设计模式

# 零．一些废话

## 1.设计模式是啥?

对于一类问题，软件设计的特定套路。找出变化的部分，套路化的修改。

## 2.设计模式解决的局限性?

不能解决性能问题，只能解决软件复用，代码可读性的问题。

滥用模式，会导致过度设计，代码可读性和复杂性会上升，反而不好维护。

## 3.能正确的使用设计模式需要什么知识？

最重要的，产品知识，知道产品哪里容易变，哪里容易可能出现扩展。

知道哪些代码我们有权限修改，哪些代码我们只有使用和阅读的权限。

次重要的，代码知识，知道哪些设计模式的基本套路。

## 4.设计模式能帮到我们什么？

1.更好的沟通。

2.阅读代码，比如很多framework的代码，都是使用了特定的设计模式的，比如context使用了装饰，wifi模块使用了状态模式。熟悉设计模式，就很容易掌握代码结构，排查问题。

3.设计代码。

# 一.单例模式singleton

## 1最简单的单例

class Singleton{  
 //私有构造，让外部不能调用构造函数，新建这个对象  
 private Singleton(){}  
 //私有静态实例，方便静态方法调用  
 private static Singleton *singleton*;  
 //返回这个实例的接口  
 public static Singleton getInstance(){  
 //多线程的情况下，这里可能出现多个实例  
 if(*singleton*==null){  
 *singleton*=new Singleton();  
 }  
 return *singleton*;  
 }  
}

## 2双重校验锁

class Singleton{  
 //volatile防止编译器修改指令次序  
 private volatile static Singleton *singleton*;  
 private Singleton(){}  
 public static Singleton getInstance(){  
 if(*singleton*==null){  
 synchronized(Singleton.class){  
 if(*singleton*==null){  
 *singleton*=new Singleton();  
 }  
 }  
 }  
 return *singleton*;  
 }  
}

## 3.急切实例化

class Singleton{  
 private static Singleton *singleton*=new Singleton();  
 private Singleton(){}  
 public static Singleton getInstance(){  
 return *singleton*;  
 }  
}

## 4.静态内部类

class Singleton{  
 private static class SingletonHolder{  
 private final static Singleton *IN*=new Singleton();  
 }  
 private Singleton(){}  
 public static Singleton getInstance(){  
 return SingletonHolder.*IN*;  
 }  
}

## 5.枚举

enum EnumSingleton{  
 *INSTANCE*;  
 public void doSomeThing(){  
 }  
}

## 6.volatile的用法

**1)特性**

1.保证此变量对所有的线程的可见性。

2.禁止指令重排序优化。

**2)性能**

volatile 的读性能消耗与普通变量几乎相同，但是写操作稍慢，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执。

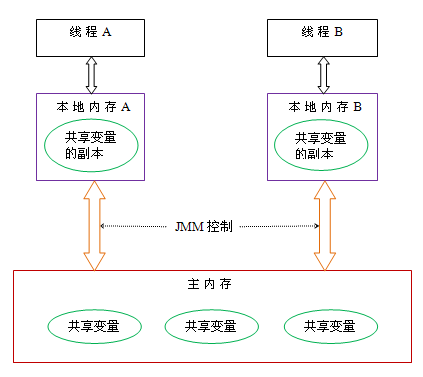
**3）用法**

1.简单赋值

2.一处赋值，多处使用

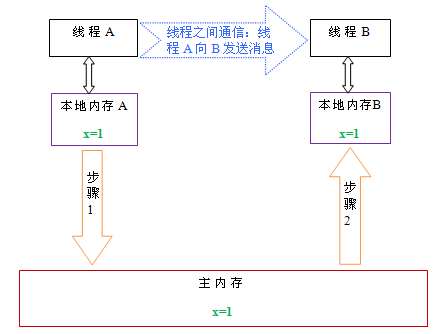
## 7.Java多线程内存模型

线程之间的共享变量存储在**主内存**（main memory）中，每个线程都有一个私有的**本地内存**（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在，它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。Java内存模型的抽象示意图如下：



线程间通信的步骤：

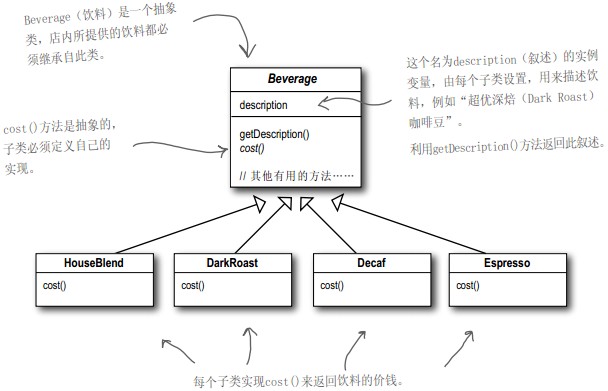
1. 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去。
2. 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。



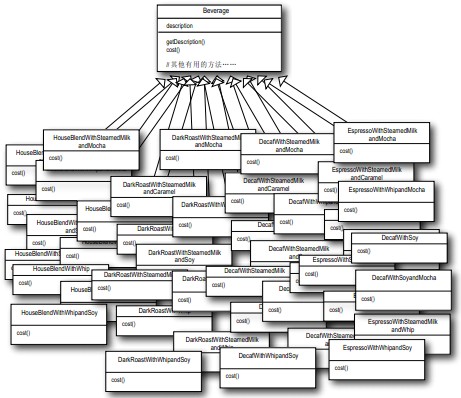
# 二.装饰者

星巴兹咖啡准备更新订单系统，以合乎他们的饮料供应需求。

他们原先的类设计为：

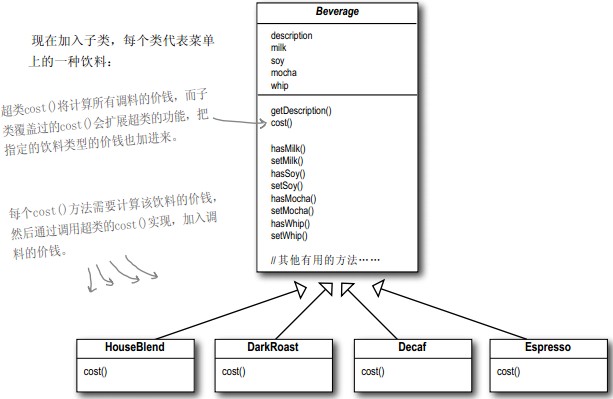


这样的订单系统没有办法考虑到咖啡调料的部分，把加入不同调料的咖啡看做不同的类会导致类爆炸（每个类的cost方法计算出咖啡加调料的价钱）：



很明显，这样的系统难以维护，一旦牛奶的价钱上扬或新增一种焦糖调料，系统将难以改变。

采用实例变量和继承的设计也许能解决一些问题：



Beverage作为一个饮料类，加上实例变量代表是否加入了饮料。

然而当用户想要双倍摩卡咖啡时，这样的系统就显得有些无所适从。

对于冰茶，饮料基类里的有些调料根本不适用，但是也一起继承了过来！

到目前为止，使用继承会造成的问题有：类爆炸，设计死板，以及基类加入的新功能并不适用于所有的子类。

所以继承并不是解决问题的方法，应当使用组合来使系统更有弹性且易于维护。

**开放-关闭原则：**

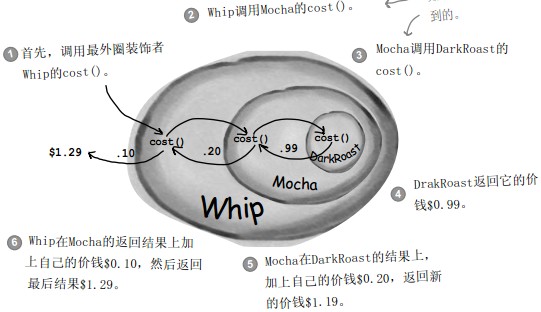
设计原则：

类应该对扩展开放，对修改关闭。

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。

这个目标需要使用装饰着模式实现：以饮料为主体，然后运行调料来“装饰”饮料。

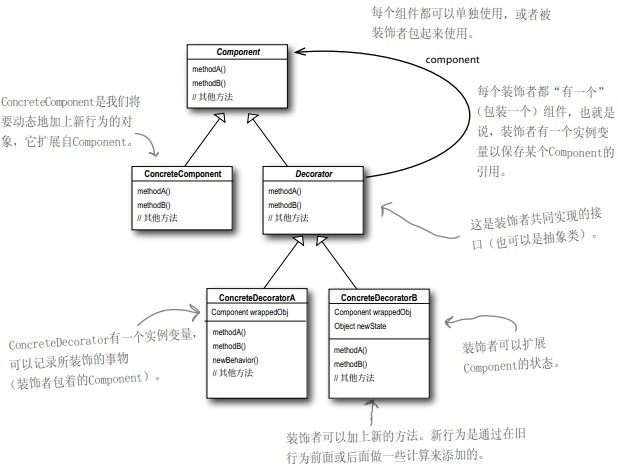
如图为一个摩卡和奶泡DarkRoast咖啡的设计图：



**定义装饰者模式：**

装饰者模式动态的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具有弹性的替代方案。

装饰者模式类图：



现在让星巴兹咖啡系统也符合装饰者类图：

具体实现:

从饮料下手，将饮料作为一个抽象类：

**[java]** [view plain](http://blog.csdn.net/lissdy/article/details/7540090) [copy](http://blog.csdn.net/lissdy/article/details/7540090)

1. **package** com.cafe;
3. **public** **abstract** **class** Beverage {
4. String description = "Unknow Beverage";
6. **public** String getDescription() {
7. **return** description;
8. }
10. **public** **abstract** **double** cost();
11. }

调料抽象类，也就是装饰者类：

**[java]** [view plain](http://blog.csdn.net/lissdy/article/details/7540090) [copy](http://blog.csdn.net/lissdy/article/details/7540090)

1. **package** com.cafe;
3. **public** **abstract** **class** CondimentDecorator **extends** Beverage{
4. **public** **abstract** String getDescription();
5. }

实现具体的饮料（浓缩咖啡和综合咖啡）：

**[java]** [view plain](http://blog.csdn.net/lissdy/article/details/7540090) [copy](http://blog.csdn.net/lissdy/article/details/7540090)

1. **package** com.cafe;
3. **public** **class** Espresso **extends** Beverage {
4. **public** Espresso() {
5. description = "Espresso";
6. }
8. **public** **double** cost() {
9. **return** 1.99;
10. }
11. }

**[java]** [view plain](http://blog.csdn.net/lissdy/article/details/7540090) [copy](http://blog.csdn.net/lissdy/article/details/7540090)

1. **package** com.cafe;
3. **public** **class** HouseBlend **extends** Beverage {
4. **public** HouseBlend() {
5. description = "HouseBlend";
6. }
8. **public** **double** cost() {
9. **return** 0.89;
10. }
11. }

实现具体装饰者类（摩卡）

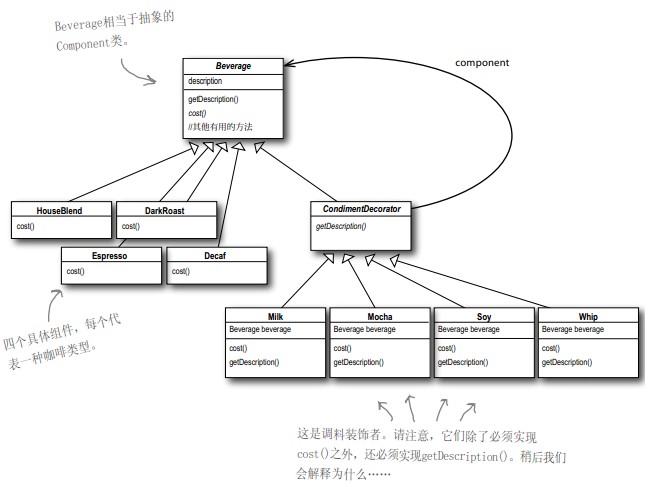
**[java]** [view plain](http://blog.csdn.net/lissdy/article/details/7540090) [copy](http://blog.csdn.net/lissdy/article/details/7540090)

1. **package** com.cafe;
3. **public** **class** Mocha **extends** CondimentDecorator{
4. Beverage beverage;
5. **public** Mocha(Beverage beverage)
6. {
7. **this**.beverage=beverage;
8. }
9. **public** String getDescription()
10. {
11. **return** beverage.getDescription()+", Mocha";
12. }
13. **public** **double** cost()
14. {
15. **return** 0.20+beverage.cost();
16. }
17. }

其他装饰者类的实现方式与摩卡类似。

[**测试**](http://lib.csdn.net/base/softwaretest)代码：

1. **package** com.cafe;
3. **public** **class** StartbuzzCoffee {
4. **public** **static** **void** main(String args[]) {
5. Beverage beverage1 = **new** Espresso();
6. System.out.println(beverage1.getDescription() + "  $"
7. + beverage1.cost());
9. // Beverage beverage2 = new Whip(new Mocha(new Soy(new HouseBlend())));
10. Beverage beverage2 = **new** HouseBlend();
11. beverage2 = **new** Soy(beverage2);
12. beverage2 = **new** Mocha(beverage2);
13. beverage2 = **new** Whip(beverage2);
15. System.out.println(beverage2.getDescription() + "  $"
16. + beverage2.cost());
17. }
19. }

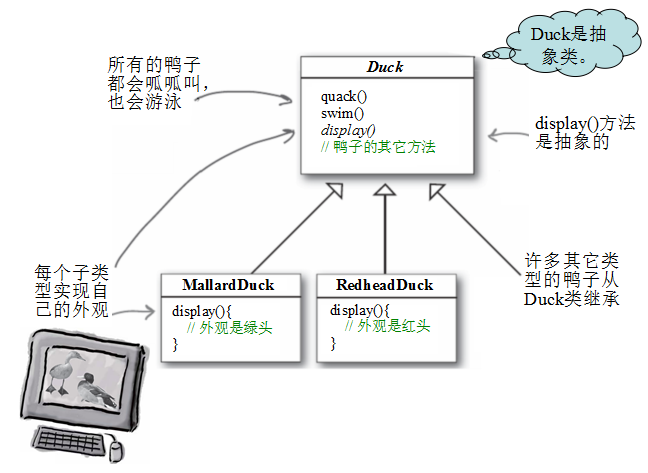


# 三．策略模式

策略模式定义了算法族，分别封装起来，让它们之间可以相互替换，此模式让算法的变化独立于使用算法的客户。

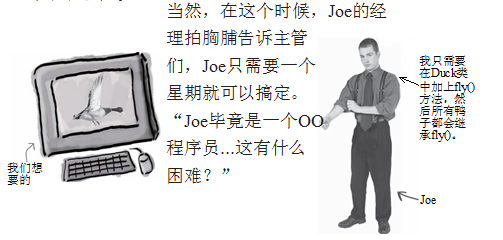
## 问题的心路历程

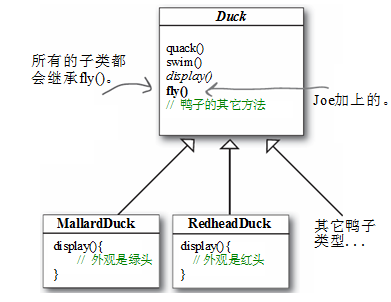
TCL公司做了一套相当成功的模拟鸭子游戏SimUDuck，游戏中出现各种鸭子，一边游戏戏水，一边呱呱叫。此系统的内部设计使用了标准的OO技术，设计了一个鸭子超类，并让各种鸭子继承此超类。

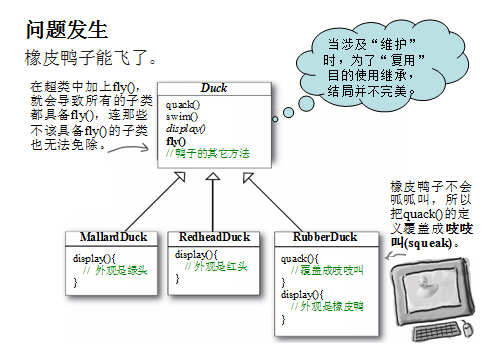


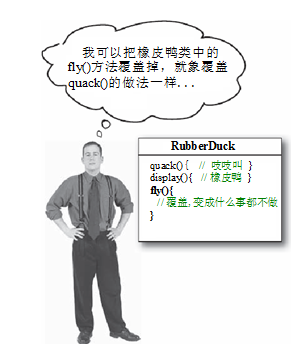
**让鸭子能飞**

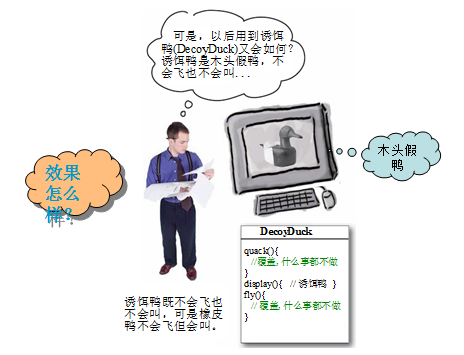
去年，公司的竞争力加剧，公司xu哥认为该是创新的时候了。xu哥认为，此模拟程序需要会飞的鸭子，将竞争者抛在后面。





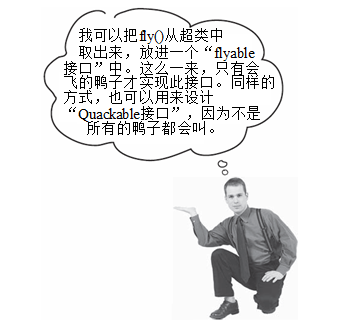


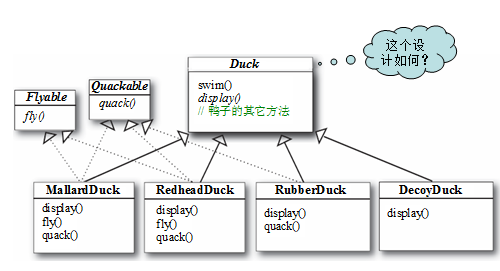


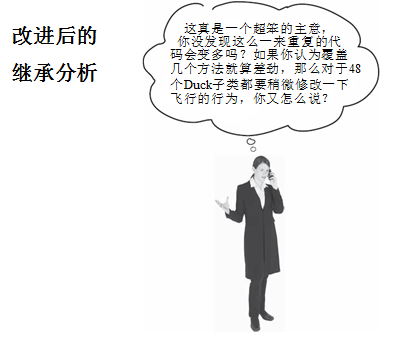


**改进继承**

Joe认识到继承可能不是一个好的解决办法，因为他刚刚拿到来自xu哥的备忘录，希望以后每两个星期更新产品（至于更新办法，他们还没想到）。Joe知道规格会常常改变，每当有新的鸭子子类出现，他就要被迫检视并可能需要覆盖fly()和quack().....这简直是无穷尽的噩梦。所以，他需要一个更清晰的方法，让某些（而不是全部）鸭子类型可飞或可叫。







其实，并非所有的鸭子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可以解决一部分的问题（不会再有会飞的橡皮鸭），但是却造成代码无法复用，这只能算是从一个噩梦跳进另一个噩梦。甚至，在会飞的鸭子中，飞行的动作可能还有多种变化......

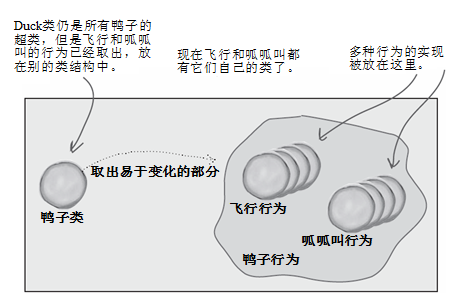
现在我们知道使用继承有一些缺失，因为改变鸭子的行为会影响所有种类的鸭子，而这并不恰当。Flyable与Quackable接口一开始似乎还挺不错，解决了问题（只有会飞的鸭子才继承Flyable），但是接口不具有实现代码，所以继承接口无法达到代码的复用。这意味着：无论何时你需要修改某个行为，你被迫得往下追踪并修改每一个有定义此行为的类，一不小心，可能造成新的错误。

## 第一个设计原则

设计原则：找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代码混在一起。这个概念很简单，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它部分。

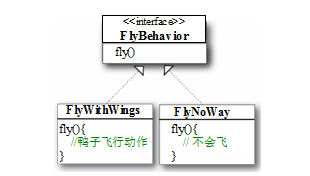
**分开变化和不变化的部分**

为了要分开变化和不变化的部分，我们准备建立两组类，一个是fly相关，一个是quack相关的，每一组类将实现各自的动作。比如说，我们可能有一个类实现“呱呱叫”，另一个类实现“叽叽叫”，另一个类实现“安静”。



**设计鸭子行为**

我们利用接口代表每个行为，比方说，IFlyBehavior与IQuackBehaivor，而行为的每个实现都必须实现这些接口之一。所以这次鸭子类不会负责实现fly与quack接口，而是由其他类专门实现IFlyBehavior与IQuackBehaivor，这就称为“行为类”。由行为类实现行为接口，而不是由Duck类实现接口。



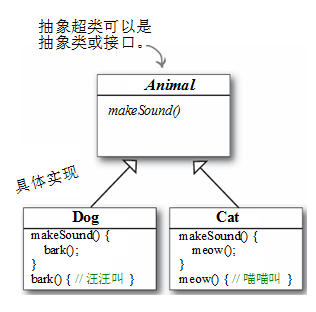
 这样的做法迥异于以往，以前的做法是：行为是继承自Duck超类的具体实现而来，或是继承某个接口并由子类自行实现而来。这两种做法都是依赖于实现，我们被实现绑的死死，没办法更改行为（除非写更多的代码）。

## 第二个设计原则

针对接口编程，而不是针对实现编程。

**关于接口编程和实现编程：**

假设有一个抽象类Animal，有两个具体的实现（Dog与Cat）继承自Animal。



针对实现编程的作法如下：

1 Dog d = new Dog();

2 d.bark();

针对接口/超类型编程作法如下：

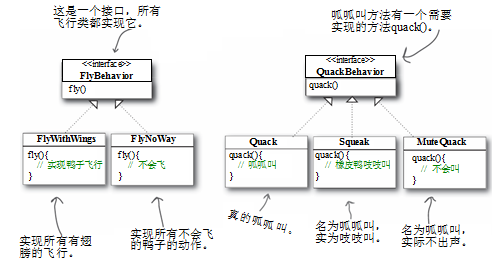
1 Animal animal = new Dog();

2 animal.makeSound();

子类型实例化的动作是“在运行时才指定具体实现的对象”

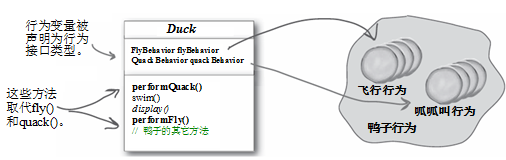
1 a = getAnimal();

2 a.makeSound();

**实现鸭子的行为**  


这样的设计，可以让飞行和呱呱叫的动作被其他的对象复用，因为这些行为已经与鸭子无关了。而我们可以新增一些行为，不会影响到既有的行为类，也不会影响有使用到飞行行为的鸭子类。

**集成鸭子的行为**  
鸭子现在会将飞行和呱呱叫的动作，委托(delegate)别人处理，而不是使用定义在自己类(或子类)内的方法。  
首先，在鸭子中加入两个实例变量， 分别为FlyBehavior与QuackBehavior，声明为接口类型(而不是具体类实现类型)，每个变量会利用多态的方式在运行时引用正确的行为类型(例如：FlyWithWings、Squeak . . . 等)。  
我们也必须将Duck类与其所有子类中的fly() 与quack( ) 移除，因为这些行为已经被搬移到FlyBehavior与QuackBehavior类中了。我们用performFly()和performQuack()取代Duck类中的fly()与quack()。



## 第三个设计原则

**“有一个”(has a)可能比“是一个”(is a)更好**

有一个关系相当有趣：每一鸭子都有一个FlyBehavior且有一个QuackBehavior，让鸭子将飞行和呱呱叫委托它们代为处理。

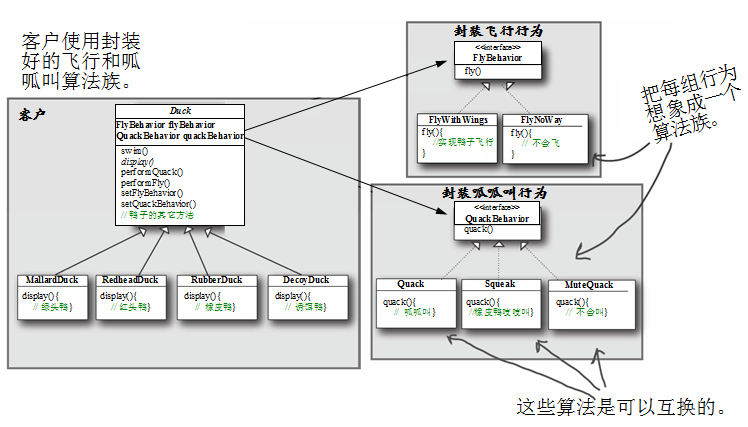
如果将两个类结合起来使用(如同本例)，这就是组合(Composition)。这种作法和继承不同的地方在于：鸭子的行为不是继承而来，而是和适当的行为对象组合而来。

多用组合，少用继承

使用组合建立系统具有很大的弹性，不仅可将算法族封装成类，更可以在运行时动态地改变行为。

## 总结

所有鸭子从Duck继承，飞行行为实现FlyBehavior接口，呱呱叫行为实现QuackBehavior接口。



**优缺点**

优点：

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转移到父类里面，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。但是，这样一来算法或行为的使用者就和算法或行为本身混在一起。决定使用哪一种算法或采取哪一种行为的逻辑就和算法或行为的逻辑混合在一起，从而不可能再独立演化。继承使得动态改变算法或行为变得不可能。

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。

缺点：

1、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。换言之，策略模式只适用于客户端知道所有的算法或行为的情况。

2、 策略模式造成很多的策略类，每个具体策略类都会产生一个新类。

结构图：

