# 策略模式（Strategy Pattern）

简单理解就是把会变化的逻辑单独抽取出来封装成类，当然需要先定义一个接口，当又有新的变化的时候只需要创建另外的一个类实现此接口即可。就比如别人常比喻的诸葛亮给了刘备三个锦囊，刘备“怎么打仗”这个逻辑是会变化的，把“怎么打仗”逻辑封装成锦囊，给刘备传入什么锦囊，则按照什么锦囊进行打仗。

有一个游戏，在游戏中会出现各种鸭子，一边游泳一边呱呱叫。

*/\*\* 鸭子 \*/***public abstract class** Duck {  
  
 */\*\* 鸭叫声 \*/* **public void** quack() {  
 System.***out***.println(**"嘎嘎叫"**);  
 }  
  
 **public void** swim() {  
 System.***out***.println(**"游泳"**);  
 }  
  
 */\*\* 显示鸭子外观 \*/* **public abstract void** display();  
}

*/\*\* 野鸭 \*/***public class** MallardDuck **extends** Duck {  
 @Override  
 **public void** display() {  
 System.***out***.println(**"显示绿头鸭"**);  
 }  
}

*/\*\* 红头鸭 \*/***public class** RedheadDuck **extends** Duck {  
 @Override  
 **public void** display() {  
 System.***out***.println(**"显示红头鸭"**);  
 }  
}

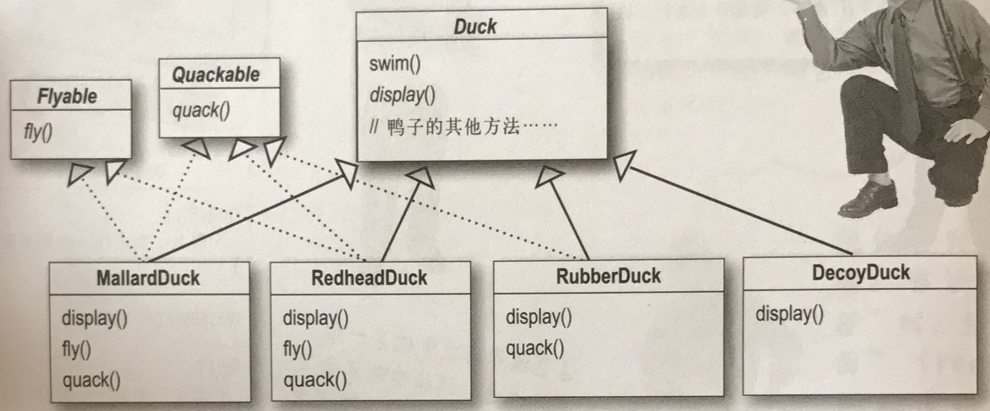
这时需要部要求鸭子会飞，开发员在Duck类上增加了fly()方法，问题来了，领导发现游戏里有很多橡皮鸭飞来飞去。修改橡皮鸭如下：

*/\*\* 橡皮鸭 \*/***public class** RubberDuck **extends** Duck {  
  
 @Override  
 **public void** quack() {  
 System.***out***.println(**"吱吱叫"**);  
 }  
  
 @Override  
 **public void** fly() {  
 *// 什么也不写，橡皮鸭不飞* }  
  
 @Override  
 **public void** display() {  
 System.***out***.println(**"显示橡皮鸭"**);  
 }  
}

这样做好像可以，但是以后如果我们加入一个木头鸭（DecoyDuck）呢？木头鸭不会叫也不会飞。又比如有1000个各类的鸭子呢？都要这样覆盖方法吗？来看看利用继承Duck提供行为，这会导致下列的缺点：

* 代码在多个子类中重复
* 很难知道所有鸭子的全部行为
* 改变会牵一发动全身，造成其他鸭子不想要的改变

了解到这些问题之后，利用接口如何，让需要飞的鸭子实现Flyable，不需要的就不实现，如下：



但是这样又导致了另外一个问题，就是叫的无法复用fly和quack行为。

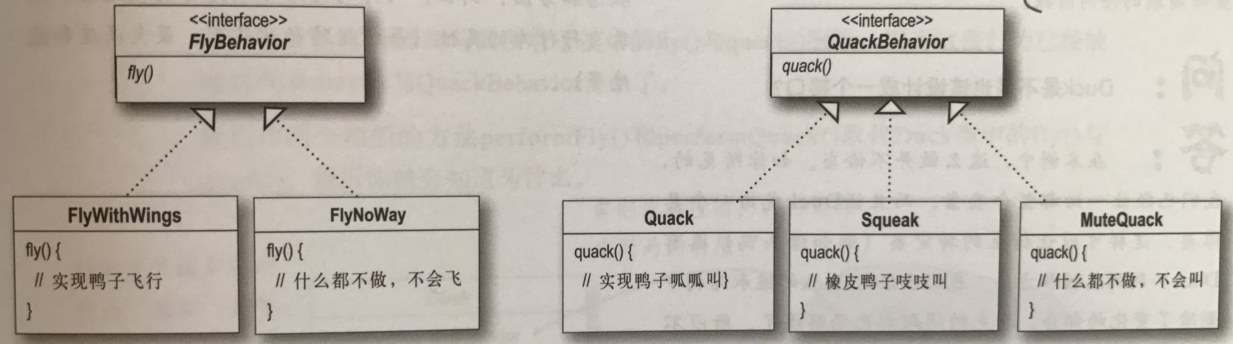
这时通过两个设计原则来解决这个问题：

* + 找出应用中可能需要变化的地方，把它们独立出来
  + 面向接口编程

鸭子的叫声和飞的行为在一些鸭子中相同需要复用，但是在另一些鸭子中又不需要这些行为，所以这些行为要独立出来，以便可以复用，如下：

*/\*\* 鸭叫的行为 \*/***public interface** QuackBehavior {  
 **void** quack();  
}

*/\*\* 飞的行为 \*/***public interface** FlyBehavior {  
 **void** fly();  
}



我们把鸭子飞行和叫的行为抽取了出来，这样，Duck鸭子类就不再需要知道这些行为的实现细节了，而且这些行为可以被其它的对象复用，而且我们新增加一些行为也不会影响到即有的行为类，也不会影响到使用飞行行为的鸭子类。

现在修改Duck类，如下：

*/\*\* 鸭子 \*/***public abstract class** Duck {  
 **protected** FlyBehavior **flyBehavior**;  
 **protected** QuackBehavior **quackBehavior**;

**public void** setFlyBehavior(FlyBehavior flyBehavior) { **this**.**flyBehavior** = flyBehavior; }  
**public void** setQuackBehavior(QuackBehavior quackBehavior) { **this**.**quackBehavior** = quackBehavior; }

**public void** swim() {  
 System.***out***.println(**"游泳"**);  
 }**public void** performQuack() { **quackBehavior**.quack(); }  
 **public void** performFly() { **flyBehavior**.fly(); }  
**public abstract void** display();  
}

在这里，飞和叫的行为，在所有的鸭子各类中，有些各类它们的行为相同，有的不相同，所以不能简单的把把这两种行为在Duck中声明抽象方法，这样没法复用，所以把这些行为抽取成独立的类，这样可以复用，而且可以扩展而不影响Duck类。

具体鸭子的实现如下：

*/\*\* 野鸭 \*/***public class** MallardDuck **extends** Duck {  
 **public** MallardDuck() {  
 **quackBehavior** = **new** Quack();  
 **flyBehavior** = **new** FlyWithWings();  
 }  
  
 @Override  
 **public void** display() {  
 System.***out***.println(**"显示绿头鸭"**);  
 }

}

运行示例：

**public static void** main(String[] args) {  
 Duck duck = **new** MallardDuck();  
 duck.performQuack();  
 duck.performFly();  
  
 duck.setFlyBehavior(**new** FlyNoWay());  
 duck.performFly();  
}

如上面代码，我们在运行时动态的改变了鸭子的行为，就比如玩飞机大战，吃到特效后可以动态地改变发射的子弹行为（发散型、激光型、跟踪型等）。

“有一个”可能比“是一个”更好，每个鸭子都有一个FlyBehavior和一个QuackBehavior，将飞和叫的行为委托给它们代为处理。当将两个类结合起来使用就是组合，这种做法和“继承”不同的地方在于，鸭子的行为不是继承的，而是和适当的行为对象“组合”来的。这是我们的第三个设计原则：多用组合，少用继承。使用组合建立系统有很大的弹性。