面向对象设计模式　装饰者

曹东刚
caodg@pku.edu.cn

北京大学信息学院研究生课程 - 面向对象的分析与设计
http://sei.pku.edu.cn/~caodg/course/oo

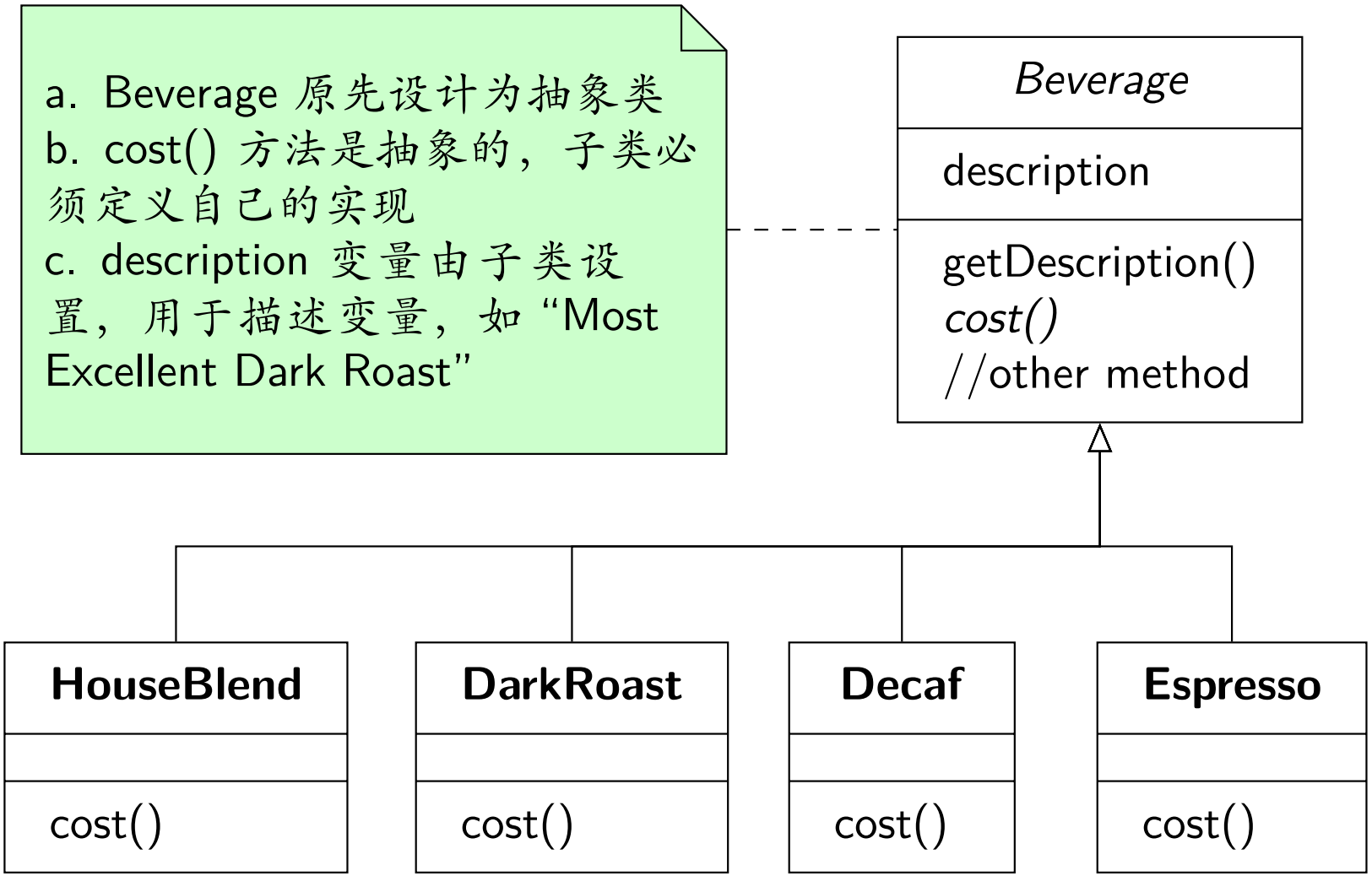
装饰对象（Decorating Objects）

本章将学习如何使用对象聚合的方式，做到在运行时装饰类

一旦熟悉了装饰的技巧，就可在不修改任何底层代码的情况下，给对象 赋予新的职责

运行时扩展远比编译时继承威力大!

Starbuzz 咖啡店准备更新订单系统



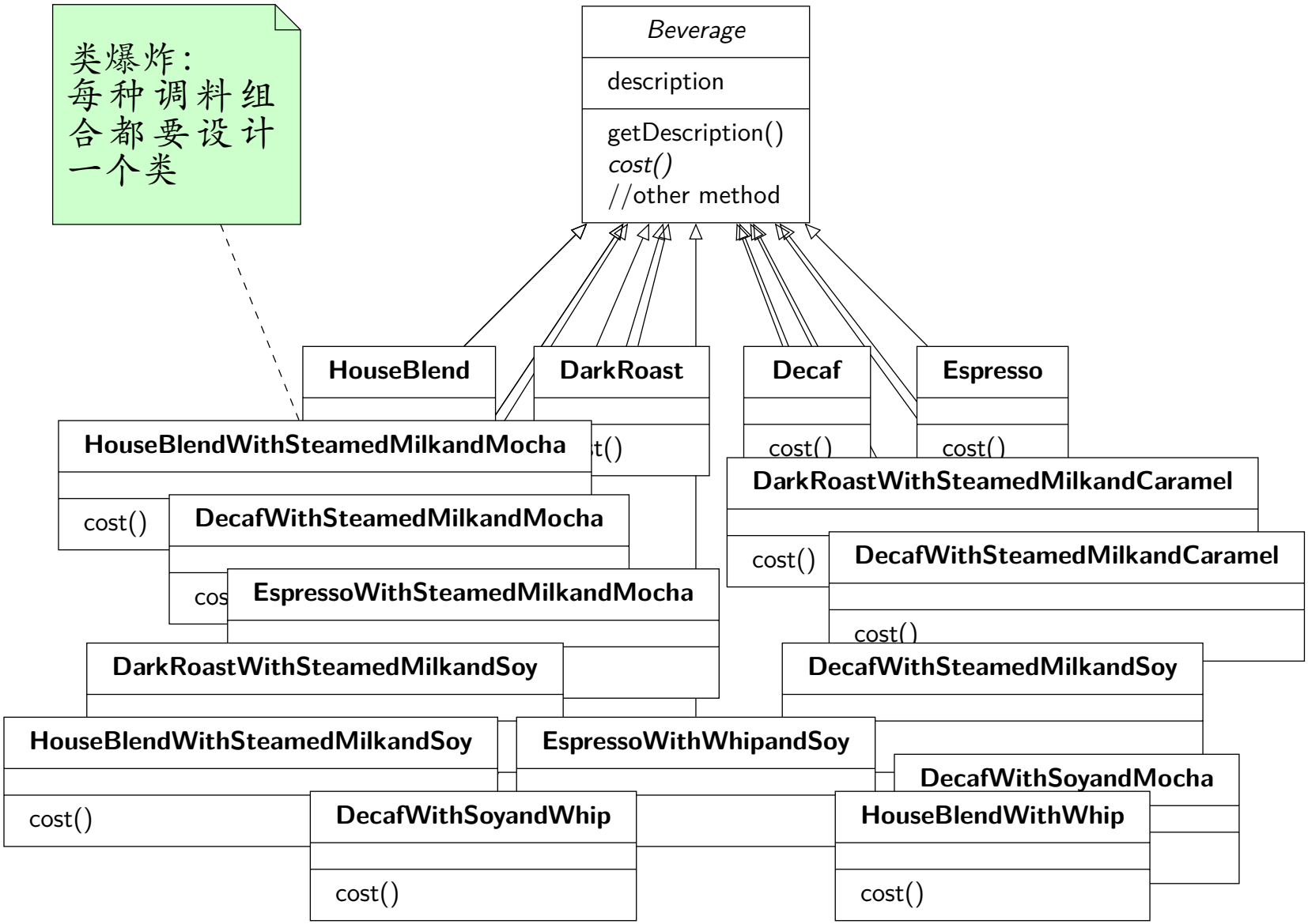
Starbuzz 咖啡店准备更新订单系统

需求:

购买咖啡时，可以要求在其中加入各种调料，如 Steamed Milk, Soy, Mocha 或者 Whipped Milk。店家会根据加入的调料收取不同的费用。订单必须考虑到这些调料部分。

他们的第一个设计 —

Starbuzz 咖啡店准备更新订单系统



Starbuzz 咖啡店准备更新订单系统

这种设计会造成严重的维护问题:

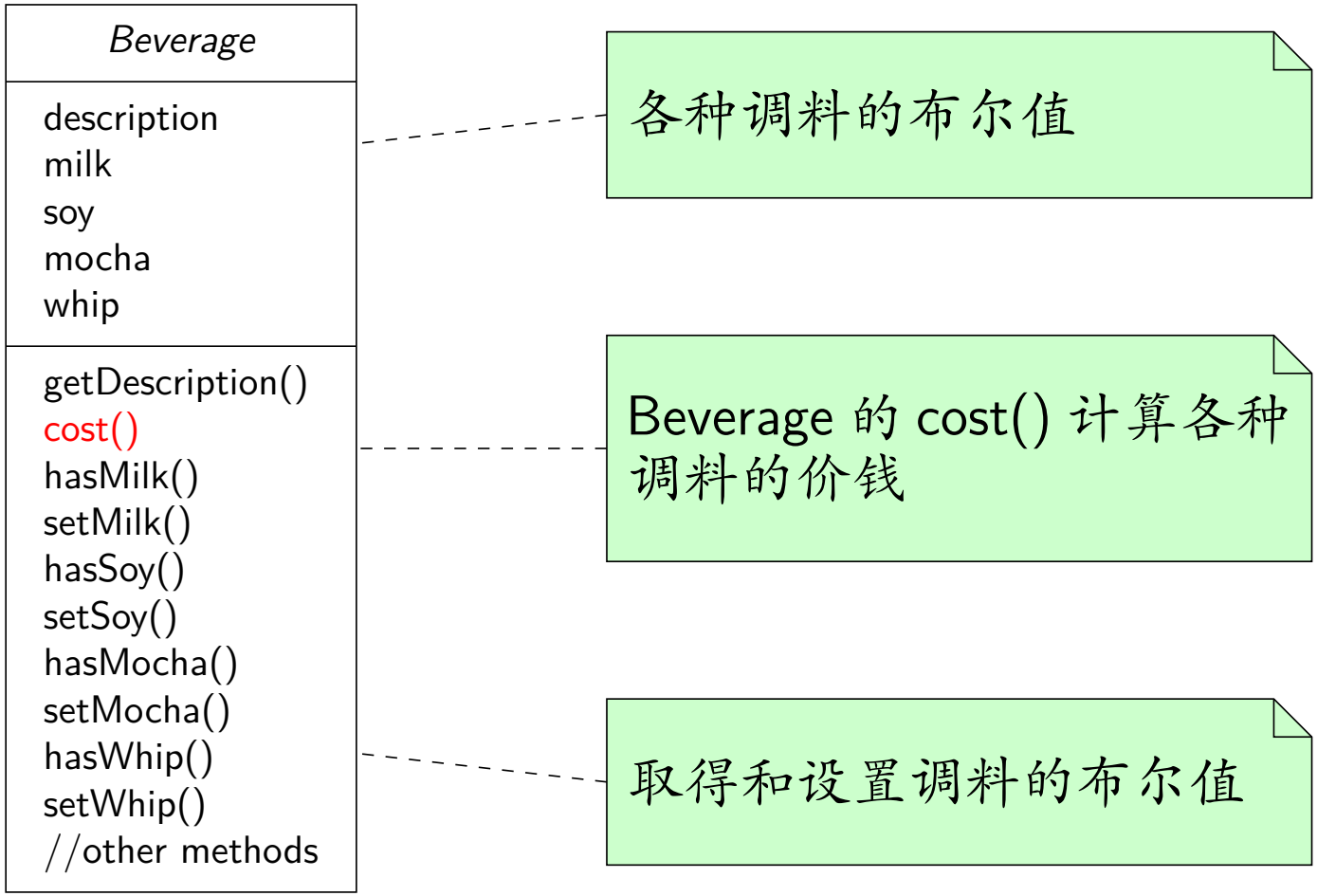
- 如果某种饮料价格变动，应如何处理?
- 如果要新增一种调料，应如何处理?

造成这种困境是因为该设计违背了如下设计原则

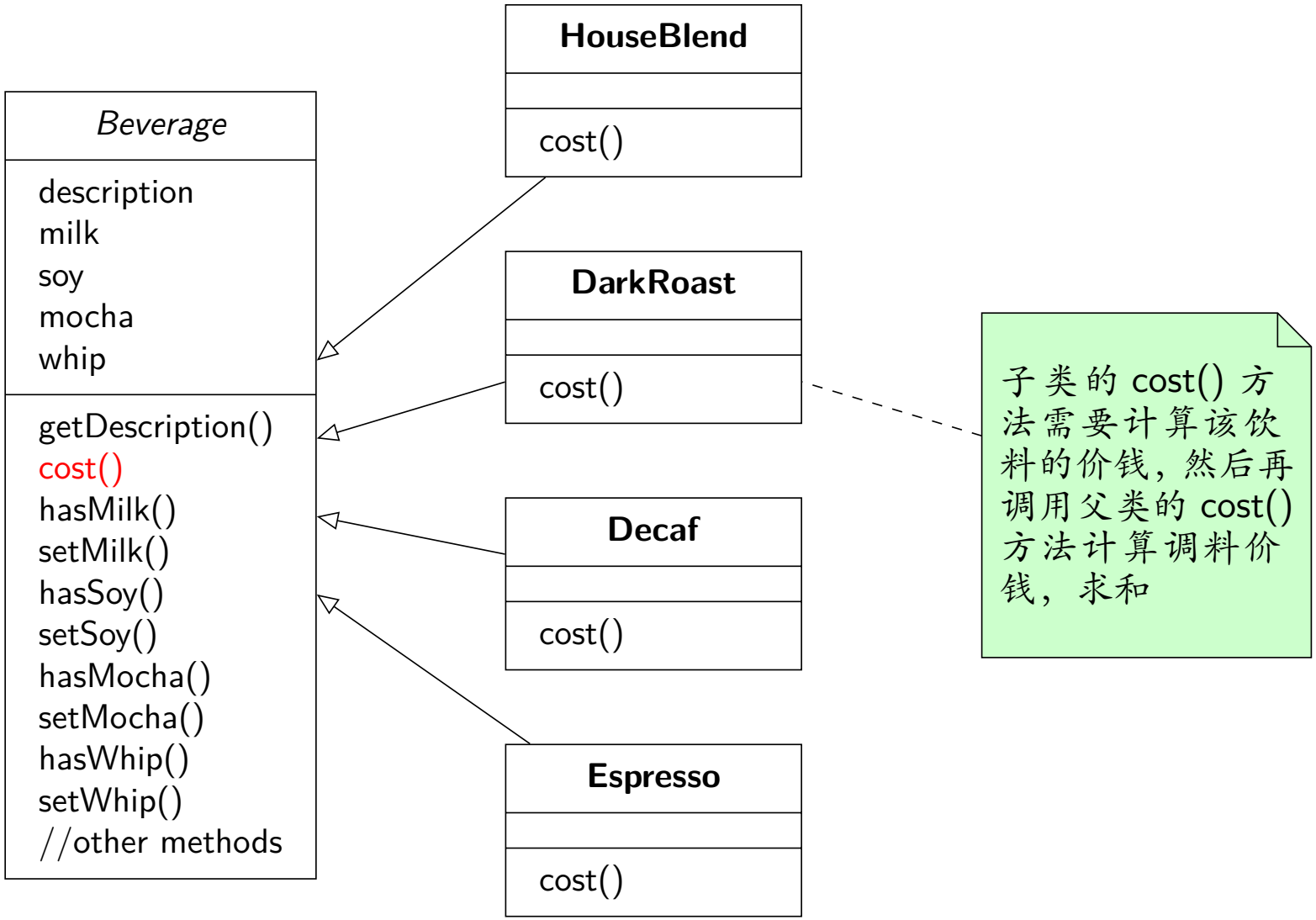
封装变化

多用聚合、少用继承

新的设计: 取消调料类


Beverage
description
milk
soy
mocha
whip
getDescription()
cost()
hasMilk()
setMilk()
hasSoy()
setSoy()
hasMocha()
setMocha()
hasWhip()
setWhip()
//other methods
各种调料的布尔值
Beverage 的 cost() 计算各种调料的价钱
取得和设置调料的布尔值

新的设计：取消调料类


Beverage
description
milk
soy
mocha
whip
getDescription()
cost()
hasMilk()
setMilk()
hasSoy()
setSoy()
hasMocha()
setMocha()
hasWhip()
setWhip()
//other methods
HouseBlend
cost()
DarkRoast
cost()
Decaf
cost()
Espresso
cost()
子类的 cost() 方法需要计算该饮料的价钱，然后再调用父类的 cost() 方法计算调料价钱，求和

新的设计: 取消调料类

Beverage 的 cost() 方法伪码

```
1  public double cost() {
2    float condimentCost = 0.0 ;
3    if (hasMilk())
4      condimentCost += milkCost ;
5    if (hasSoy())
6      condimentCost += soyCost ;
7    if (hasMocha())
8      condimentCost += mochaCost ;
9    if (hasWhip())
10     condimentCost += whipCost ;
11   return condimentCost ;
12 }
```

新的设计: 取消调料类

DarkRoast 的伪码

```
1  public class DarkRoast extends Beverage {
2    public DarkRoast() {
3      description = "Most Excellent Dark Roast" ;
4    }
5
6    public double cost() {
7      return 1.99 + super.cost() ;
8    }
9  }
```

新的设计: 取消调料类

该设计相比最初的方案有很大改善，但是:

- 调料价钱的改变会引发对代码的改动
- 一旦出现新的调料，就需要加上新的方法，并改变父类 Beverage 的 cost() 方法
- 将来可能会有新的饮料出现，如 iced tea，某些调料就可能不适合该饮料，那么继承而来的 hasWhip() 等方法可能就不适合
- 顾客可能需要双倍/三倍 mocha

仍然需要改进设计!

开放—关闭设计原则

设计原则：开放—关闭

类应该对 扩展 开放，对 修改 关闭

目标是允许“类容易扩展”，在不修改现有类代码的情况下，就可以让类具有新的行为

这种设计的好处是具有弹性，能够灵活应对改变的需求

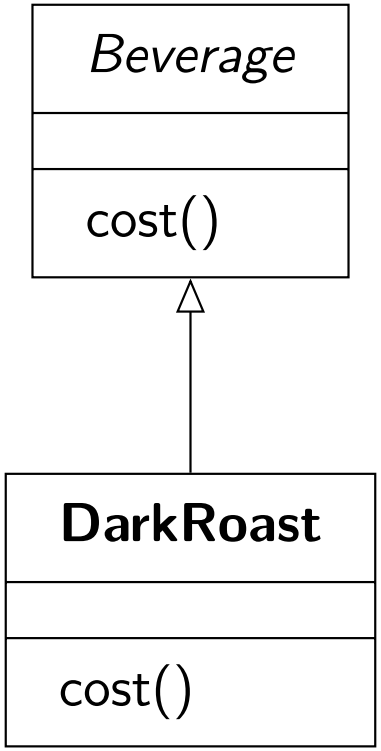
但：没必要把系统的每部分都设计为可扩展的

认识装饰者模式 (decorator)

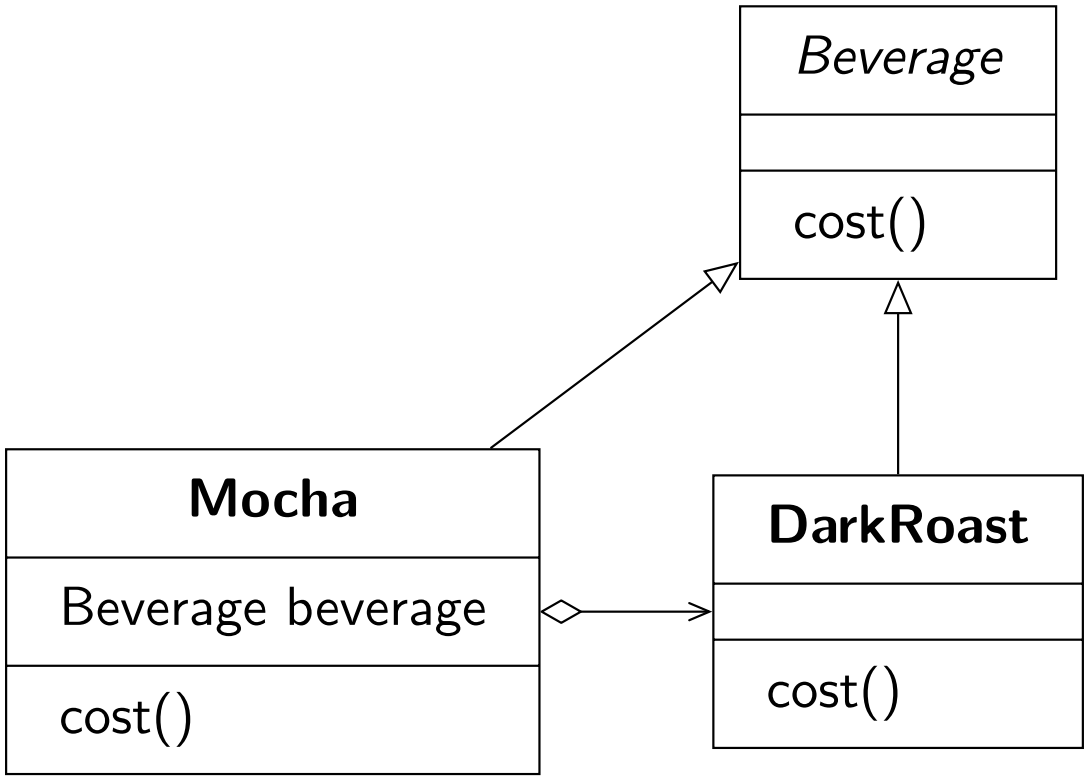
采用“继承”的设计为 Starbuzz 带来类爆炸、设计死板、父类新功能无法适应所有子类等问题，我们转而采用“装饰”的设计方法：

1. 拿一个 DarkRoast 对象
2. 用 Mocha 对象装饰它
3. 用 Whip 对装饰它
4. 用任何调料装饰它
5. 装饰对象的 cost() 方法：该方法先调用被装饰对象的 cost() 方法获得被装饰对象的价钱，加上装饰对象的调料价格求和

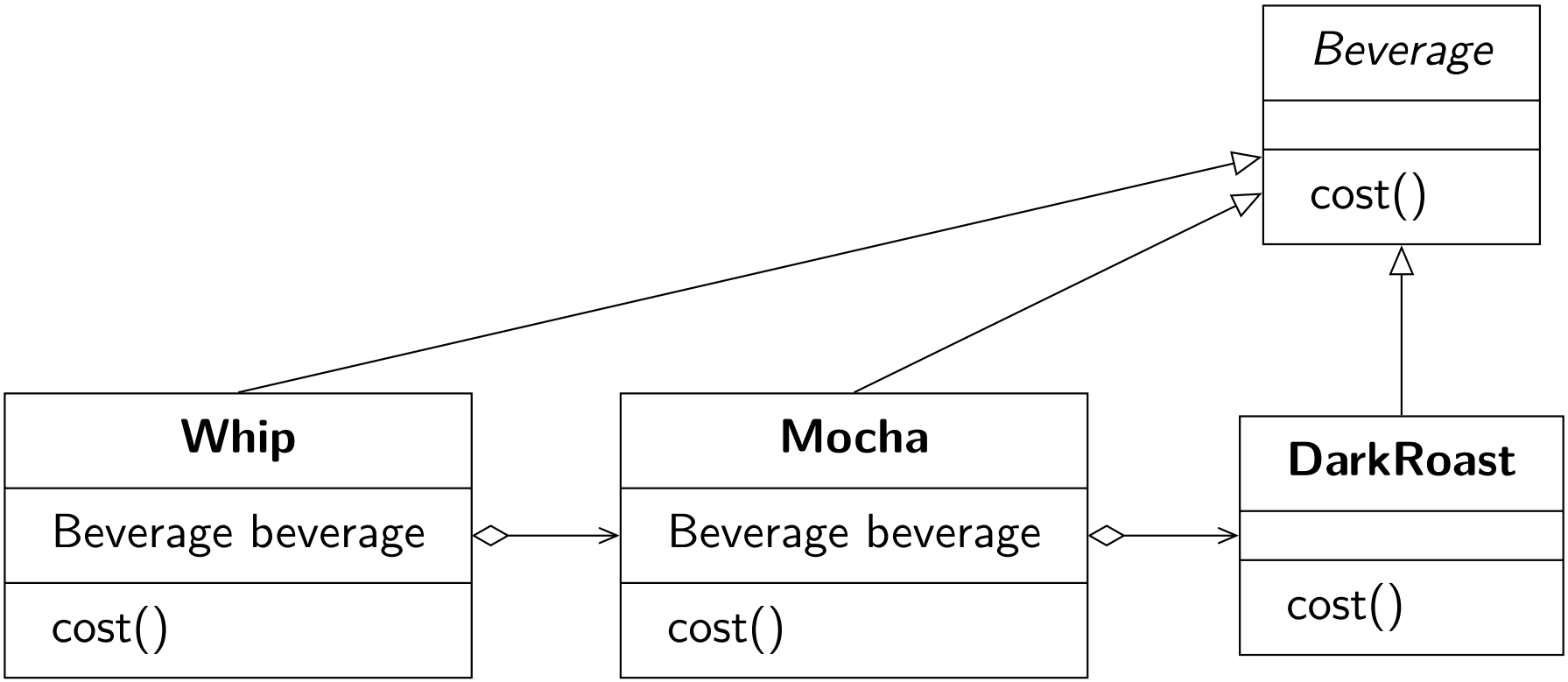
以装饰者模式构造新订单系统



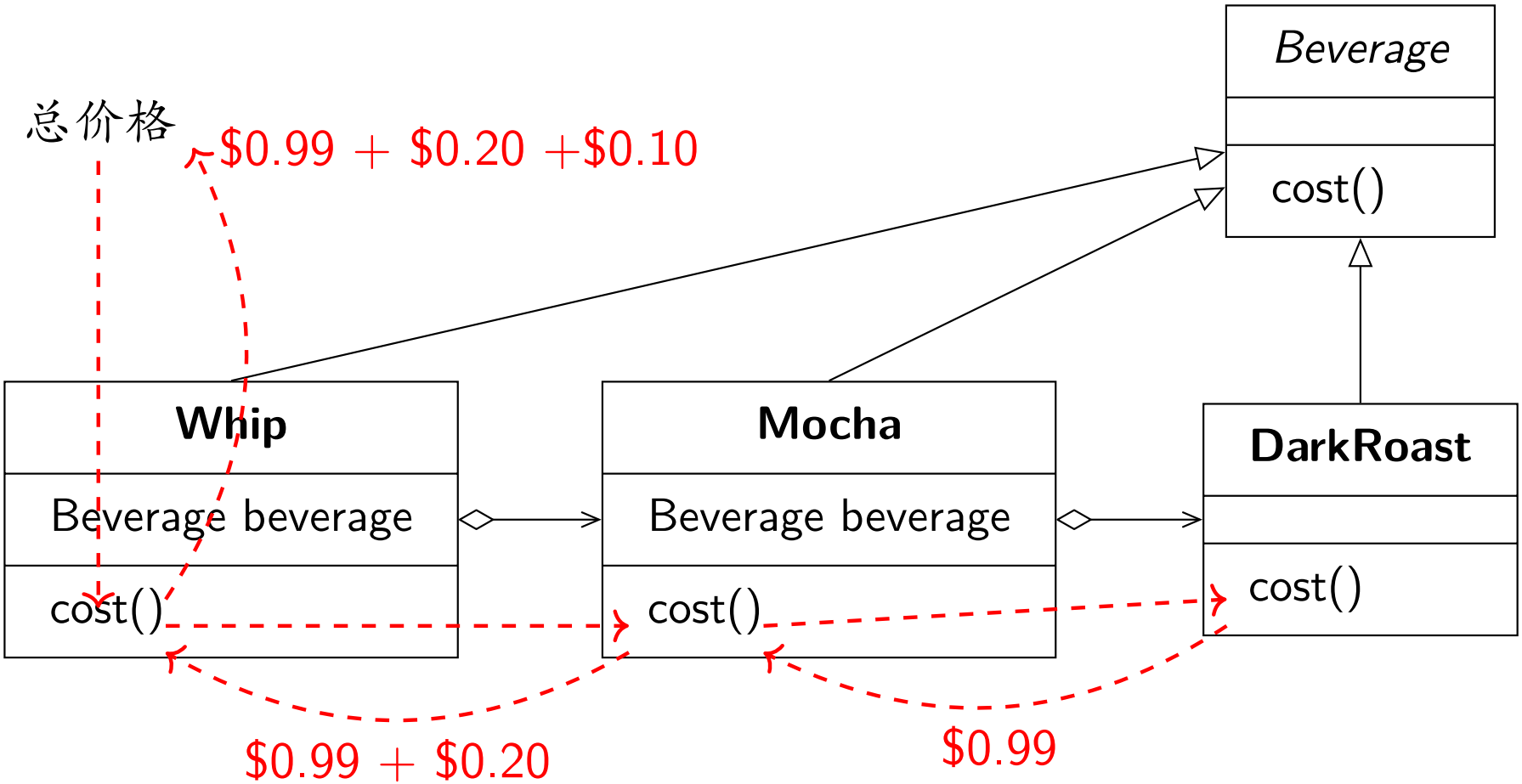
以装饰者模式构造新订单系统



以装饰者模式构造新订单系统



以装饰者模式构造新订单系统



以装饰者模式构造新订单系统

从前面的设计可以得知：

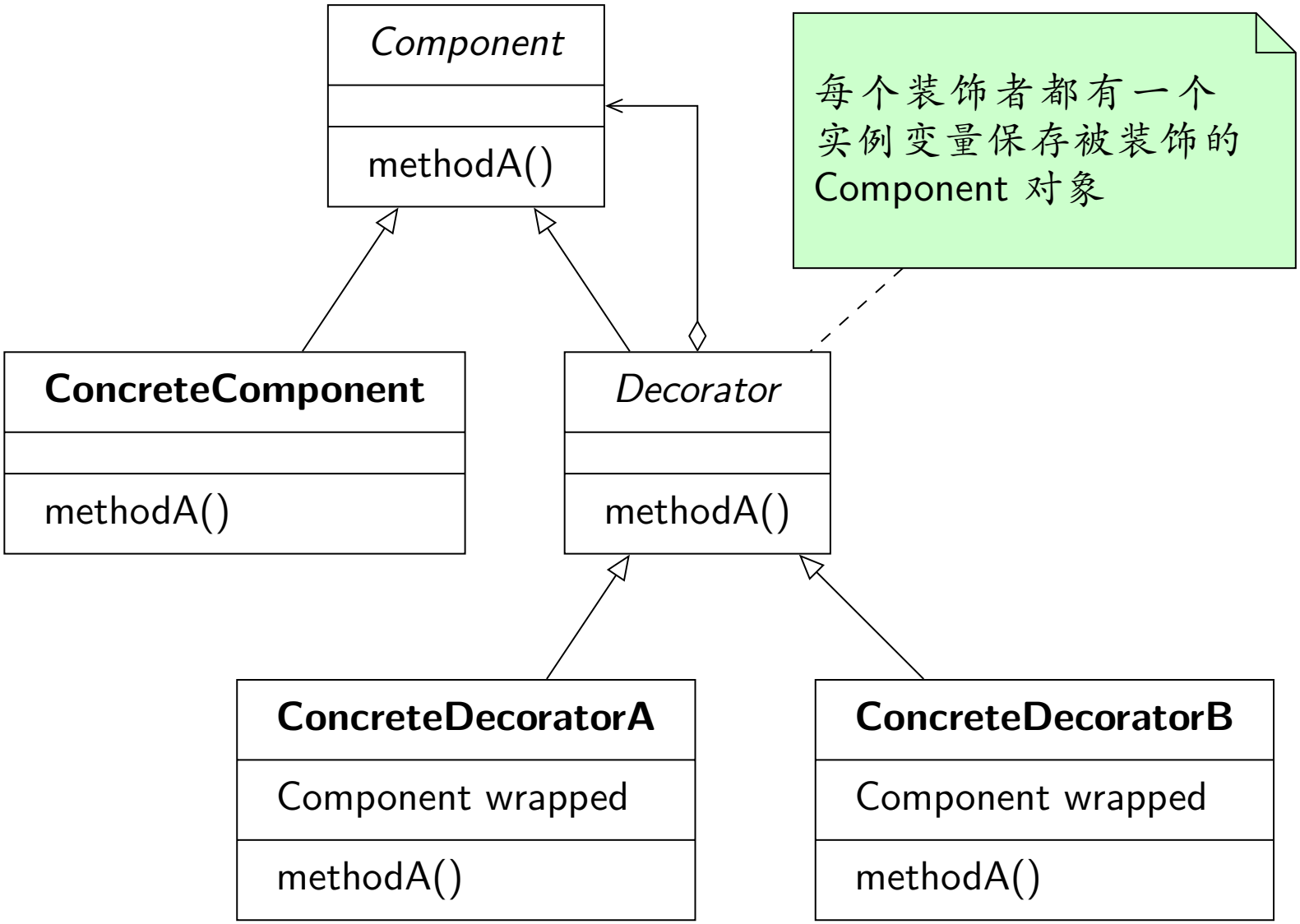
- 装饰者和被装饰者对象拥有一个相同的父类
- 可以用多个装饰者对象装饰一个被装饰者对象
- 由于装饰者和被装饰者的父类型相同，因此可以在任何需要原始对象的场合，用装饰过的对象取代它
- 装饰者可以在被装饰者的行为之前/之后，加上自己的特定行为
- 对象可以在任何时候被装饰，因此可以在运行时动态地、不受限制地进行装饰

定义装饰者模式

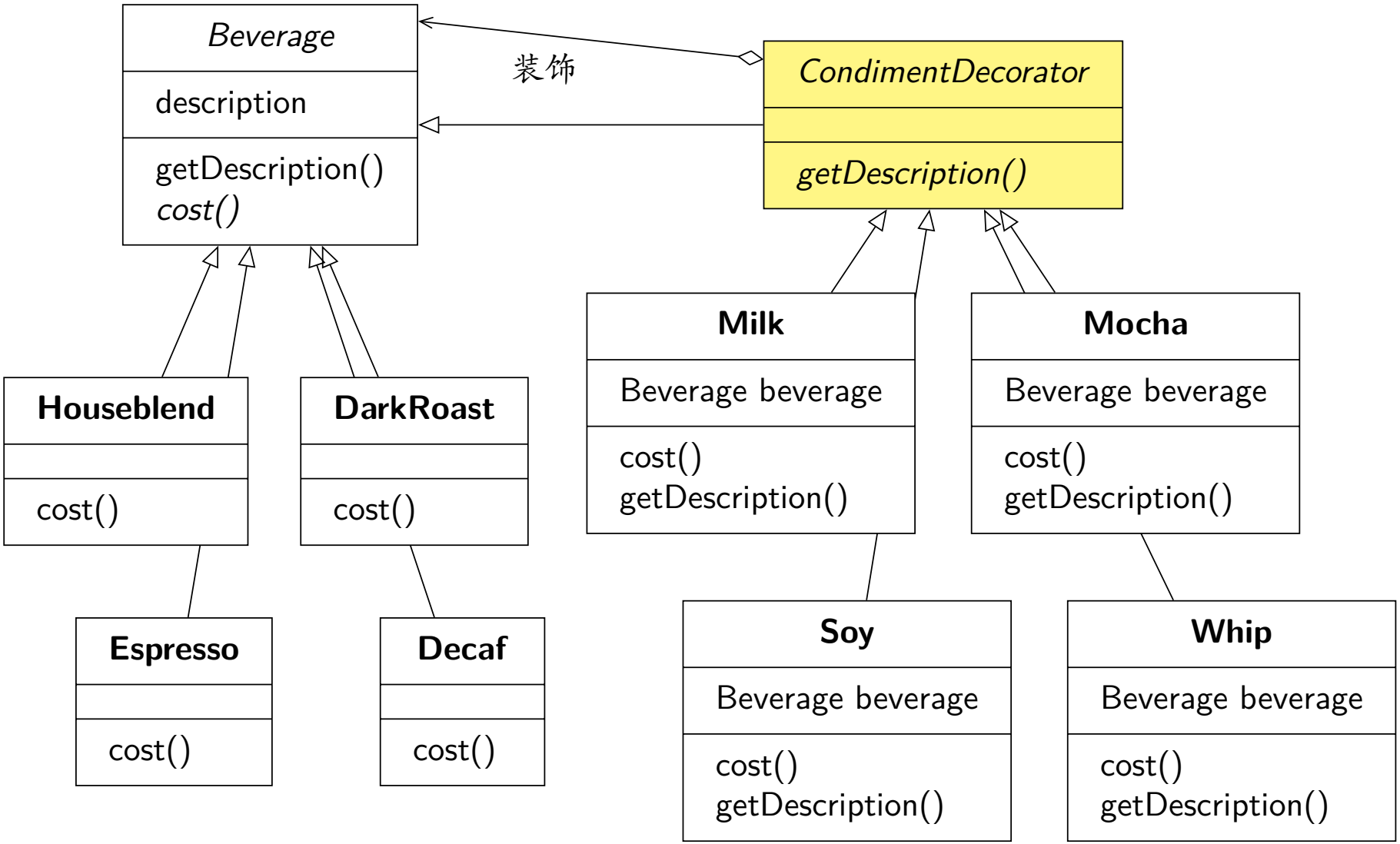
装饰者模式 (decorator pattern)

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

定义装饰者模式


Component
methodA()
每个装饰者都有一个实例变量保存被装饰的Component 对象
ConcreteComponent
methodA()
Decorator
methodA()
ConcreteDecoratorA
Component wrapped
methodA()
ConcreteDecoratorB
Component wrapped
methodA()

用装饰者模式设计 Starbuzz 饮料系统



装饰者模式中的继承和聚合

Q: 装饰者模式为什么仍然用到了继承?

装饰者模式中的继承和聚合

Q: 装饰者模式为什么仍然用到了继承?

A: 这里的继承是使装饰者和被装饰者具有一样的类型，而不是利用继承获得父类的“行为”

装饰者模式中的继承和聚合

Q: 装饰者模式为什么仍然用到了继承?

A: 这里的继承是使装饰者和被装饰者具有一样的类型，而不是利用继承获得父类的“行为”

Q: 那行为从何而来?

装饰者模式中的继承和聚合

Q: 装饰者模式为什么仍然用到了继承?
A: 这里的继承是使装饰者和被装饰者具有一样的类型，而不是利用继承获得父类的“行为”

Q: 那行为从何而来?
A: 当将装饰者和被装饰者聚合在一起时，就是在增加新的行为。新行为非来自继承，而是来自聚合

装饰者模式中的继承和聚合

Q: 装饰者模式为什么仍然用到了继承?
A: 这里的继承是使装饰者和被装饰者具有一样的类型，而不是利用继承获得父类的“行为”

Q: 那行为从何而来?
A: 当将装饰者和被装饰者聚合在一起时，就是在增加新的行为。新行为非来自继承，而是来自聚合

Q: 这里的聚合相比继承的好处是什么?

装饰者模式中的继承和聚合

Q: 装饰者模式为什么仍然用到了继承?
A: 这里的继承是使装饰者和被装饰者具有一样的类型，而不是利用继承获得父类的“行为”

Q: 那行为从何而来?
A: 当将装饰者和被装饰者聚合在一起时，就是在增加新的行为。新行为非来自继承，而是来自聚合

Q: 这里的聚合相比继承的好处是什么?
A: 若依赖继承，那么类的行为就在编译时刻决定了。若使用聚合，即可在运行时动态增加新行为，而不必修改已有代码

装饰者模式中的继承和聚合

Q: 装饰者模式为什么仍然用到了继承?
A: 这里的继承是使装饰者和被装饰者具有一样的类型，而不是利用继承获得父类的“行为”

Q: 那行为从何而来?
A: 当将装饰者和被装饰者聚合在一起时，就是在增加新的行为。新行为非来自继承，而是来自聚合

Q: 这里的聚合相比继承的好处是什么?
A: 若依赖继承，那么类的行为就在编译时刻决定了。若使用聚合，即可在运行时动态增加新行为，而不必修改已有代码

对扩展开放，对修改封闭

使用了装饰者模式的 Starbuzz 新系统伪代码

Starbuzz Coffees	
Coffees	
House Blend	0.89
Dark Roast	0.99
Decaf	1.05
Espresso	1.99
Condiments	
SteamedMilk	0.10
Mocha	0.20
Soy	0.15
Whip	0.10

根据 Starbuzz 的需求，实现各种咖啡和调料的具体类

使用了装饰者模式的 Starbuzz 新系统伪代码

被装饰者抽象类

```
1  public abstract class Beverage {
2    String description = "Unknown Beverage";
3    public String getDescription() {
4      return description;
5    }
6    public abstract double cost();
7  }
```

装饰者抽象类

```
1  public abstract class CondimentDecorator
2    extends Beverage {
3    public abstract String getDescription();
4  }
```

使用了装饰者模式的 Starbuzz 新系统伪代码

具体的被装饰者—饮料子类

```
1  public class Espresso extends Beverage {
2    public Espresso() {
3      description = "Espresso";
4    }
5
6    public double cost() {
7      return 1.99;
8    }
9  }
```

使用了装饰者模式的 Starbuzz 新系统伪代码

具体的装饰者—调料子类

```
1  public class Mocha extends CondimentDecorator {
2    Beverage beverage;
3    public Mocha(Beverage beverage) {
4      this.beverage = beverage;
5    }
6    public String getDescription() {
7      return beverage.getDescription() + ", Mocha";
8    }
9    public double cost() {
10     return .20 + beverage.cost();
11   }
12 }
```

使用了装饰者模式的 Starbuzz 新系统伪代码

测试类

```
1  public class StarbuzzCoffee {
2    public static void main(String args[]) {
3      Beverage beverage = new Espresso();
4      System.out.println(beverage.getDescription() + " $" +
5        beverage.cost());
6      Beverage beverage2 = new DarkRoast();
7      beverage2 = new Mocha(beverage2);
8      beverage2 = new Mocha(beverage2);
9      beverage2 = new Whip(beverage2);
10     System.out.println(beverage2.getDescription() +
11       " $" + beverage2.cost());
12   }
13 }
```

使用了装饰者模式的 Starbuzz 新系统伪代码

测试输出

```
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
%
```

讨论

问题：如果咖啡店会举行一些活动，比如特定型号的咖啡打折，上述设计是否适当？

讨论

问题：如果咖啡店会举行一些活动，比如特定型号的咖啡打折，上述设计是否适当？

问题：装饰者知道其他装饰者的存在吗，如果不能，如何让他们知道？

讨论

问题：如果咖啡店会举行一些活动，比如特定型号的咖啡打折，上述设计是否适当？

问题：装饰者知道其他装饰者的存在吗，如果不能，如何让他们知道？

问题：咖啡店决定区分容量大小，大杯、中杯、小杯的价格不同，应该如何调整设计？

讨论

问题：如果咖啡店会举行一些活动，比如特定型号的咖啡打折，上述设计是否适当？

问题：装饰者知道其他装饰者的存在吗，如果不能，如何让他们知道？

问题：咖啡店决定区分容量大小，大杯、中杯、小杯的价格不同，应该如何调整设计？

问题：装饰者模式会引入很多小类，是否增加了系统复杂度？

小结

关于装饰者模式:

- 继承可以扩展对象行为，但不是达到弹性设计的最佳方式
- 设计应允许扩展现有行为，而无须修改现有代码
- 聚合和委托可用于在运行时动态增加新行为
- 装饰者模式允许弹性地扩展现有行为
- 装饰者模式意味着很多装饰者类
- 可以用任意多个装饰者包装一个构件
- 过度使用装饰者模式会使程序变得复杂

小结

关于设计原则:

1. 封装变化
2. 多用聚合、少用继承
3. 针对接口编程，不针对实现编程
4. 尽最大可能将要交互的对象设计为松耦合的
5. 对扩展开放，对修改封闭