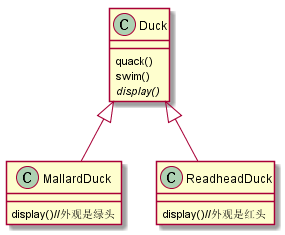
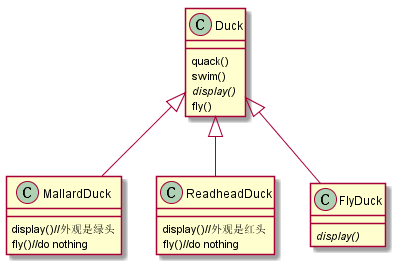
## 第一章：设计模式入门



实际案例：设计一个鸭子游戏，其中“呱呱叫”和“游泳”行为是所有鸭子的共同行为，“外观”是子类的可变行为，因此在父类中quack()和swim()方法不变，display()为抽象方法需要子类实现。

需求更改：某些鸭子会飞

修改方法：继承，在super类中加入fly()方法，然后需要在子类中对那些不能飞的鸭子复写fly方法。



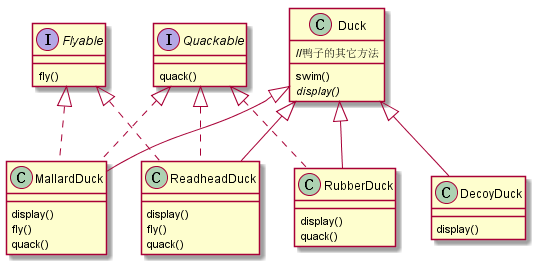
我们编写应用的时候很多时候可以想到可能有多少个子类对象，所以在有限的扩展情况下，使用继承或接口来控制少部分的变化行为并不会产生多大的影响。但这不是一个良好的编程习惯。尤其对于编写框架而言（个人以为对于应用编写，不妨这么做）。

设计模式要应对的是普遍的适应情况，因此我们编码时应该想象如果有无限个之类对象，或者有非常多的扩展时，原先的编码方式是否还能应对自如。

需求更改：需要既不会飞也不会叫的鸭子

修改方法：如同需求1中的修改方法一样，复写某些方法。但这也就同时限制了这些子类的进一步扩展，比如进一步扩展为橡皮鸭呢？此时我们需要进行大量的修改。更重要的是代码结构变得异常的混乱。

修改方法2：使用接口



这样的结果是使代码的**复用性变得非常的差**，所有的子类都需要对接口进行实现。

|  |
| --- |
| *设计原则1——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。* |

原则1告诉我们编码时尽量把会变化的部分取出并“封装”起来，好让其他部分不受影响。

上面的案例中，鸭子的行为是经常需要变化的部分，所以把这一部分取出并“封装”。那么如何封装行为呢？

|  |
| --- |
| *设计原则2——针对接口编程，而不是争对实现编程。* |

即所有的代码都是争对抽象行为的，而不是具体行为。即我不需要知道new的对象。

“针对接口编程”真正的意思是“针对超类型编程”。这一原则的关键在于利用**多态**。

简单的多态例子：假设有一个抽象类Animal，有两个具体的实现(Dog与Cat)继承Animal。做法如下：

“针对实现编程”——

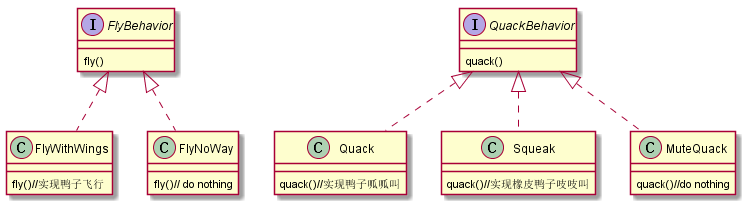
|  |
| --- |
| Dog d = new Dog();  d.bark(); |

“针对接口编程”——

|  |
| --- |
| Animal animal = new Dog();  animal.makeSound(); |

多态更重要的不需要在代码中硬编码，而是“在运行时才指定具体实现的对象”。

|  |
| --- |
| a = getAnimal();  a.makeSound(); |

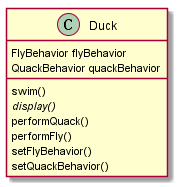


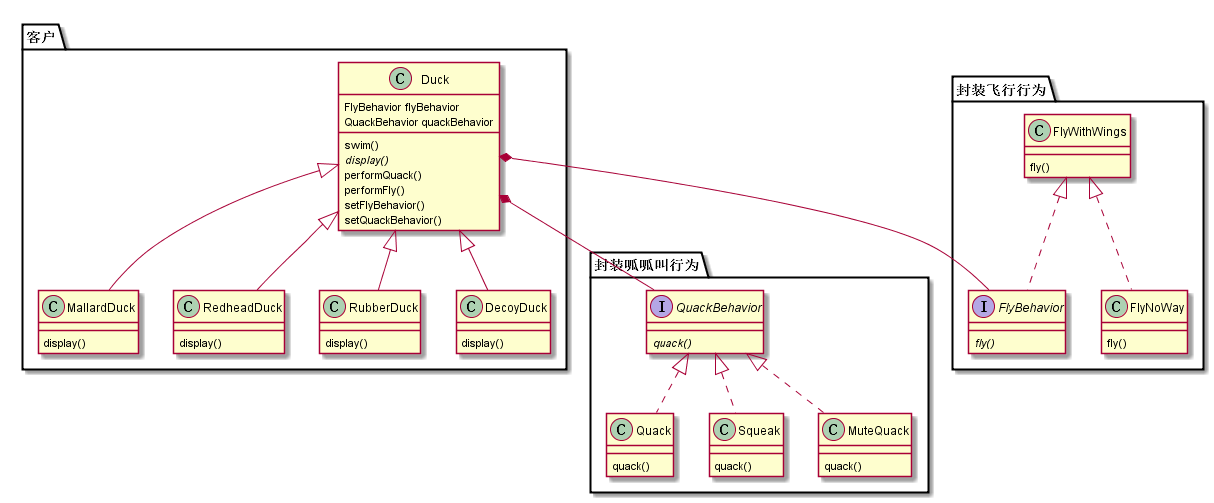
**这样的设计，可以让飞行和呱呱叫的动作被其它的对象复用，因为这些行为已经与鸭子类无关了。**

**同时还可以新增一些行为，不会影响到既有的行为类，也不会影响“使用”到飞行行为的鸭子类。**

**关键在于，鸭子现在将飞行和呱呱叫的动作“委托”别人处理，而不是使用定义在Duck类（或子类）内的呱呱叫和飞行方法。**

所以一个优化后的Duck类应该是这样的：





客户使用封装好的飞行和呱呱叫算法族。

**这种方式实际上是“策略模式”——定义算法族，并分别封装起来，让他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。**

上例使用的方法就是**组合**，这是与**继承**完全不同的实现方式。

|  |
| --- |
| *设计原则3——多用组合，少用继承* |

组合的好处时当我要实现鸭鸣器（模拟鸭叫声的装置）的时候，只需要动态的添加QuackBehavior即可。而完全不需要考虑本身是怎么实现的，对Duck类的影响等等。这样也达到了**复用**的目的。

### 总结：

OO基础：抽象，封装，多态，继承

OO原则：封装变化；多用组合，少用继承；针对接口编程，不针对实现编程。

OO模式：策略模式

### 要点：

* 良好的OO设计必须具备可复用，可扩充，可维护三个特性。
* 我们常把系统中会变化的部分抽出来封装。
* 模式让开发人员之间有共享的预研，能够最大化沟通的价值。