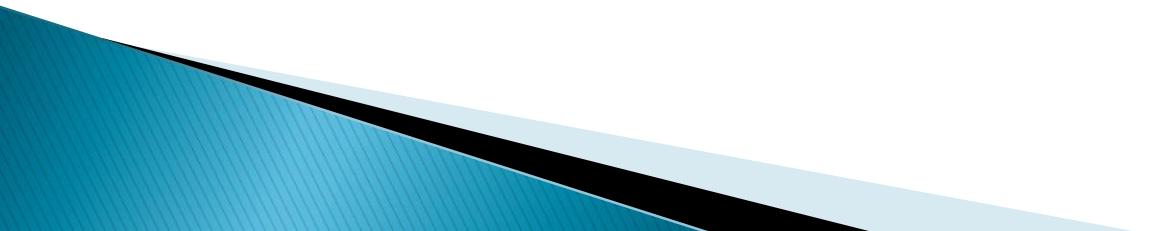


Design Patterns

Lecture 5

设计模式魅力——策略模式



全聚德场景

亲爱的工程师们：

你们好，随着全球化的趋势和电子商务的冲击，全聚德烤鸭公司面临巨大的挑战。为了做得更好，**我们需要一个可以展示我们可爱鸭子的平台。如果用户能亲眼看见那些鸭子在游泳或呱呱叫，那是多么美妙的一件事啊！**

我们最重要的需求就是这个平台要有较强的灵活性和扩展性，因为我们不打算支付任何升级或重构的费用，这对你们这群“面向对象设计工程师”来说应该不是难事吧？

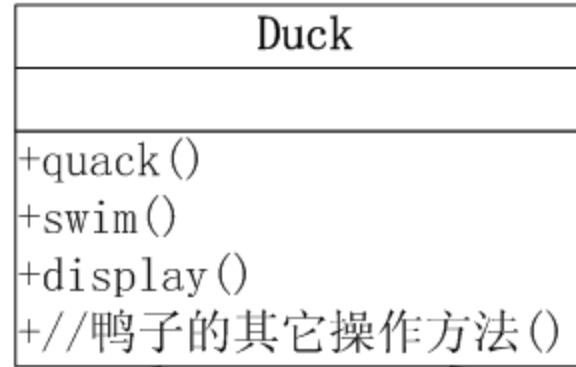
如果你们的设计能够满足我们的需求，我们将以公司的股票期权作支付。



全聚德总裁

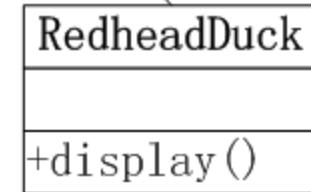
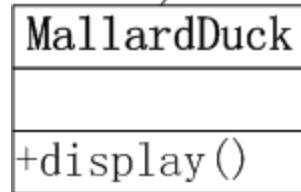
第一个设计

所有的鸭子都会呱呱叫 (Quack) , 也会游泳 (Swim) , 所以由超类负责处理这部分的实现代码



因为每一种鸭子的外观都不同 , 所以 Display 方法是抽象的...

每个鸭子子类型负责实现自己的 (Display) , 表示其外观的显示



许多其它类型的鸭子继承自 Duck 类

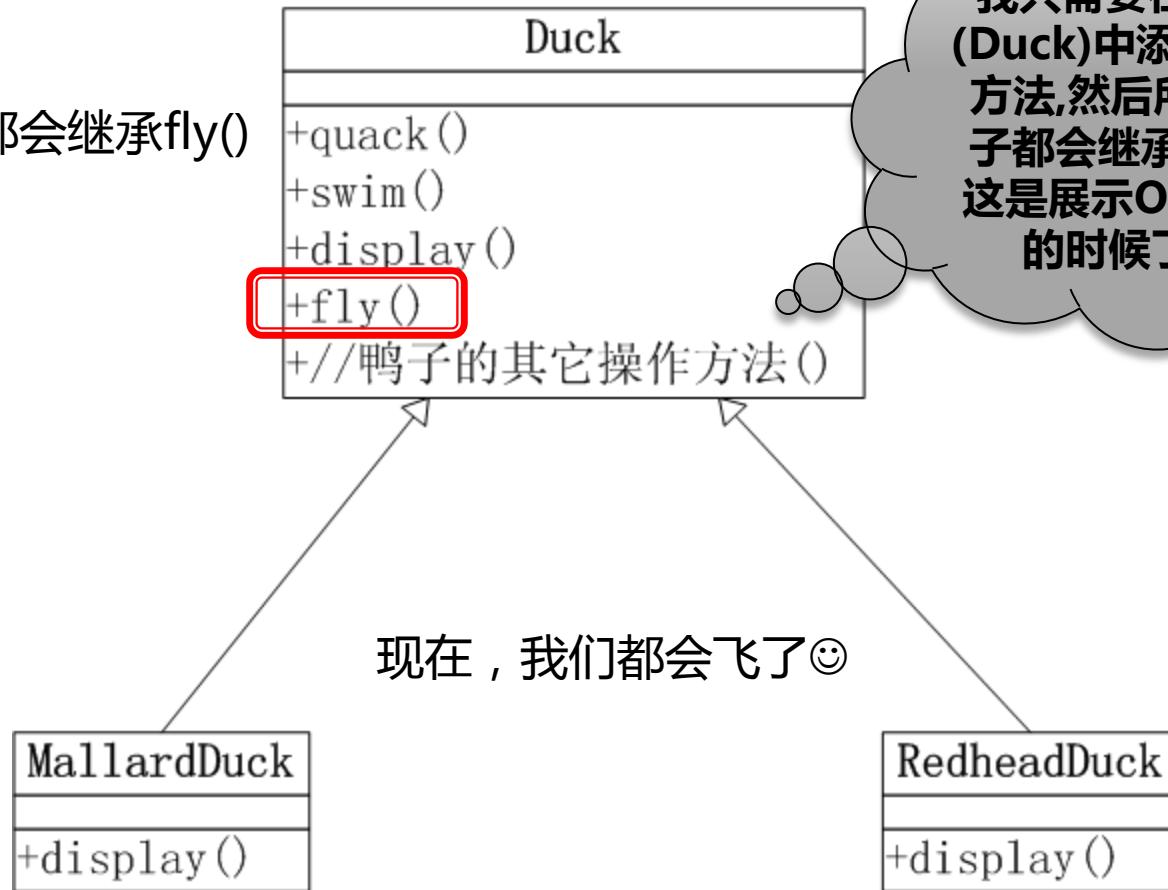
新的需求来了...

由于金融海啸的影响，全聚德公司的竞争压力增大。在为期一周的高尔夫假期兼头脑风暴会议之后，公司高层决定向潜在的客户展示一些令人印象深刻的元素，来提高公司低迷不振的股价...

让可爱的鸭子飞起来吧

需求：“全聚德烤鸭系统”添加“飞行”

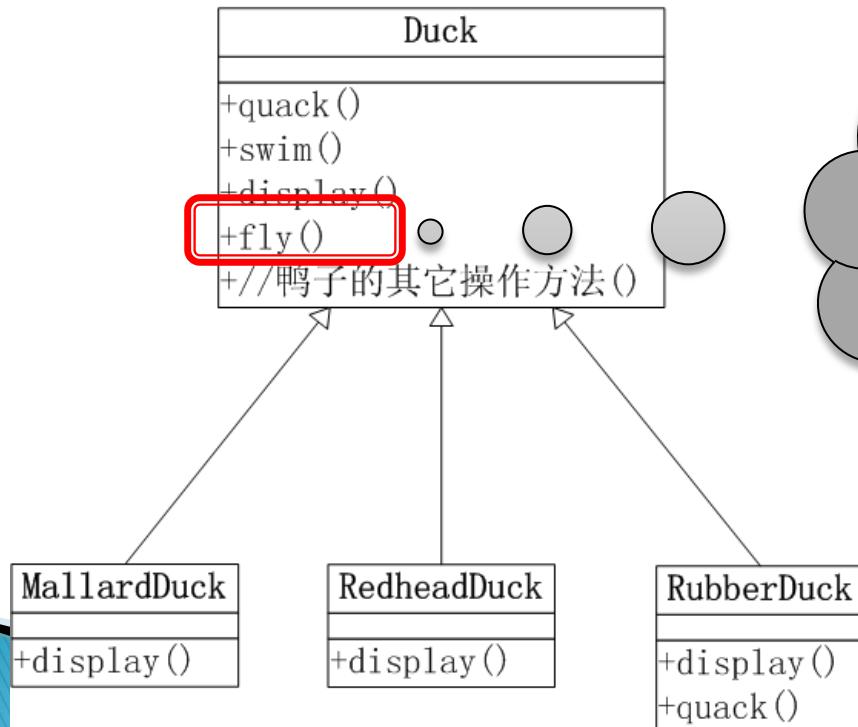
所有的子类都会继承fly()



现在，我们都会飞了☺

可怕的事情发生了...

Joe忽略了一件事，并非所有的Duck子类都会飞。如果在基类中加上新的行为，将会迫使某些不适合该行为的子类，也具有该行为；就本例来说，一些没有生命的鸭子（比如用于陪鸭子玩耍的“橡皮鸭”也会在天空中飞翔，但谁都知道，这是不合理的...），对代码所作的局部修改，影响层面可不只是局部！



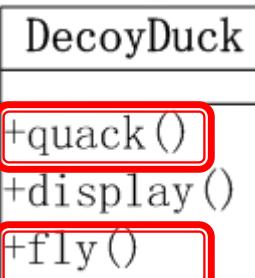
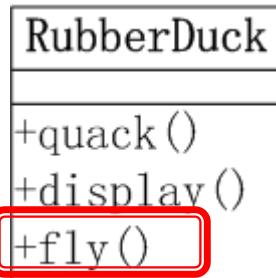
在超类中增加fly()，
将会导致所有的子
类都具备fly()，连
那些不具备fly()的
子类也无法免除

继承可以解决问题吗？

我可以直接覆盖橡皮鸭类中的fly()为空，神不知、鬼不觉，就好像一切都
没有发生过一样😊

可是如果以后加入
诱饵鸭
(DecoyDuck)，
又会如何呢？诱饵
鸭是木头假鸭，不
会飞也不会叫😊

覆盖fly()为空



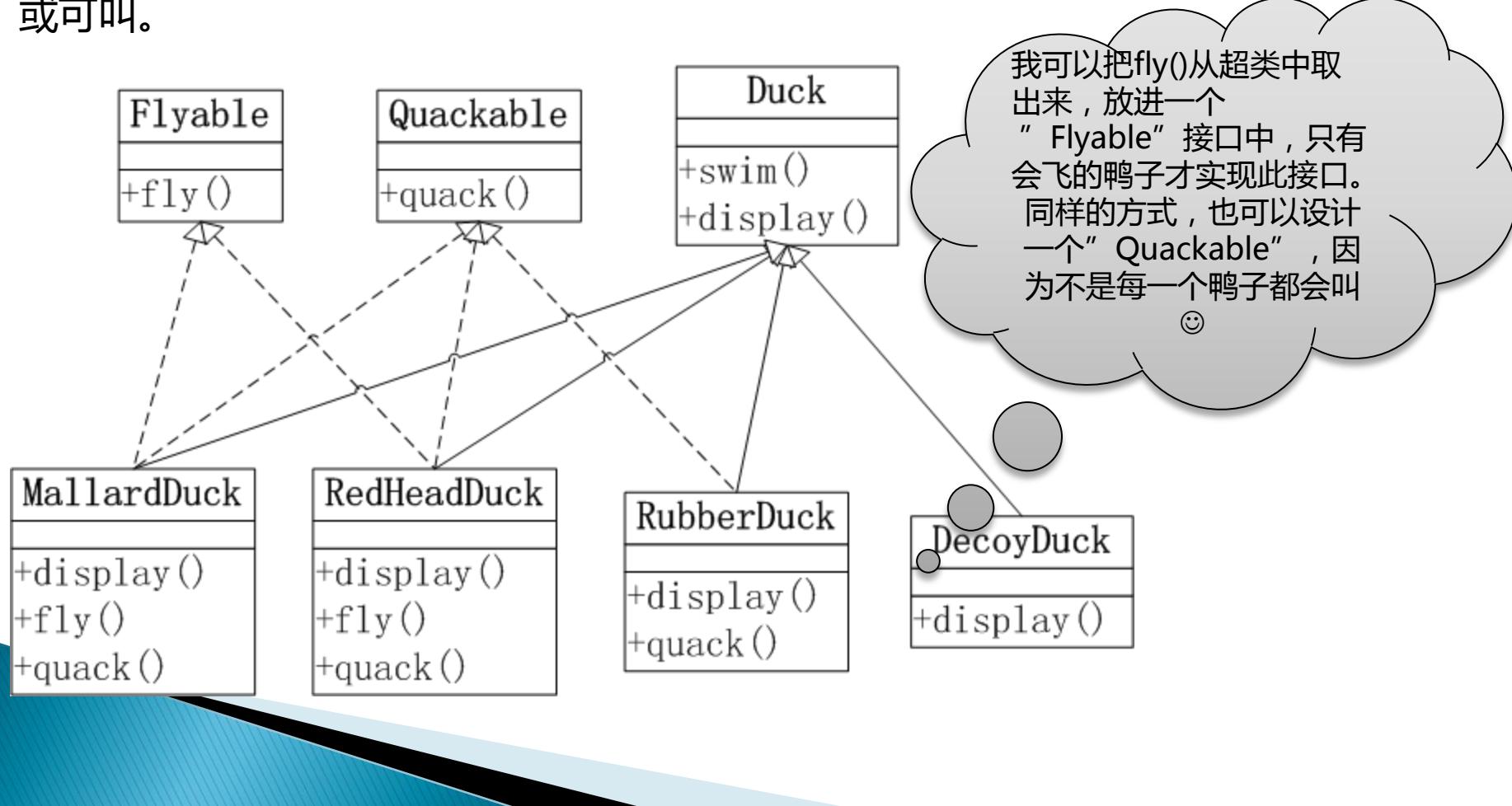
覆盖fly()和quack()为空

利用抽象类继承的方式来添加子类的行为，这会导致下列哪些缺点？

- 代码在多个子类中重复
- 运行时的行为不易改变
- 改变会牵一发而动全身，造成其它子类不想要的改变

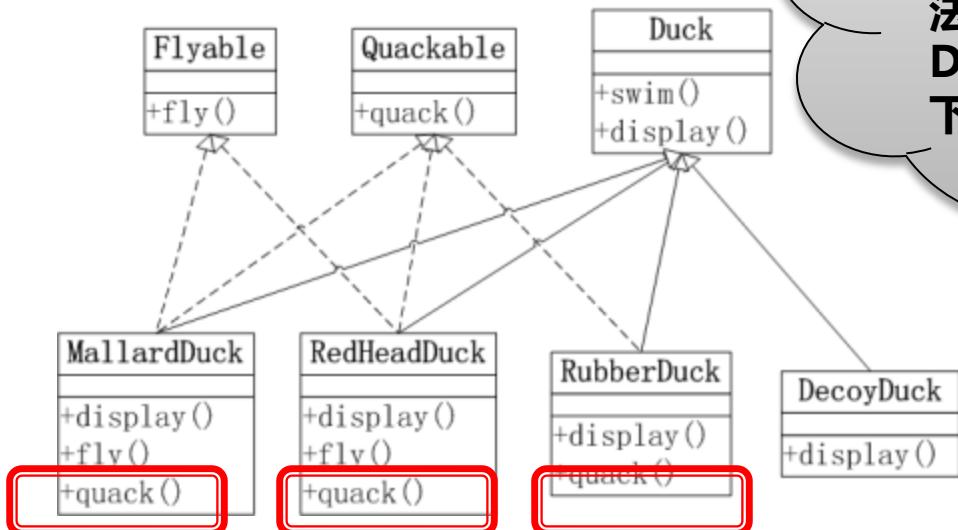
利用接口又如何？

抽象类继承可能不是答案，因为需求常常会改变；每当有新的鸭子子类出现，他就要被迫检查并可能需要覆盖fly()和quack().....这简直是无穷无尽的噩梦。我们需要的是一个更清晰的方法，让“某些”（而不是全部）鸭子类型可飞或可叫。



利用“接口”的代价

如果想改叫声“呱呱”为“吱吱”...



“全聚德”维护团队的反馈
这真是一个超笨的主意，你没发现这么一来重复的代码会增多吗？如果你认为覆盖几个方法不算什么，那么对于48个Duck的子类都需要稍微修改一下飞行的行为，你认为如何？！

所有实现此接口的方法都要修改！！！





回顾问题

我们知道，并非“所有”的子类都具有飞行和呱呱叫的行为，所以继承并不是适当的解决方式。虽然Flyable与Quackable可以解决“一部分”问题（即不会再有会飞的橡皮鸭），但是却造成代码无法复用。

继承并不能很好地解决问题，因为鸭子的行为在子类里会不断地改变。Flyable与Quackable接口一开始似乎还挺不错，但是Java接口不具有实现代码，所以继承自接口无法达到代码复用。

上述问题意味着：无论何时你需要修改某个行为，你必须得往下追踪并在每一个定义此行为的类中修改它，一不小心，就会犯下大错。

“设计原则”的宝库

来看看我们的第一个设计原则（后面还有更多😊）

封装变化：

找出应用中可能需要变化的部分，把它们独立出来，不要和那些固定不变的代码混合起来。



上面的概念很简单，几乎是每个设计模式背后的精神所在。所有的模式都提供了一套方法让“系统中的某部分改变不会影响其他部分”

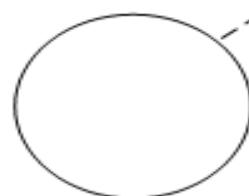
分开“变化”与“不变化”

我们知道Duck类内部的fly()和quack()会随着鸭子的不同而改变。为了要把这两个行为从Duck类中分开，我们将把它们从Duck类中取出来，建立一组新类来改变每个行为。

Duck类仍是所有鸭子的超类，但是飞行和呱呱叫的行为已经被去除，放在别的类结构中

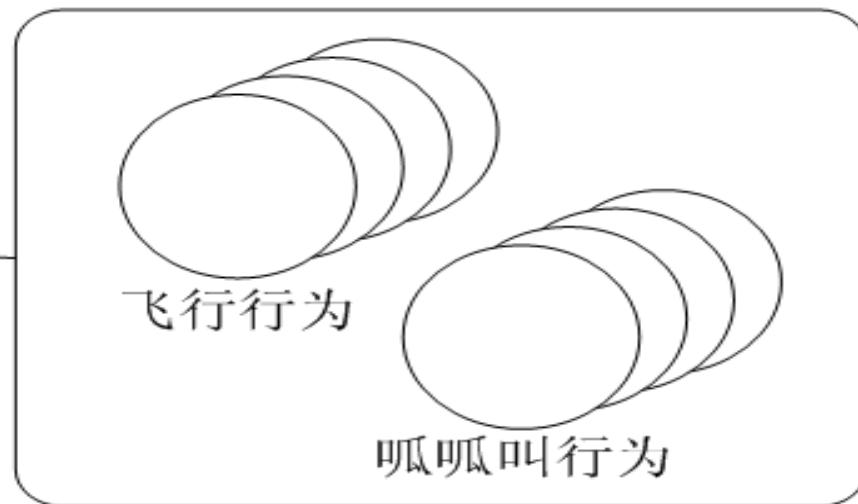
现在飞行和呱呱叫都有它们自己的类了

多种行为的实现被放在这里



取出易于变化的部分

鸭子



鸭子行为

有弹性的“设计原则”

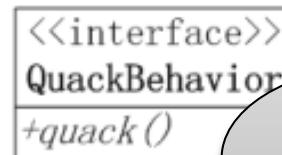
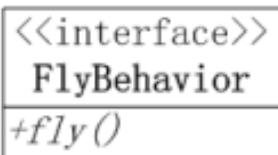
如何封装飞行和呱呱叫的行为呢？所谓弹性的设计即是可以让鸭子的行为可以动态改变。

接口编程原则：

针对接口编程，而不是针对实现编程。

这是一个接口，所有表示飞行行为的类必须实现它的fly()方法

呱呱叫的行为也一样，此接口包含一个必须要实现的quack方法



这样设计，可以让飞行和呱呱叫的动作被其他对象复用，因为这些行为已经与鸭子类无关了◎

而我们可以新增一些行为，不会影响到既有的行为类，也不会影响“使用”到表示飞行行为的类◎

FlyWithWings
+fly()

FlyNoWay
+fly()

Quack
+quack()

Squak
+quack()

Mute
+quack()

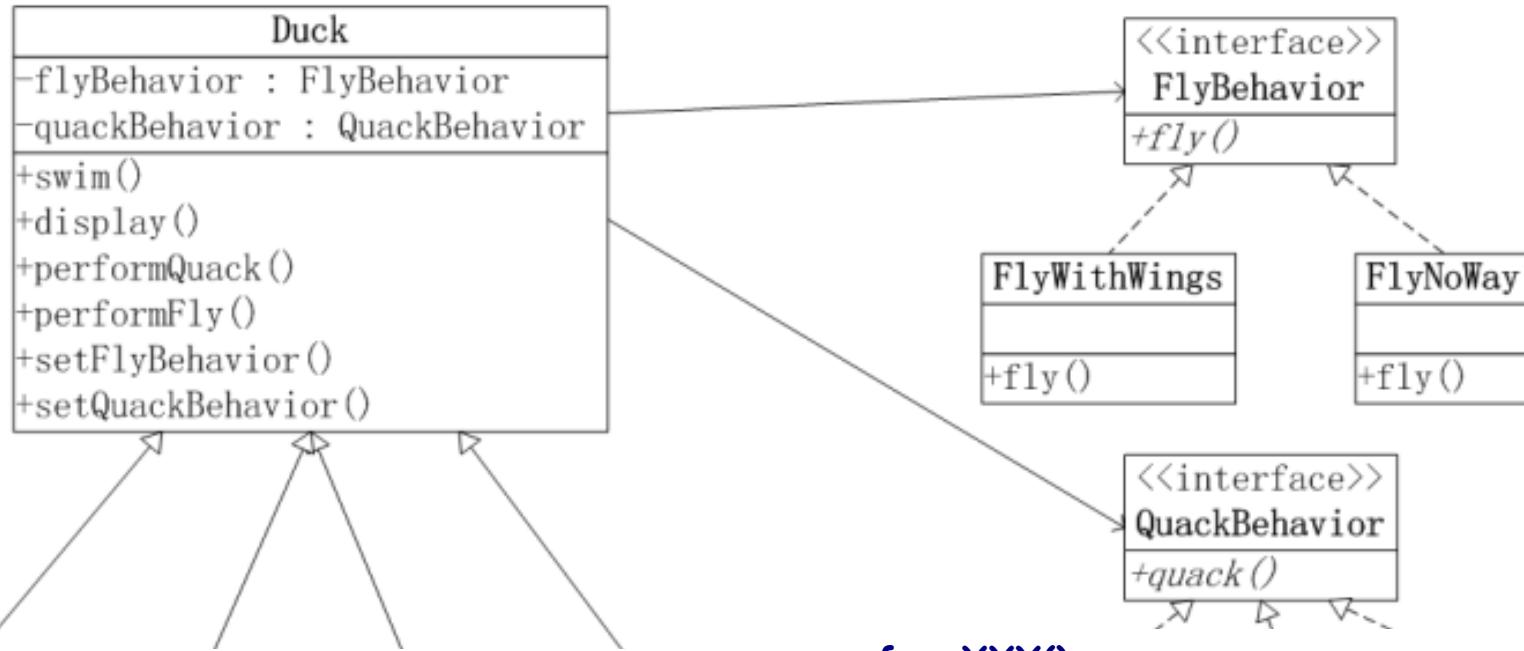
用翅膀飞行的行为 不会飞的行为

呱呱叫

吱吱叫

默不作声

新的设计



setXXXBehavior():

```
public void setFlyBehavior(FlyBehavior fb){  
    this.flyBehavior = fb  
}
```

```
public void setQuackBehavior(QuackBehavior qb){  
    this.quackBehavior = qb  
}
```

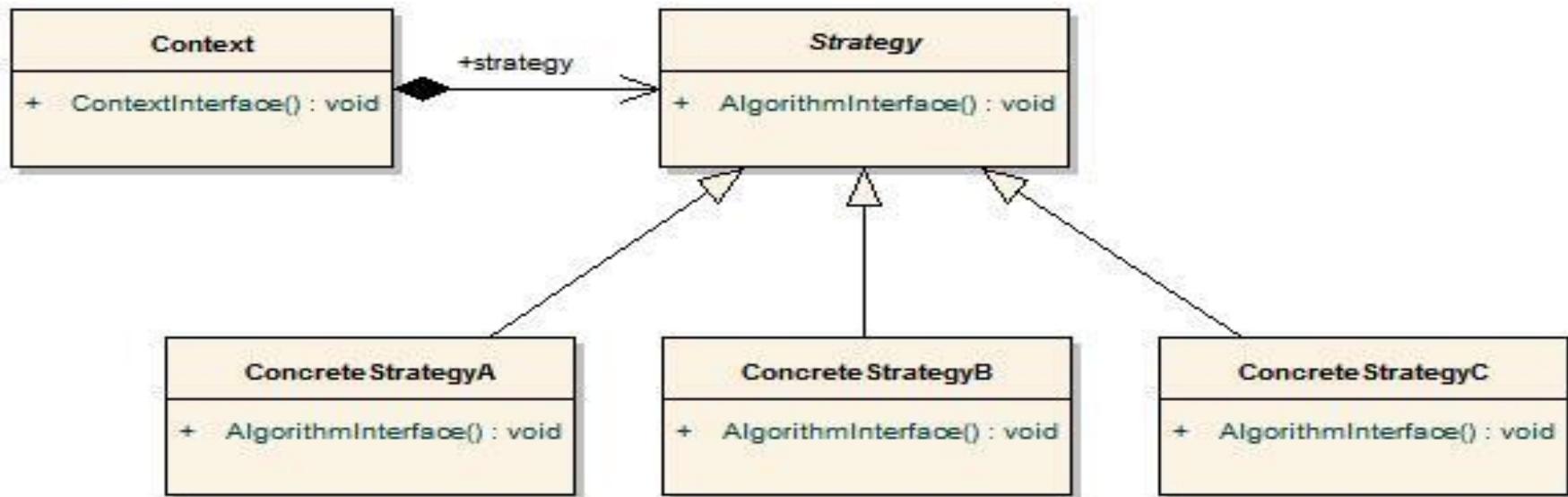
performXXX():

```
public void performFly{  
    flyBehavior.fly();  
}
```

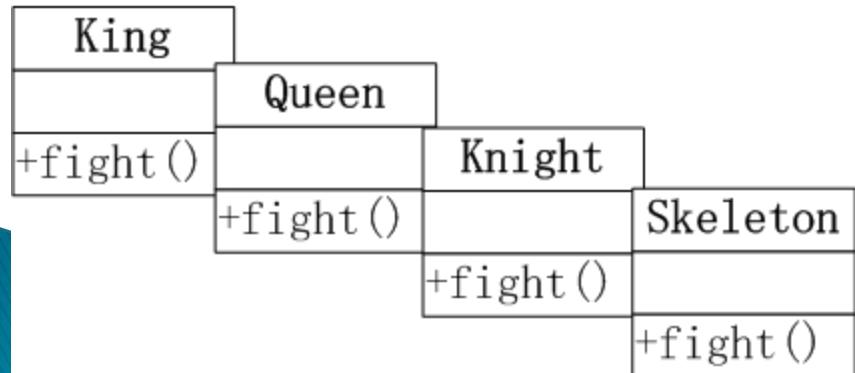
```
public void performQuack{  
    quackBehavior.quack();  
}
```

策略模式

你刚才用到了你的第一个设计模式...



“亚瑟王”的模式世界



Alibaba社区开发工程师：

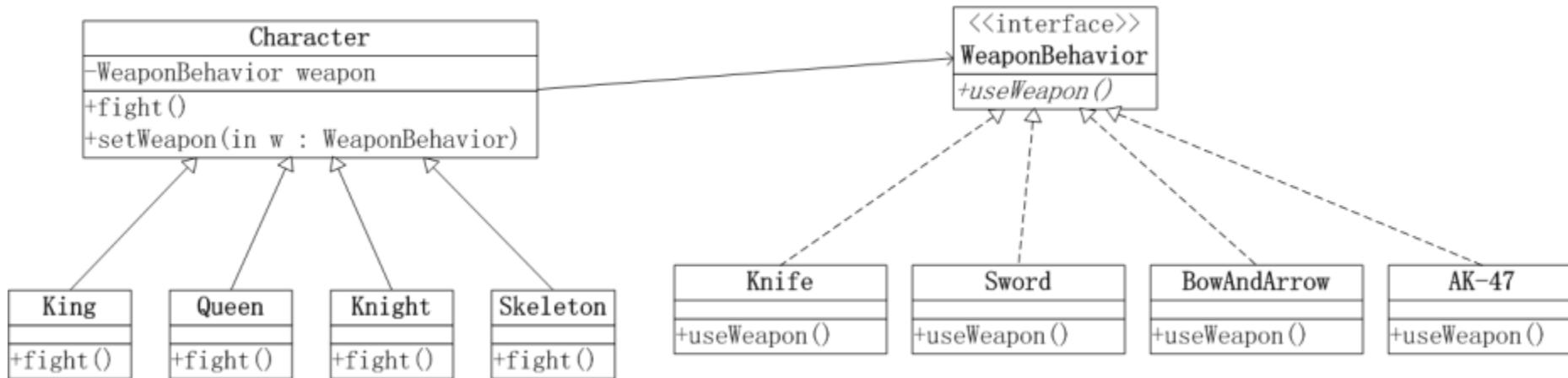
您好，我们对您开发的“全聚德”印象十分深刻，为此邀请您参与我们最新款的动作冒险游戏《圆桌骑士》进行设计。您将看到代表游戏角色的类和代表武器的类，**每名角色一次只能使用一种武器，但在游戏过程中可以切换、买卖、装备、舍弃武器，您能为我们的架构进行改进么？**

Capcom

Knife	Sword	BowAndArrow
+stab()		+shoot()
	+wield()	
		AK-47
		+fire()

一群“富有策略”的圆桌骑士

一个Character “有一个” WeaponBehavior，
在冒险过程中切换、装备、买卖或是丢弃武器
时，只要调用setWeapon方法即可



每一个具体的Character 都包含一个
weapon的属性以及一个fight方法，后者在其
内部调用WeaponBehavior接口的
useWeapon方法

任何武器都可以实现WeaponBehavior接口
包括拳头、弹弓、口水😊，只要它实现了
useWeapon方法。不同的武器的
useWeapon可能千差万别，比如霹刺、挥砍、
射击、喷吐等等

策略模式小结

1. 策略模式提供了管理相关的算法族的办法。策略类的等级结构定义了一个算法或行为族。恰当使用继承可以把公共的代码移到父类里面，从而避免重复的代码。
2. 策略模式提供了可以替换继承关系的办法。继承可以处理多种算法或行为。如果不是用策略模式，那么使用算法或行为的环境类就可能会有一些子类，每一个子类提供一个不同的算法或行为。
3. 使用策略模式可以避免使用多重条件转移语句。多重转移语句不易维护，它把采取哪一种算法或采取哪一种行为的逻辑与算法或行为的逻辑混合在一起，统统列在一个多重转移语句里面，比使用继承的办法还要原始和落后。
4. 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。
5. 策略模式造成很多的策略类

设计模式魅力——装饰模式





星巴克的烦恼

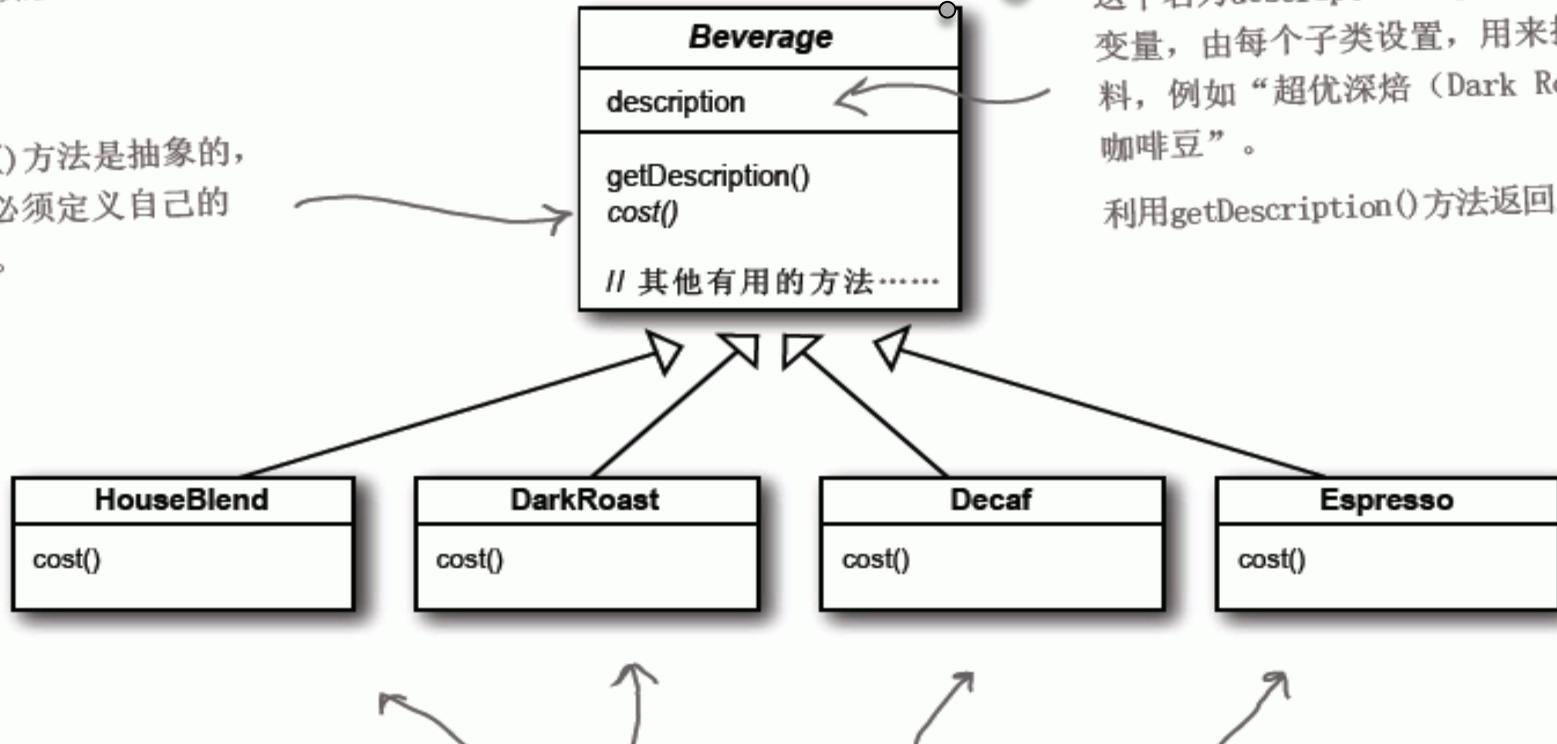
Beverage (饮料) 是一个抽象类，店内所提供的饮料都必须继承自此类。

cost() 方法是抽象的，子类必须定义自己的实现。

星巴克称霸的秘诀在于一套完美的OO设计☺

这个名为description (叙述) 的实例变量，由每个子类设置，用来描述饮料，例如“超优深焙 (Dark Roast) 咖啡豆”。

利用getDescription()方法返回此叙述。



每个子类实现cost()来返回饮料的价钱。

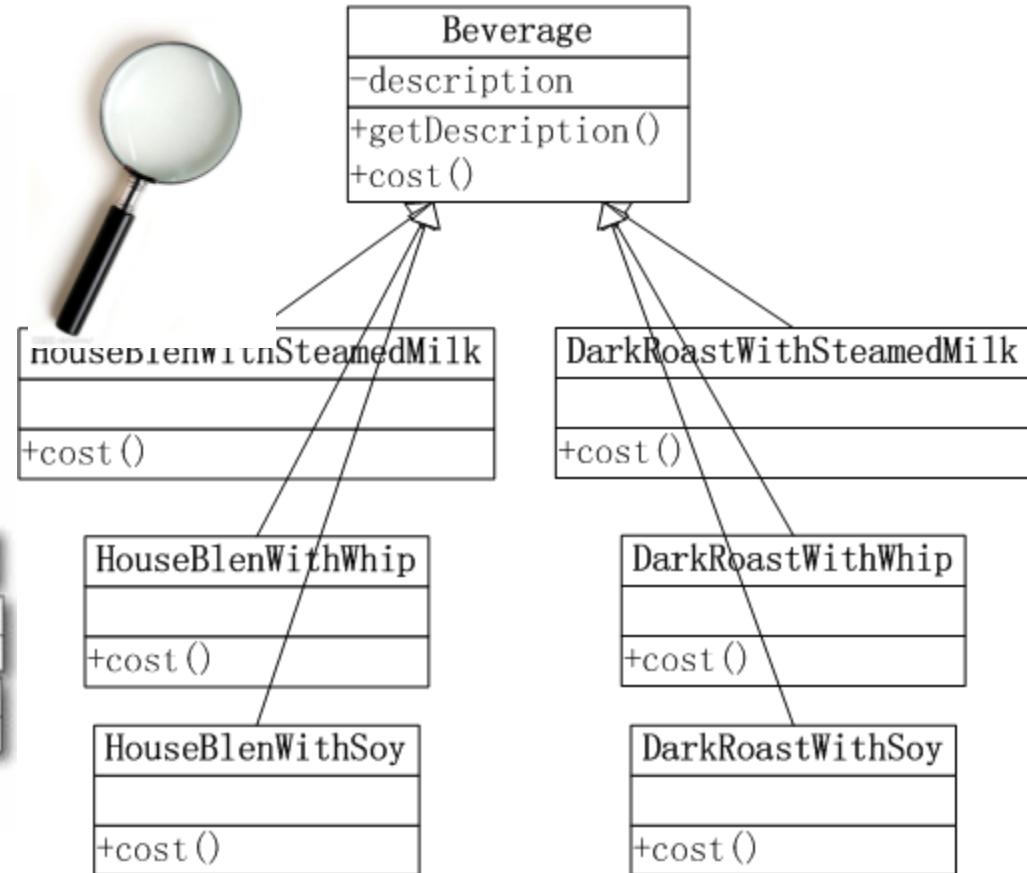
寻求“更灵活的菜单”



当我们来到滨江后，发觉他们更希望在咖啡中加入各种调料，例如蒸奶（*Steamed Milk*）、豆浆（*Soy*）、摩卡（*Mocha*）或是奶泡（*Whip*），并愿意支付相应的费用...

一个“很面向对象”的设计

这样的系统，
你愿意维护
吗？



如果牛奶的价格上涨，怎么办？
如果新增一种焦糖风味的调料，怎么办？
某些变态的客人要求两份牛奶外加一份豆浆，怎么办？

改良后的设计

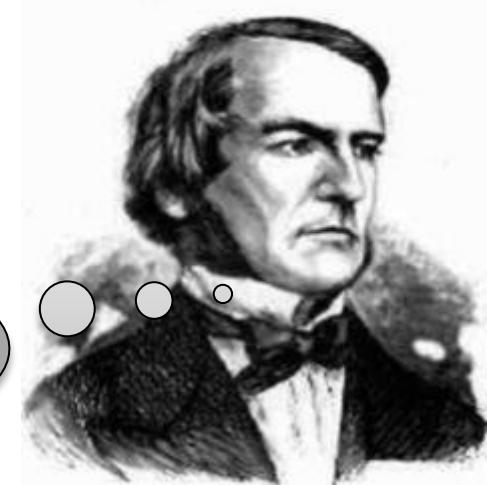
Beverage
-description
-milk
-soy
-mocha
-whip
+getDescription()
+cost()
+hasMilk()
+setMilk()
+hasSoy()
+setSoy()
+hasMocha()
+setMocha()
+hasWhip()
+setWhip()

前面的方法真是笨透了，利用实例变量、布尔值、get/set、继承，完全不必谢这么多类啊！

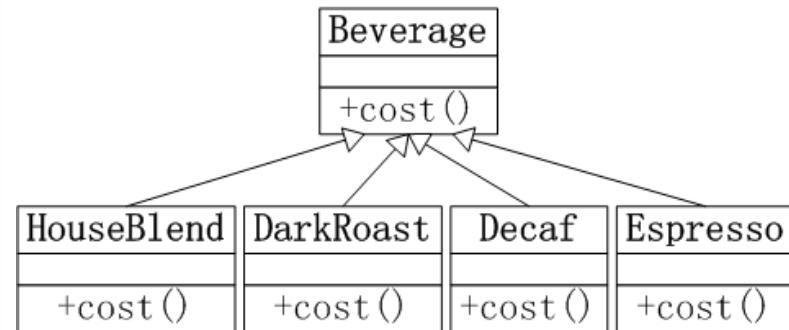
表示各种调料的布尔值

现在的cost方法不再是抽象方法了，基类提供了计算所有调料（milk、Soy等）价格的方法。子类仍将覆盖cost方法，添加基本饮料的价格（DarkRoast、Decaf等）

setXXX 和 hasXXX 分别用来设置和获取调料的布尔值



G.Boole



现在的咖啡可以随意添加调料，并且可以通过调用子类和父类的 cost 方法结账

布尔的设计有哪些问题？

- 调料价钱的改变会使我们更改现有的代码。
- 一旦出现新的调料，我们就被迫修改*Beverage*类，并改变*cost*方法。
- 以后若决定出售新的饮料（例如：可乐），某些调料（比如豆浆）并不适合。但在当前的设计中，可乐若是继承自*Beverage*类，则必然含有*setSoy*（加豆浆）方法。
- 若某些变态的用户需要加双份的摩卡调料，怎么办？
- 基类*cost*方法中包含大段的*if/else*使我们闻到了代码的“坏味道”。

```
public double cost{
    double payment = 0;
    if (hasMilk) //如果加牛奶，额外付¥1.9
        payment += 1.9;
    if (hasSoy) //如果加豆浆，额外付¥0.85
        payment += 0.85;
    if (hasMocha) //如果加巧克力，额外付¥2.4
        payment += 2.4;
    if (hasWhip) //如果加奶泡，额外付¥1.2
        payment += 1.2;
    return payment;
}
```

面向过程设计和面向对象设计的主要区别是：是否在业务逻辑层使用冗长的*if else*判断。

——《重构》

ocp设计原则



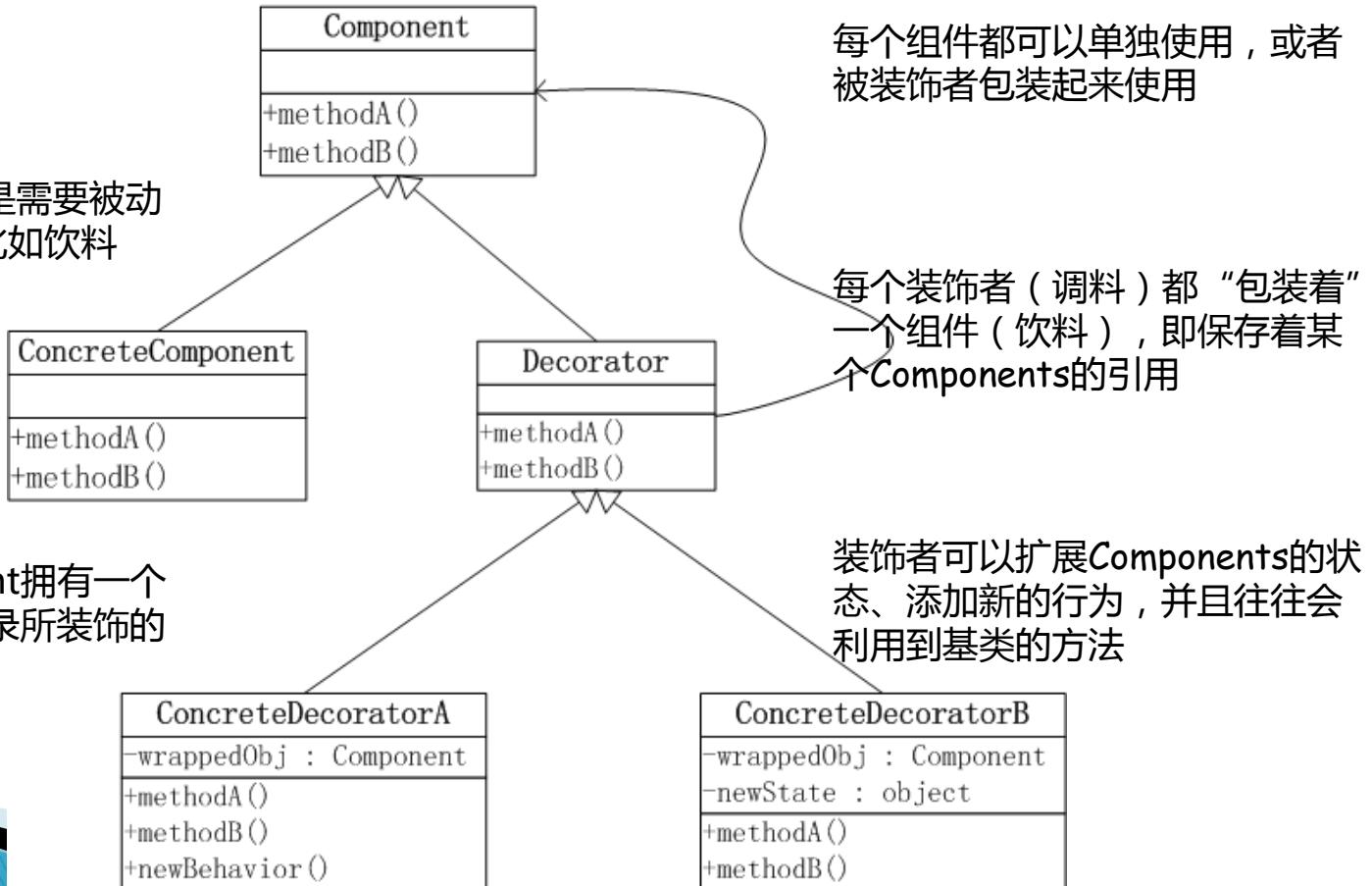
开发 - 关闭原则：
类应该对扩展开放，对修改关闭。

装饰者模式

装饰者模式动态地将职责附加到对象上。对于扩展功能来说，装饰者提供了比继承更有弹性的替代方案

— GOF

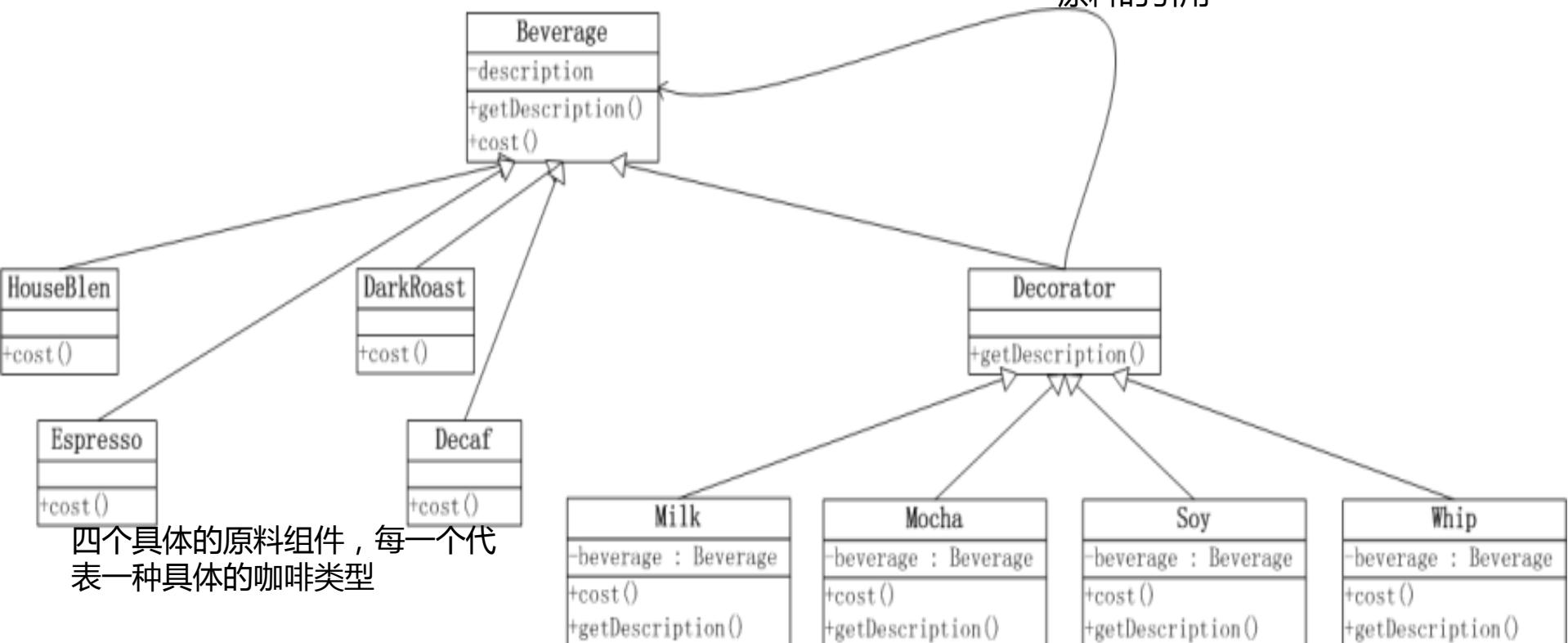
ConcreteComponent是需要被动态添加行为的对象，比如饮料



星巴克的装饰者模式

Beverage相当于前面的Component类，作为被包装的原料类

一个用于包装的抽象类，保存了原料的引用



四个具体的原料组件，每一个代表一种具体的咖啡类型

所有调料都继承自Decorator类，也变相地继承了Beverage类，它们实现了cost方法（计算价格）以及getDescription方法（更新描述）

“装饰咖啡”的代码

Beverage是一个抽象类

```
public abstract class Beverage {  
    String description = "Unknown Beverage";  
    public String description() {  
        return description;  
    }  
    public abstract double cost();  
}  
getDescription已经实现了，但cost方法必须在子类中实现
```

```
public class Espresso extends Beverage {  
    public Espresso() {  
        description = "Espresso";  
    }  
    public double cost() {  
        return 1.99;  
    } Espresso是一种饮料，它包含自己的  
    描述和价格  
}
```

```
public class HouseBlend extends Beverage {  
    public HouseBlend() {  
        description = "House Blend";  
    }  
    public double cost() {  
        return 0.89;  
    }  
}
```

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

Decorator必须能够取代Beverage，这样我们才能“对装饰过的东西再进行装饰”

Mocha是一个装饰者，它保存一个被装饰者的引用，这里采用的是构造器注入的方式

```
public class Mocha extends Decorator {  
    Beverage beverage;  
  
    public Mocha(Beverage beverage) {  
        this.beverage = beverage;  
    }  
  
    public String getDescription() {  
        return beverage.getDescription() + ", Mocha";  
    } getDescription可以获取咖啡的完整描述，  
    例如 "DarkRoast, Mocha"  
    public double cost() {  
        return 0.21 + beverage.cost();  
    } 要计算咖啡加Mocha的价钱，首先把调用委托给被装  
    饰者，然后再加上Mocha的价钱，得到最后结果  
}
```

首先我们生成一个DarkRoast对象

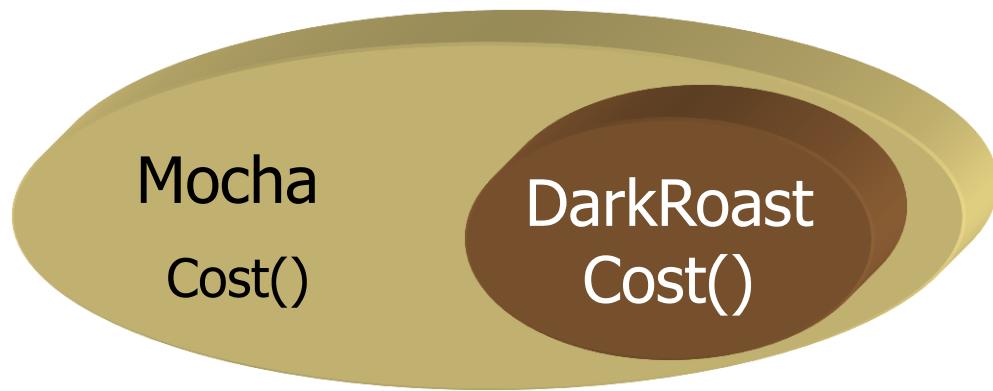


DarkRoast
Cost()

DarkRoast继承了Beverage，拥有一个计算饮料价格的方法cost()。

然后

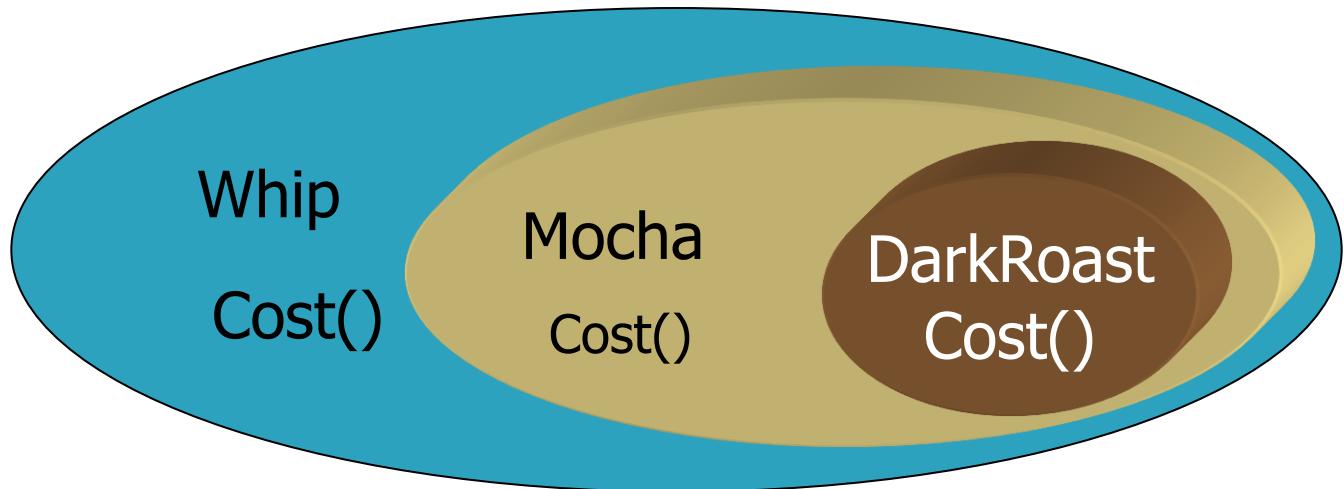
顾客想要穆哈，所以我们创建一个**Mocha**对象，并用它包装**DarkRoast**.



Mocha对象是装饰者，他与被它装饰的对象**DarkRoast**具有相同的类型（是**Beverage**的子类），也有一个**cost()**方法。

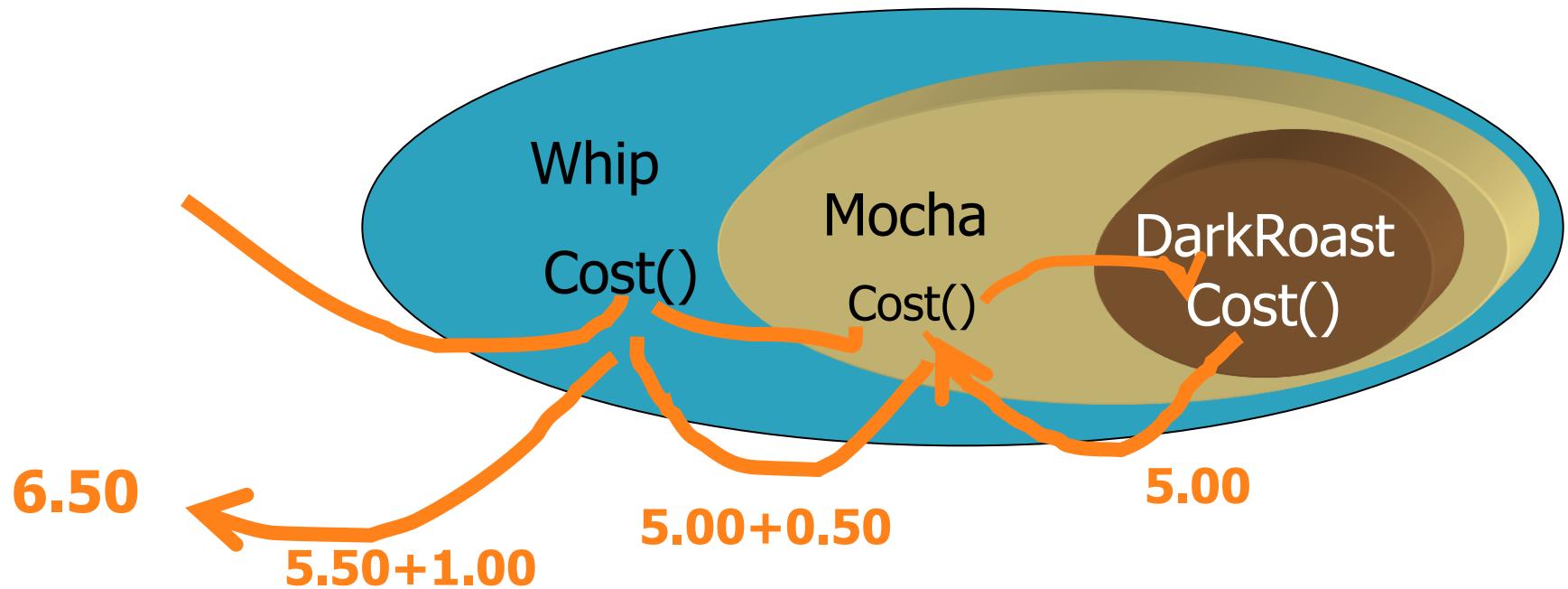
再然后

顾客还想要蛋奶，所以我们创建一个Whip对象，并用它包装Mocha.



whip对象是装饰者，他与被它装饰的对象DarkRoast具有相同的类型，也有一个cost()方法。

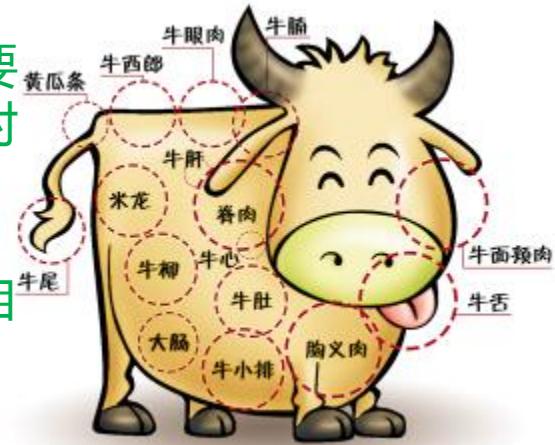
现在，要计算饮料的价格



调用最外层的装饰者whip的cost(),whip再将计算任务委派给被它包装的对象，得到一个价格后，再加上whip自己的价格...

庖丁解 “Decorator”

- 装饰者和被装饰着就有相同的超类
- 可以同时用一个或多个装饰者包装一个对象
- 既然装饰者和被装饰者具有相同的超类，所以在任何需要原始对象（被装饰者）的时候，都可以用已经装饰过的对象来代替它
- 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
- 对象可以在任何时候被装饰，所以可以再运行时动态地、无限地用你喜欢的装饰者来进行包装
- 装饰者模式会产生大量的小类，增加调用此API的程序员困扰和学习成本

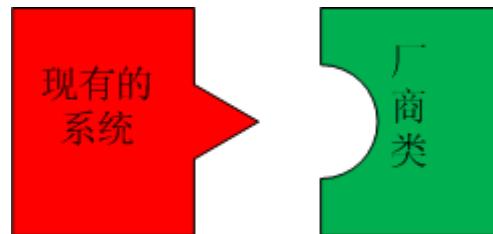


设计模式魅力——适配器

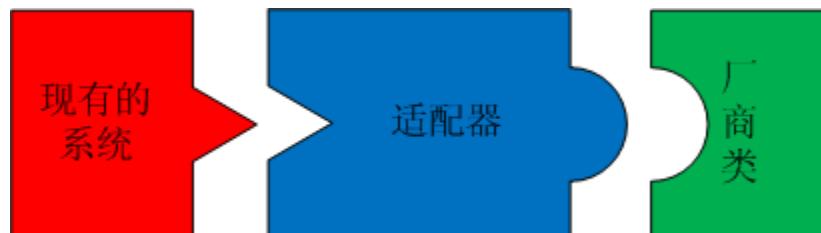
适配器

When in Rome, do as the romans 入乡随俗

“面向对象牌”适配器



这两个接口无法匹配，所以无法工作



这个适配器实现了“现有的系统”的接口，也能和厂商的接口沟通



现在，一切都可以工作了
(注意：很多厂商自己就提供了适配器类)

不需要改变代码

新的代码

不需要改变代码

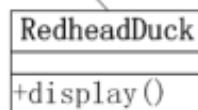
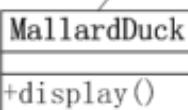
还记得“全聚德”的鸭子们吗？



事实上，我们现在又碰到了麻烦。由于现在鸭肉紧缺，我们准备用火鸡来代替。但是，我发觉你们的系统似乎失灵了



你要搞清楚，我们从不呱呱叫(quack)，只会咯咯叫(gobble)；我们的飞行距离很短；另外，你见过会游泳的火鸡吗？！



鸭子和火鸡的区别

```
public interface Duck {  
    public void quack();  
    public void fly();  
    public void swim();  
}
```

```
public class MallardDuck implements Duck {  
    public void quack() {  
        System.out.println("Quack");  
    }  
  
    public void fly() {  
        System.out.println("fly");  
    }  
  
    public void swim() {  
        System.out.println("swim");  
    }  
}
```



我会呱呱叫、
长途飞行以及
游泳

```
public interface Turkey {  
    public void gobble();  
    public void fly();  
}
```

```
public class WildTurkey implements Turkey {  
    public void gobble() {  
        System.out.println("Gobble");  
    }  
}
```

```
public void fly() {  
    System.out.println("Flying a short distance");  
}
```



我会咯咯叫、
短距离飞行；
但不会游泳

试试最新的“火鸡适配器”

由于“全聚德的鸭子展示系统”只认识Duck接口，因此我们实现了一个适配器：

```
public class TurkeyAdapter implements Duck {  
    Turkey turkey;
```

首先，你必须实现你想要转换成的、客户所期望看到的类型接口（Duck）

```
    public TurkeyAdapter(Turkey turkey){  
        this.turkey = turkey;  
    }
```

必须采用某种方式，获得需要适配的对象的引用，比如构造器注入...

```
    public void quack(){  
        turkey.gobble();  
    }
```

有些方法需要简单转换，比如用火鸡的咯咯叫（gobble）来代替鸭子的呱呱叫（quack）

```
    public void fly(){  
        for (int i=0; i<5; i++)  
        {  
            turkey.fly();  
        }  
    }
```

有些方法需要复杂抓换，比如，由于火鸡的飞行距离较短，要达到鸭子的飞行效果，火鸡需要加倍的努力（调用fly五次）

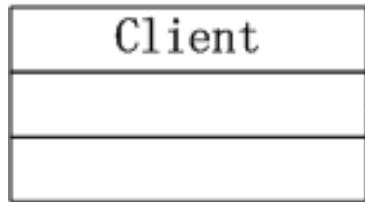
```
    public void swim(){  
        System.out.println("No Turkey can SWIM!!!");  
    }
```

还有些方法无法适配，这是可以覆盖为空、或者抛出异常、或者干脆像火鸡一样说：“这个我干不了” 😊

适配器模式

适配器模式将一个类的接口，转换成客户期望的另一个接口。适配器让原来不兼容的类可以合作无间。

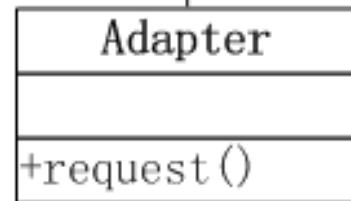
—— GOF



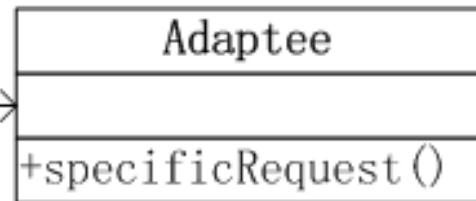
客户只看到目标接口



适配器实现目标接口



适配器与被适配器组合



所有的请求都委托给被适配器

适配器模式的工作过程



- 客户通过目标接口调用适配器的方法对适配器发出请求
- 适配器使用被适配者接口把请求转换为被适配者的一个或多个调用接口
- 客户接收到调用的结果，但并未察觉到这一切是适配器在起转换作用
- 当客户没有指定所需要的接口时，我们仍然可以应用 *Adapter* 模式，并使用现有类的实例来创建一个新的客户子类。这种方法会创建一个对象适配器，将客户调用转发给现有类的实例。但是这种方法也很危险，特别是当我们没有（或者不可以）重写客户可能调用的所有方法时。

设计模式魅力——观察者

观察者

观察，观察，再观察 —— 巴甫洛夫

《观察者日报》



Hi All :

总部设在纽约的《观察者日报》急需新增一个电子版报纸系统，要求订阅电子版报纸的用户可以实时的获取最新报纸。

系统需要有注册用户和注销用户功能。当有最新的报纸出版时，所有订阅了报纸的用户可以自动获得最新版报纸。

系统不需要了解究竟是哪些用户订阅了报纸，它负责的只是发布、发布、再发布！

用户可以随时退订报纸，之后也可以重新再订阅，考虑到每天都有数以千计的用户订阅/退订，你不会想让我们每次都修改发报的代码吧？

《观察者日报》

一个错误的范例

```
public void publishNewsPaper() {  
    NewsPaperOffice newPaperOffice = new NewsPaperOffice("观察者日报社");  
    Paper = newsPaperOffice.getPaper("《观察者日报》");  
  
    Google.send(paper);  
    Yahoo.send(paper);  
    Sina.send(paper);  
}
```

这样的设计有什么问题？

如果Alibaba也想订阅报纸，怎么办？

如果Yahoo想订阅两份报纸（Yahoo与Yahoo中国各一份），怎么办？

如果Google想退订《观察者日报》，怎么办？

如果Google退订之后又想再订阅《观察者日报》，怎么办？

按照目前的设计，只有一个方法，就是
修改publishNewsPaper的代码

深入研究“订阅报纸”的业务

- ① 报社的业务就是出版报纸
- ② 向某家报社订阅报纸，只要它们有新报纸出版，就会给你送来
- ③ 当你不想再看报纸的时候，取消订阅，就不会再有新报纸送来
- ④ 只要报社还在运营，就会一直有人（或单位）向他们订阅报纸或取消订阅报纸

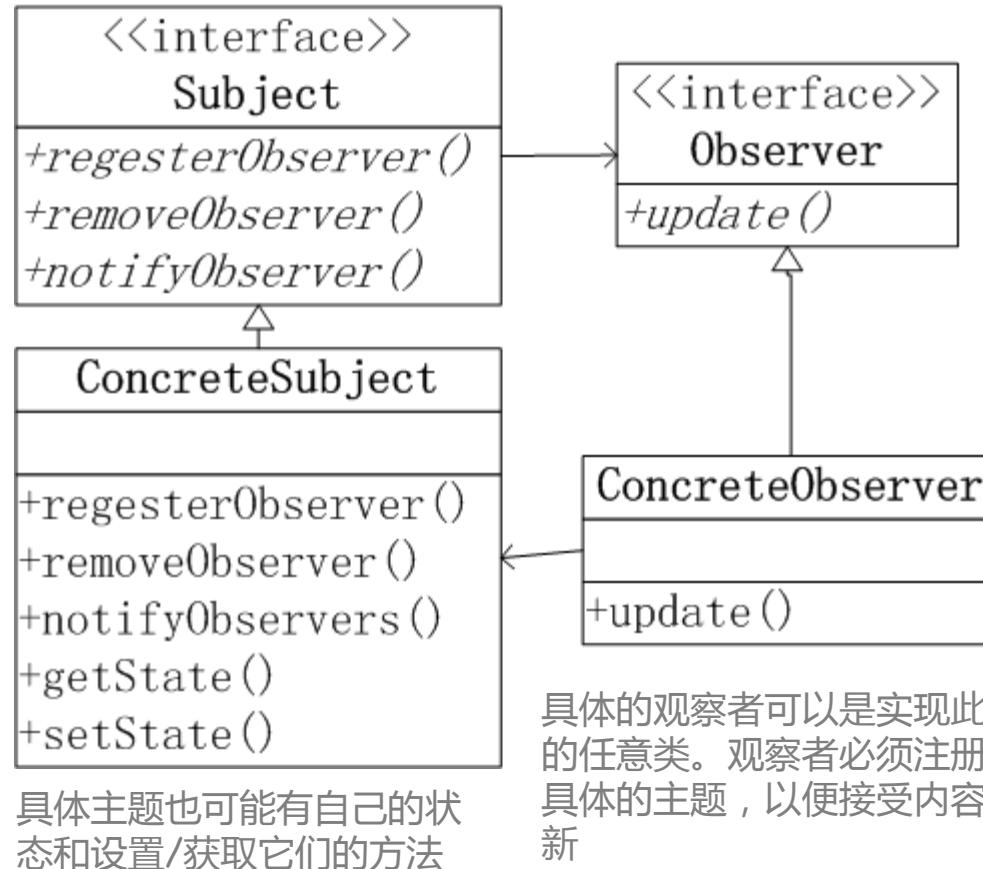


出版商 + 订阅者 = 观察者模式

每个主题可以有很多个观察者

这是主题接口，对象使用此接口注册为观察者，或者把自己从观察者队列中删除

一个具体主题总是实现主体接口，除了register和remove之外，还有notify，此方法用于在状态改变时更新所有已注册的观察者



所有潜在的观察者都必须实现此接口的update方法。当主题状态改变时，该方法就会被调用

具体的观察者可以是实现此接口的任意类。观察者必须注册某个具体的主题，以便接受内容的更新

松耦合的威力

耦合一内聚原则：

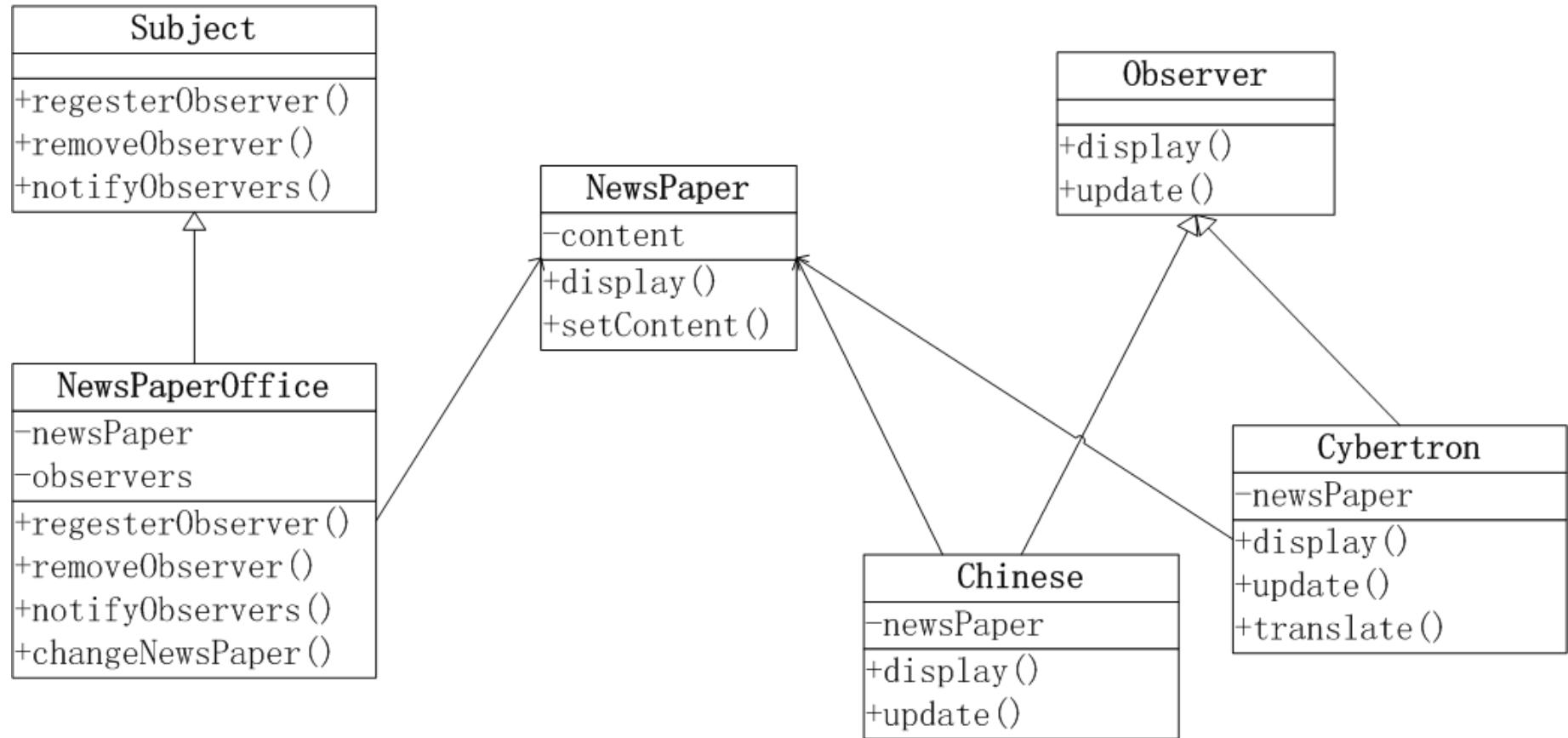
为了交互对象之间的松耦合设计而努力

松耦合设计的特点

- 主题和观察者之间松耦合，意味着它们虽然可以互相作用但是不太清楚彼此的细节
- 任何时候都可以增加新的观察者，因为主题唯一依赖的是一个Observer接口的列表
- 有新类型的观察者出现时，主题的代码不需要因为兼容性的问题而修改
- 改变主题或是观察者中的其中一方，并不会影响另一方，因为两者是松耦合的
- 可以独立地复用主题或观察者



“松耦合”的《观察者日报》



“松耦合”的《观察者日报》

NewsPaperOffice

-newsPaper
-observers
+regesterObserver()
+removeObserver()
+notifyObservers()
+changeNewsPaper()



```
List<Observer> observers;  
Newspaper newspaper;  
public NewspaperOffice()  
{  
    observers = new List<Observer>();  
}  
public void changeNewspaper(Newspaper n)  
{  
    newspaper = n;  
    notifyObservers();  
}  
public void registerObserver(Observer o)  
{  
    observers.Add(o);  
}  
public void removeObserver(Observer o)  
{  
    observers.Remove(o);  
}  
public void notifyObservers()  
{  
    foreach (Observer var in observers)  
    {  
        var.update(newspaper);  
    }  
}
```

“松耦合”的《观察者日报》

Chinese
-newsPaper
+display()
+update()

《观察者日报》最初是面向中国地区的，有许多中国订阅者



“松耦合”的《观察者日报》

Cybertron

-newsPaper

+display()

+update()

+translate()

《观察者日报》采用新的系统后大获成功，威震天感觉应该让所有霸天虎人人手一份，所以又来了许多霸天虎订阅者。这时问题出现了，日报没有霸天虎语言版的，怎么办呢？为了让所有人都能读懂报纸，必须在霸天虎订阅者中添加一个翻译方法，将中文版的《观察者日报》翻译成霸天虎版日报。

```
class Cybertron extends Observer
{
    Newspaper newspaper;

    public Newspaper translate(Newspaper n)
    {
        Newspaper tmp = n;
        //translate the newspaper.
        tmp.setContent("@###(@*&@&^@**@(!)!)#(@&@$@*#$^*@(");
        return tmp;
    }

    public void update(Newspaper n)
    {
        this.newspaper = n;
        display();
    }

    public void display()
    {
        System.out.println("I am a Cybertron and I have got the
newspaper");
        translate(newspaper).display();
    }
}
```



《观察者日报》的扩展性吧

《观察者日报》最近新推出了一个版面——“身边的特异功能者”，Sylar发现了这个版面后兴奋的三天没合眼，立即订阅了报纸，只要每天订阅这个报纸就可以轻而易举的找到那些超能者，挖开他们的大脑获得他们的能力。Sylar立即订阅了一份《观察者日报》。现在Sylar也能不断收到最新的报纸了，其他Hero们要小心了😊



Sylar
-newsPaper
+display()
+update()

```
class Sylar extends Observer
{
    Newspaper newspaper;
    Observer Members;

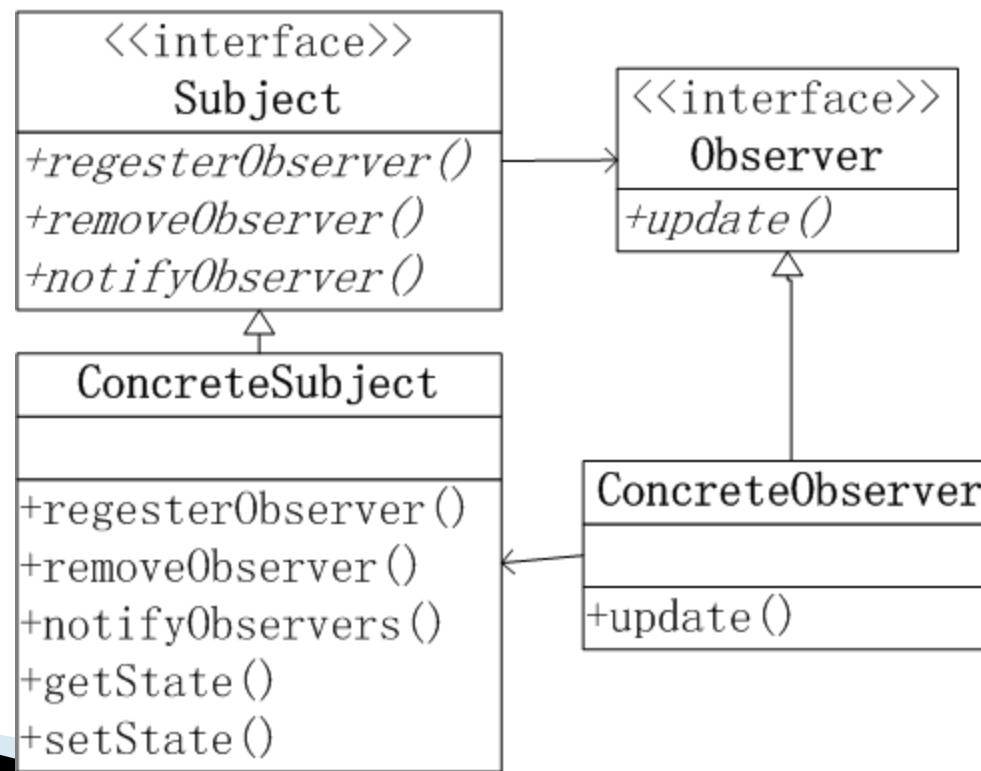
    public void update(Newspaper n)
    {
        this.newspaper = n;
        display();
    }

    public void display()
    {
        System.out.println("I am Sylar and I have
                           got the newspaper");
        this.newspaper.display();
    }
}
```

定义观察者模式

观察者模式定义了对象之间的一对多关系，这样一来，当一个对象改变状态时，它的所有依赖者都会接收到通知，并自动更新

—— GOF



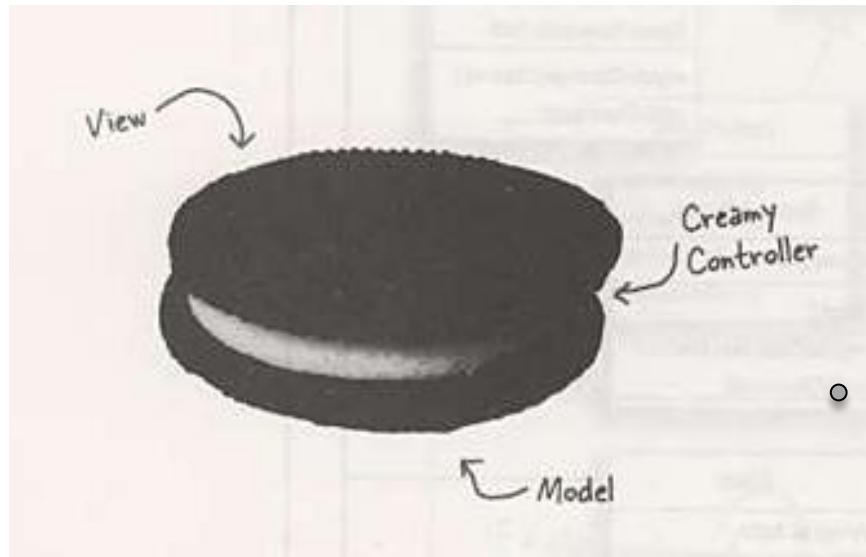
设计模式魅力——复合模式

复合模式

团结就是力量。 —— 毛泽东

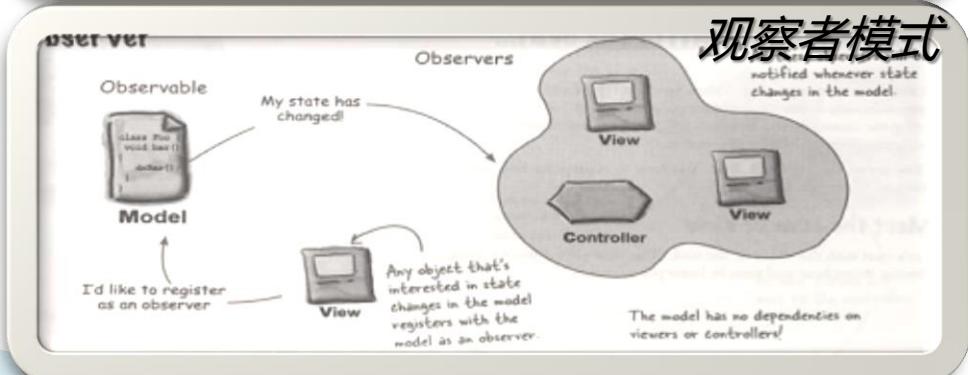
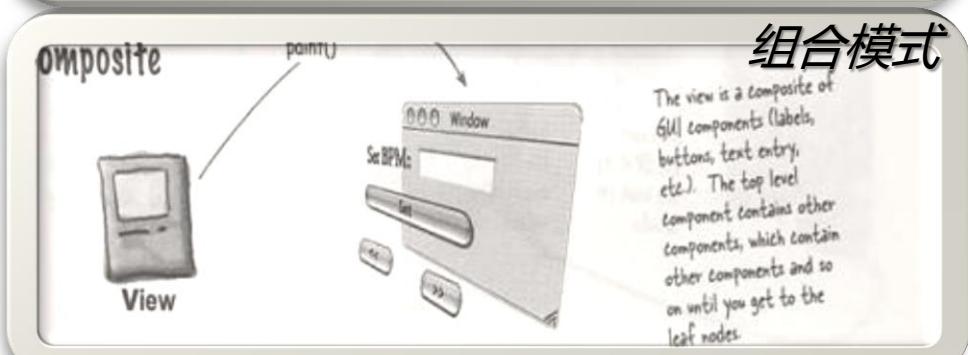
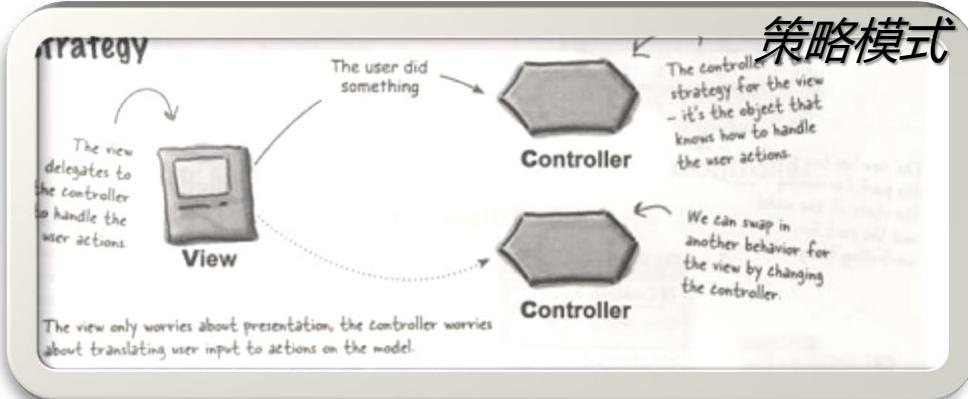
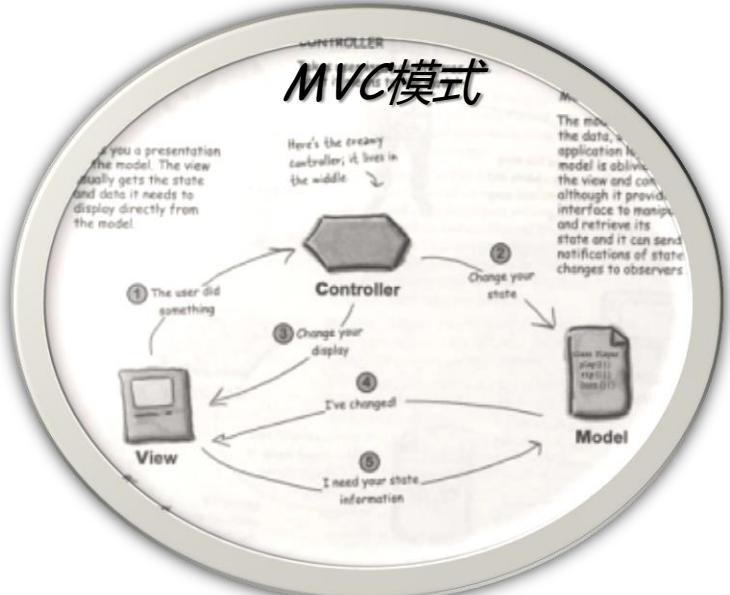
模式居然可以携手合作

复合模式 在一个解决方案中结合两个或多个模式，以解决一般或重复发生的问题



MVC是复合
模式之王

MVC中的设计模式



问题2：

- ▶ 简单解析一下MVC中用到的设计模式，以及这些模式的合作。