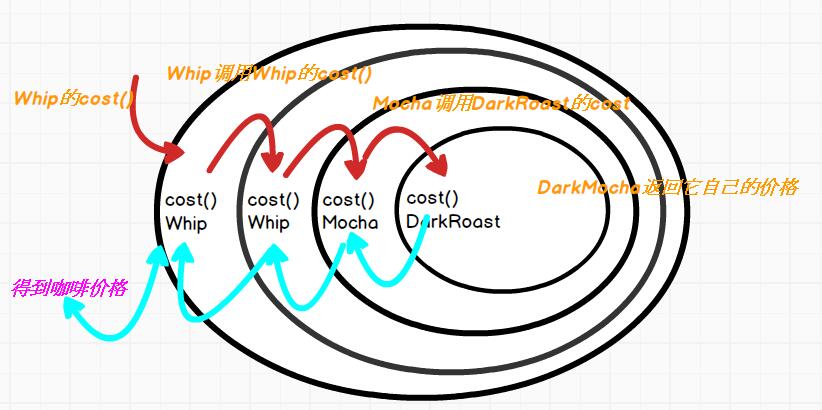
**3.顾客也想要奶泡(Whip)，所以需要建立一个Whip装饰者，并用它将Mocha对象包装起来。别忘了，DarkRoast继承自Beverage，且有一个cost()方法，用来计算饮料价格。**

****

**4.顾客其实对奶泡情有独钟，所以点了双倍的奶泡.**

****

**5.现在该结账了**

****

**好了，这是目前我们了解的装饰者模式：**

**1.装饰者和被装饰者对象有相同的超类型。**

**2.你可以用一个或者多个装饰者包装一个对象。**

**3.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。**

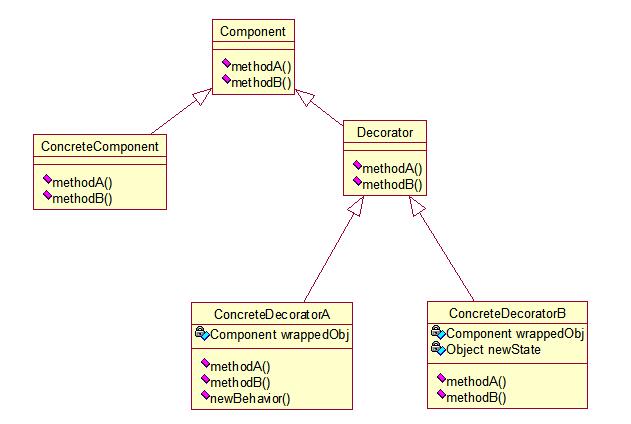
**4.装饰者可以在所委托被装饰者的行为之前或之后，加上自己的行为，以达到特定的目的。**

**5.对象可以在任何时候被装饰，所以在运行时动态地、不限量地使用你喜欢的装饰者来装饰对象，**

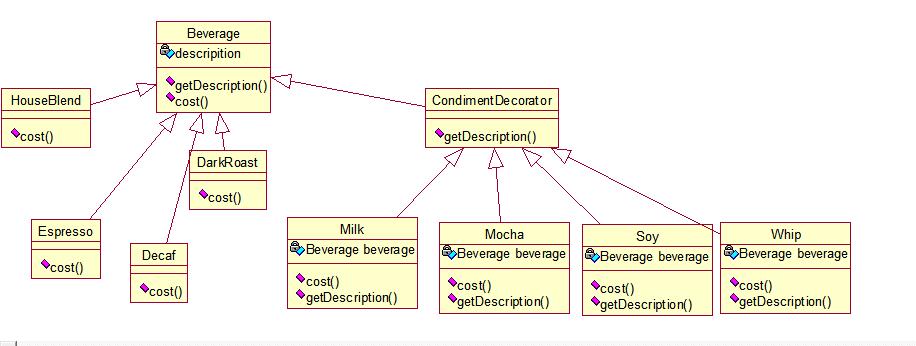
**定义装饰者模式：**

**动态的将责任附加到对象上。想要扩展功能，装饰者提供比继承更有弹性的替代方案。**

　　我们来看一下类图：



　　星巴兹咖啡的类图：



**现在写下星巴兹的代码：**

**先从Beverage类入手：**

public abstract class Beverage { //Beverage是一个抽象类

String description = "Unknow Beverage";

public String getDescription() { //getDescription已经实现了，但是cost需要在子类中实现

return description;

}

public abstract double cost();

}

**现在来实现Condiment（调料）抽象类，也就是装饰者类：**

public abstract class CondimentDecorator extends Beverage {

public abstract String getDescription();//所有的调料类都必须重新该方法

}

**写饮料代码：**

public class Espresso extends Beverage {//Espresso扩展自 Beverage

public Espresso() {

description = "Espresso";//设置饮料的描述

}

public double cost(){

return 1.99;//返回Espresso的价格

}

}

public class HouseBlend extends Beverage {//HouseBlend扩展自 Beverage

public HouseBlend(){

description ="HouseBlend";//设置饮料的描述

}

public double cost(){

return 0.89;//返回HouseBlend的价格

}

}

**写调料代码：**

public class Mocha extends CondimentDecorator {//Mocha扩展自 CondimentDecorator

Beverage beverage;//用一个实例记录饮料,也就是被装饰者

public Mocha(Beverage beverage) {//让被装饰者(饮料)被记录到实例变量中

this.beverage = beverage;

}

public String getDescription() {//不仅仅描述饮料，同样可以将调料完整地显示出来

return beverage.getDescription()+" , Mocha";

}

public double cost() {

return 0.20 + beverage.cost();//返回带Mocha饮料的价钱

}

}