

1 设计模式入门

欢迎来到 设计模式世界

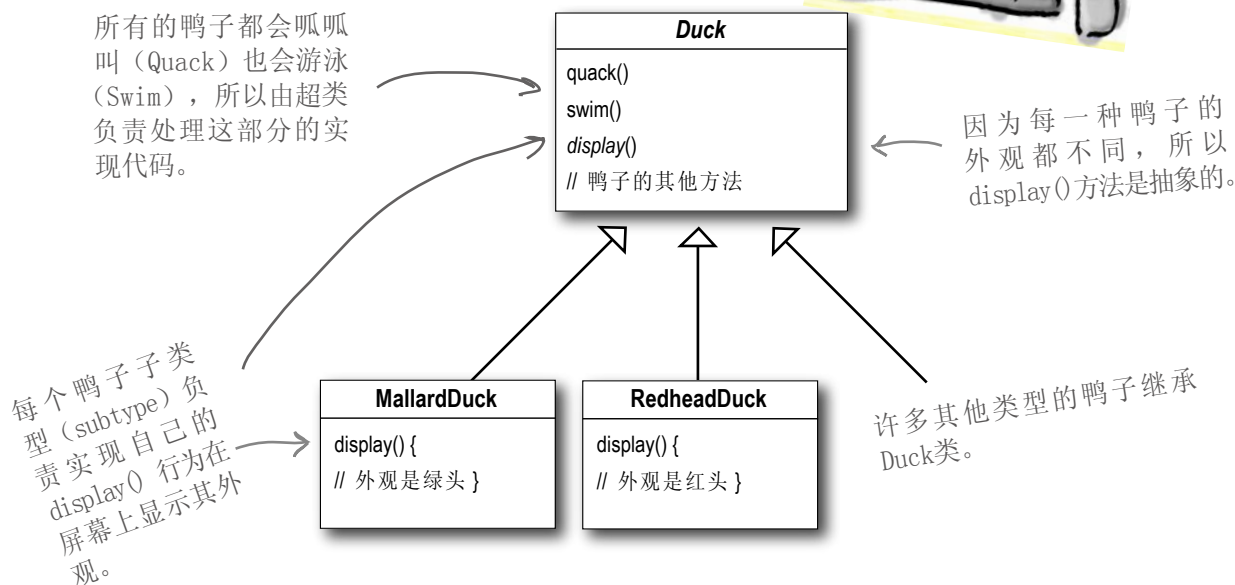
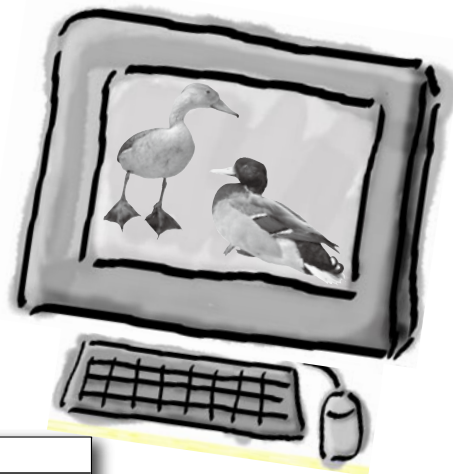


我们已经搬到对象村，刚刚开始着手设计模式……这里每个人都在使用设计模式。很快我们就会通过设计模式跻身上流社会。

有些人已经解决你的问题了。在本章，你将学到为何（以及如何）利用其他开发人员的经验与智慧。他们遭遇过相同的问题，也顺利地解决过这些问题。本章结束前，我们会看看设计模式的用途与优点，再看一些关键的OO设计原则，并通过一个实例来了解模式是如何运作。使用模式最好的方式是：“把模式装进脑子里，然后在你的设计和已有的应用中，寻找何处可以使用它们。”以往是代码复用，现在是经验复用。

先从简单的模拟鸭子应用做起

Joe上班的公司做了一套相当成功的模拟鸭子游戏：SimUDuck。游戏中会出现各种鸭子，一边游泳戏水，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（Superclass），并让各种鸭子继承此超类。



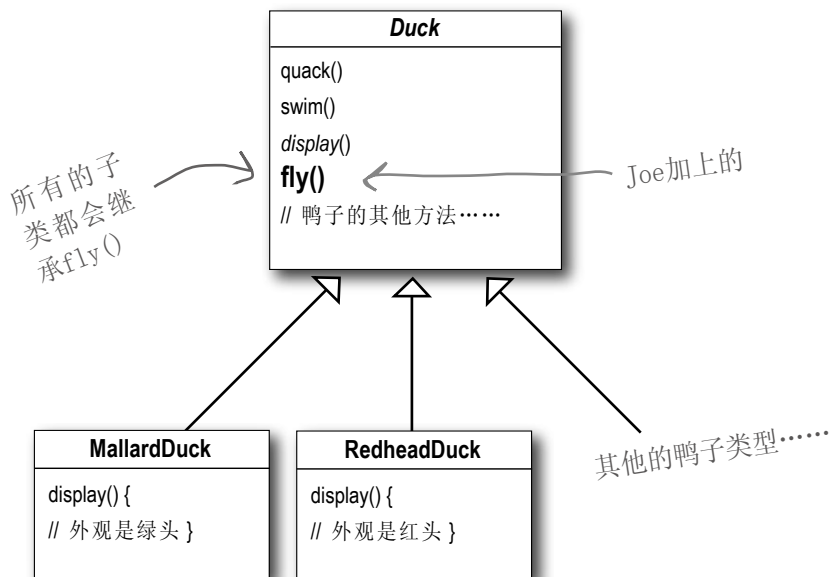
去年，公司的竞争压力加剧。在为期一周的高尔夫假期兼头脑风暴会议之后，公司主管认为该是创新的时候了，他们需要在“下周”毛伊岛股东会议上展示一些“真正”让人印象深刻的东西来振奋人心。

现在我们要得让鸭子能飞

主管们确定，此模拟程序需要会飞的鸭子来将竞争者抛在后头。当然，在这个时候，Joe的经理拍胸脯告诉主管们，Joe只需要一个星期就可以搞定。“毕竟，Joe是一个OO程序员……这有什么困难？”



我只需要在Duck类中加上
fly()方法，然后所有鸭子都会继承
fly()。这是我大显身手，展示OO才
华的时候了。



但是，可怕的问题发生了……

Joe，我正在股东会议上，刚刚看了一下展示，有很多“橡皮鸭子”在屏幕上飞来飞去，这是你在开玩笑吗？你可能要开始去逛逛Monster.com（编注：美国最大的求职网站）了……



怎么回事？

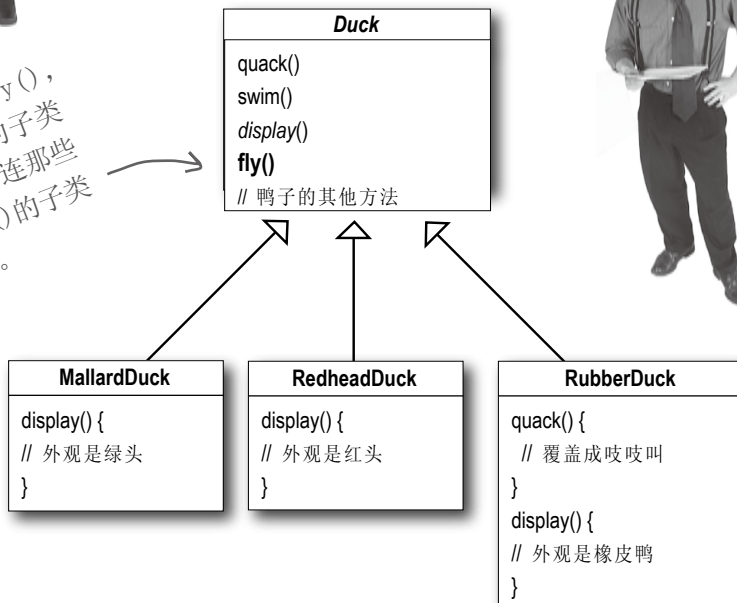
Joe忽略了一件事：并非Duck所有的子类都会飞。Joe在Duck超类中加上新的行为，会使得某些并不适合该行为的子类也具有该行为。现在可好了！SimUDuck程序中有了一个无生命的会飞的东西。

对代码所做的局部修改，影响层面可不只是局部（会飞的橡皮鸭）！

好吧！我承认设计中有一小点疏失。但是，他们怎么不干脆把这当成一种“特色”，其实还挺有趣的呀……

他体会到了一件事：当涉及“维护”时，为了“复用”（reuse）目的而使用继承，结局并不完美。

在超类中加上fly()，就会导致所有的子类都具备fly()，连那些不该具备fly()的子类也无法免除。



橡皮鸭子不会呱呱叫，所以把 quack() 的定义覆盖成“吱吱叫”（squeak）。

Joe想到继承

我可以把橡皮鸭类中的fly()方法覆盖掉，就好像覆盖quack()的做法一样……



RubberDuck
quack() { // 吱吱叫 }
display() { // 橡皮鸭 }
fly() { // 覆盖，变成什么事都不做 }

可是，如果以后我加入诱饵鸭（DecoyDuck），又会如何？诱饵鸭是木头假鸭，不会飞也不会叫……



DecoyDuck
quack() { // 覆盖，变成什么事都不做 }
display() { // 诱饵鸭 }
fly() { // 覆盖，变成什么事都不做 }

这是继承层次中的另一个类。注意，诱饵鸭既不会飞也不会叫，可是橡皮鸭不会飞但会叫。

Sharpen your pencil

利用继承来提供Duck的行为，这会导致下列哪些缺点？（多选）

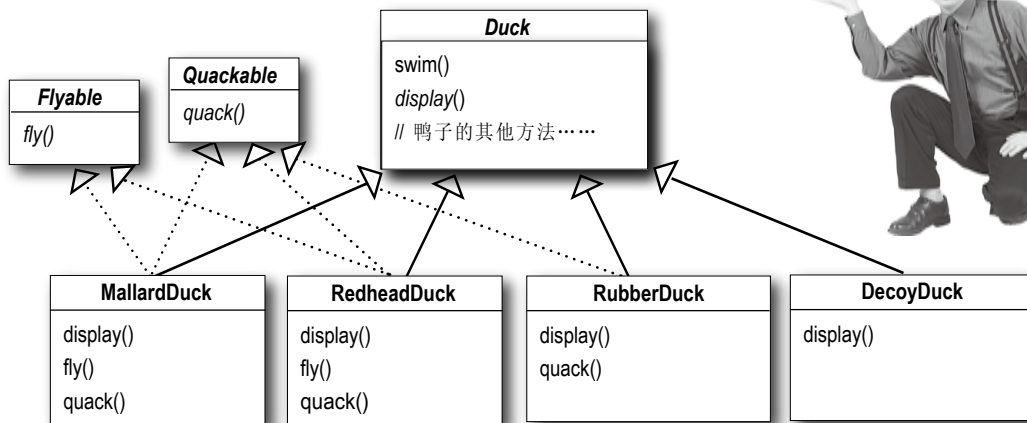
- ☐ A. 代码在多个子类中重复。
- ☐ B. 运行时的行为不容易改变。
- ☐ C. 我们不能让鸭子跳舞。
- ☐ D. 很难知道所有鸭子的全部行为。
- ☐ E. 鸭子不能同时又飞又叫。
- ☐ F. 改变会牵一发而动全身，造成其他鸭子不想要的改变。

利用接口如何？

Joe认识到继承可能不是答案，因为他刚刚拿到来自主管的备忘录，希望以后每六个月更新产品（至于更新的方法，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，他就要被迫检查并可能需要覆盖fly()和quack()……这简直是无穷无尽的噩梦。

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。

我可以把fly()从超类中取出来，放进一个“Flyable接口”中。这么一来，只有会飞的鸭子才实现此接口。同样的方式，也可以用来设计一个“Quackable接口”，因为不是所有的鸭子都会叫。



你觉得这个设计如何？

这真是一个超笨的主意，你没发现这么一来重复的代码会变多吗？如果你认为覆盖几个方法就算是差劲，那么对于48个Duck的子类都要稍微修改一下飞行的行为，你又怎么说？！



如果你是Joe，你要怎么办？

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（不会再有会飞的橡皮鸭），但是却造成代码无法复用，这只能算是从一个恶梦跳进另一个恶梦。甚至，在会飞的鸭子中，飞行的动作可能还有多种变化……

此时，你可能正期盼着设计模式能骑着白马来解救你离开苦难的一天。但是，如果直接告诉你答案，这有什么乐趣？我们会用老方法找出一个解决之道：“采用良好的OO软件设计原则”。

如果能有一种建立软件的方法，好让我们可以用一种对既有的代码影响最小的方式来修改软件该有多好。我们就可以花较少时间重做代码，而多让程序去做更酷的事……



软件开发的一个不变真理

好吧！在软件开发上，有什么是你深信不疑的？

不管你在何处工作，构建些什么，用何种编程语言，在软件开发上，一直伴随你的那个不变真理是什么？

CHANGE

（用镜子来看答案）

不管当初软件设计得多好，一段时间之后，总是需要成长与改变，否则软件就会“死亡”。



驱动改变的因素很多。找出你的应用中需要改变代码的原因，一一列出来。（我们写下了一些我们的原因，给你起个头。）

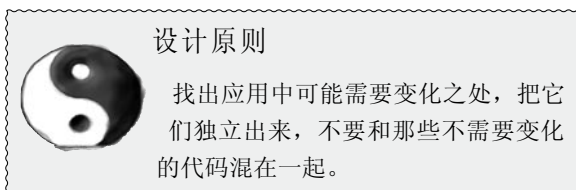
我们的顾客或用户需要别的东西，或者想要新功能。

我的公司决定采用别的数据库产品，又从另一家厂商买了数据，这造成数据格式不兼容。唉！

把问题归零……

现在我们知道使用继承并不能很好地解决问题，因为鸭子的行为在子类里不断地改变，并且让所有的子类都有这些行为是不恰当的。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Java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，可能会造成新的错误！

幸运的是，有一个设计原则，恰好适用于此状况。



这是我们的第一个设计原则，以后还有更多原则会陆续在本书中出现。

换句话说，如果每次新的需求一来，都会使某方面的代码发生变化，那么你就可以确定，这部分的代码需要被抽出来，和其他稳定的代码有所区分。

下面是这个原则的另一种思考方式：“把会变化的部分取出并封装起来，以便以后可以轻易地改动或扩充此部分，而不影响不需要变化的其他部分”。

这样的概念很简单，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”。

好，该是把鸭子的行为从Duck类中取出的时候了！

把会变化的部分取出并“封装”起来，好让其他部分不会受到影响。

结果如何？代码变化引起的不经意后果变少，系统变得更有弹性。

分开变化和不会变化的部分

从哪里开始呢？就我们目前所知，除了fly()和quack()的问题之外，Duck类还算一切正常，似乎没有特别需要经常变化或修改的地方。所以，除了某些小改变之外，我们不打算对Duck类做太多处理。

现在，为了要分开“变化和不会变化的部分”，我们准备建立两组类（完全远离Duck类），一个是“fly”相关的，一个是“quack”相关的，每一组类将实现各自的动作。比方说，我们可能有一个类实现“呱呱叫”，另一个类实现“吱吱叫”，还有一个类实现“安静”。

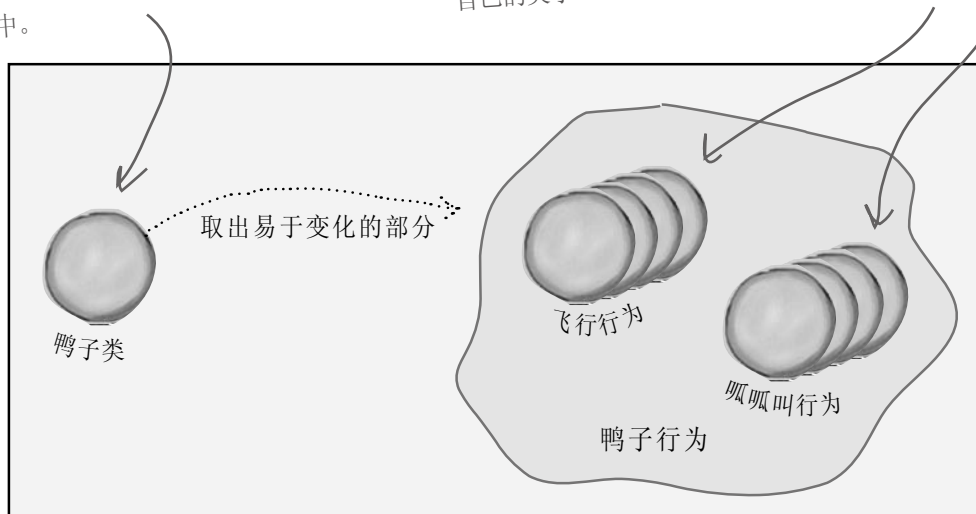
我们知道Duck类内的fly()和quack()会随着鸭子的不同而改变。

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，建立一组新类来代表每个行为。

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取出，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。



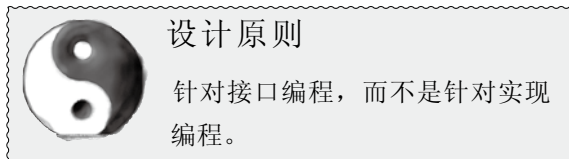


设计鸭子的行为

如何设计那组实现飞行和呱呱叫的行为的类呢？

我们希望一切能有弹性，毕竟，正是因为一开始鸭子行为没有弹性，才让我们走上现在这条路。我们还能能够“指定”行为到鸭子的实例。比方说，我们想要产生一个新的绿头鸭实例，并指定特定“类型”的飞行行为给它。干脆顺便让鸭子的行为可以动态地改变好了。换句话说，我们应该在鸭子类中包含设定行为的方法，这样就可以在“运行时”动态地“改变”绿头鸭的飞行行为。

有了这些目标要实现，接着看看第二个设计原则：



我们利用接口代表每个行为，比方说，FlyBehavior与QuackBehavior，而行为的每个实现都将实现其中的一个接口。

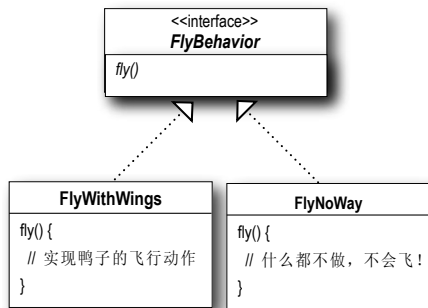
所以这次鸭子类不会负责实现Flying与Quacking接口，反而是由我们制造一组其他类专门实现FlyBehavior与QuackBehavior，这就称为“行为”类。由行为类而不是Duck类来实现行为接口。

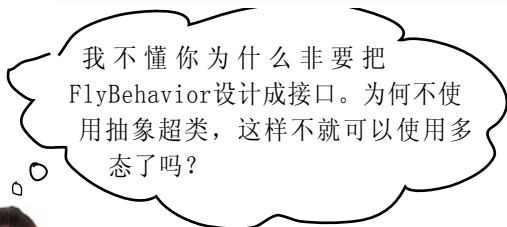
这样的做法迥异于以往，以前的做法是：行为来自Duck超类的具体实现，或是继承某个接口并由子类自行实现而来。这两种做法都是依赖于“实现”，我们被实现绑得死死的，没办法更改行为（除非写更多代码）。

在我们的新设计中，鸭子的子类将使用接口（FlyBehavior与QuackBehavior）所表示的行为，所以实际的“实现”不会被绑定在鸭子的子类中。（换句话说，特定的具体行为编写在实现了FlyBehavior与QuackBehavior的类中）。

从现在开始，鸭子的行为将被放在分开的类中，此类专门提供某行为接口的实现。

这样，鸭子类就不再需要知道行为的实现细节。





我不懂你为什么非要把
FlyBehavior设计成接口。为何不使
用抽象超类，这样不就可以使用多
态了吗？

“针对接口编程”真正的意思是“针对超类型（supertype）编程”。

这里所谓的“接口”有多个含义，接口是一个“概念”，也是一种Java的interface构造。你可以在不涉及Java interface的情况下，“针对接口编程”，关键就在多态。利用多态，程序可以针对超类型编程，执行时会根据实际状况执行到真正的行为，不会被绑死在超类型的行为上。“针对超类型编程”这句话，可以更明确地说成“变量的声明类型应该是超类型，通常是一个抽象类或者是一个接口，如此，只要是具体实现此超类型的类所产生的对象，都可以指定给这个变量。这也意味着，声明类时不用理会以后执行时的真正对象类型！”

这可能不是你第一次听到，但是请务必注意我们说的是同一件事。看看下面这个简单的多态例子：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();
```

声明变量“d”为Dog类型（是Animal的具体实现），会造成我们必须针对具体实现编码。

但是，“针对接口/超类型编程”做法会如下：

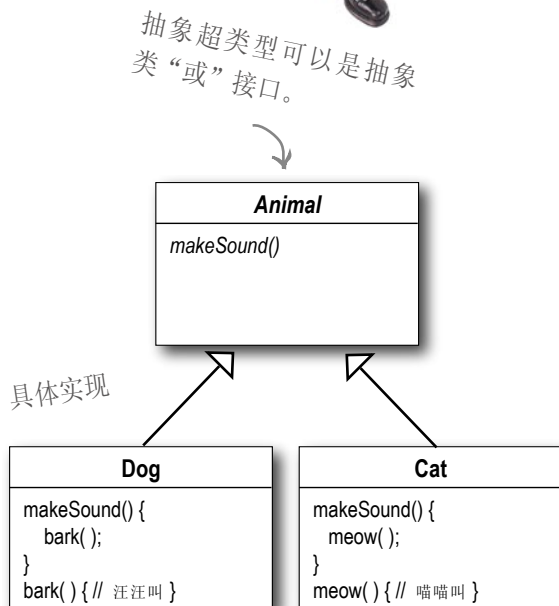
```
Animal animal = new Dog();
animal.makeSound();
```

我们知道该对象是狗，但是我们现在利用animal进行多态的调用。

更棒的是，子类实例化的动作不再需要在代码中硬编码，例如new Dog()，而是“在运行时才指定具体实现的对象”。

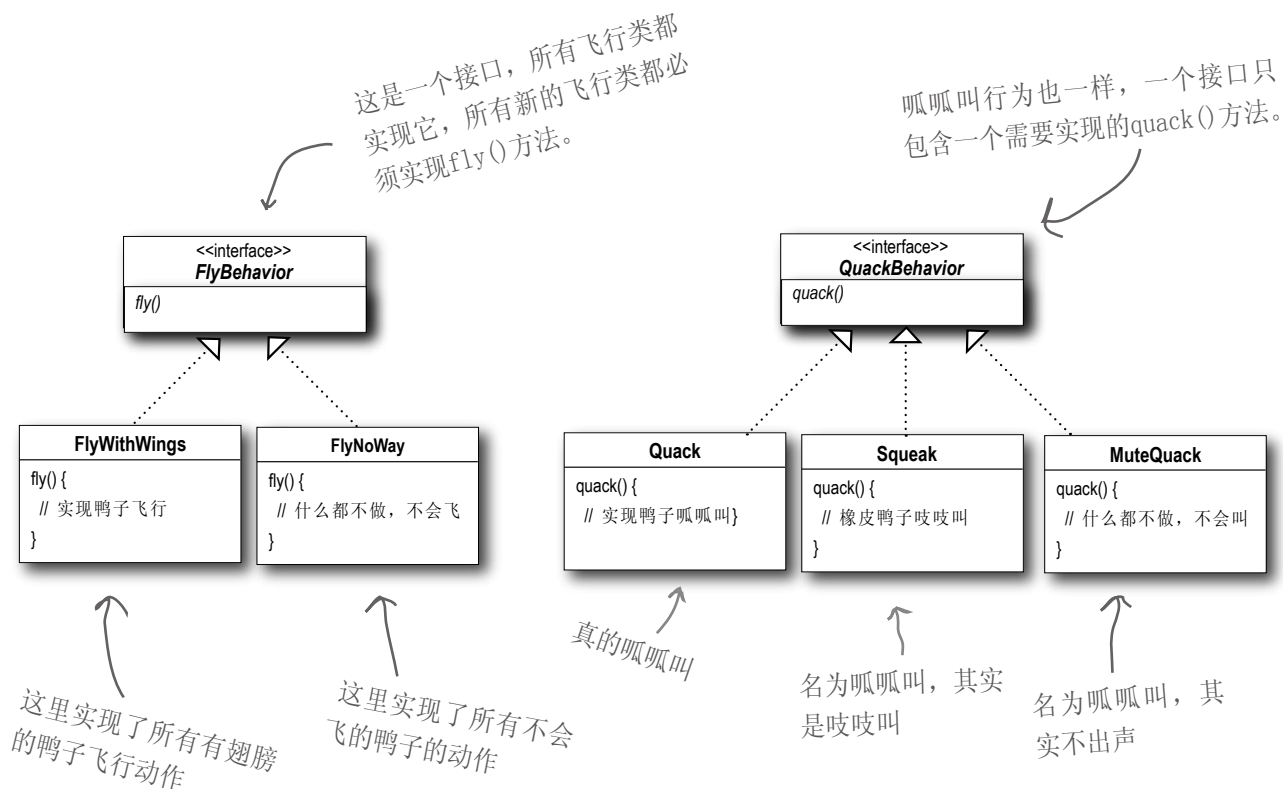
```
a = getAnimal();
a.makeSound();
```

我们不知道实际的子类型是“什么”……我们只关心它知道如何正确地进行makeSound()的动作就够了。



实现鸭子的行为

在此，我们有两个接口，FlyBehavior和QuackBehavior，还有它们对应的类，负责实现具体的行为：



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”到飞行行为的鸭子类。

这么一来，有了继承的“复用”好处，却没有继承所带来的包袱。

Database of Dumb Questions

问： 我是不是一定要先把系统做出来，再看看有哪些地方需要变化，然后才回头去把这些地方分离&封装？

答： 不尽然。通常在你设计系统时，预先考虑到有哪些地方未来可能需要变化，于是提前在代码中加入这些弹性。你会发现，原则与模式可以应用在软件开发生命周期的任何阶段。

问： Duck是不是也该设计成一个接口？

答： 在本例中，这么做并不恰当。如你所见的，我们已经让一切都整合妥当，而且让Duck成为一个具体类，这样可以让衍生的特定类（例如绿头鸭）具有Duck共同的属性和方法。我们已经从Duck的继承结构中删除了变化的部分，原先的问题都已经解决了，所以不需要把Duck设计成接口。

问： 用一个类代表一个行为，感觉似乎有点奇怪。类不是应该代表某种“东西”吗？类不是应该同时具备状态“与”行为吗？

答： 在OO系统中，是的，类代表的东西一般都是既有状态（实例变量）又有方法。只是在本例中，碰巧“东西”是个行为。但是即使是行为，也仍然可以有状态和方法，例如，飞行的行为可以具有实例变量，记录飞行行为的属性（每秒翅膀拍动几下、最大高度和速度等）。

Sharpen your pencil

- 1 使用我们的新设计，如果你要加上一个火箭动力的飞行动作到SimUDuck 系统中，你该怎么做？
- 2 除了鸭子之外，你能够想出有什么类会需要用到呱呱叫的行为？

答案：
1) 建立一个FlyRocketPowered类，实现FlyBehavior接口。
2) 例如：鸭鸣器（DuckCall）（一种会产生鸭叫声的装置）。

整合鸭子的行为

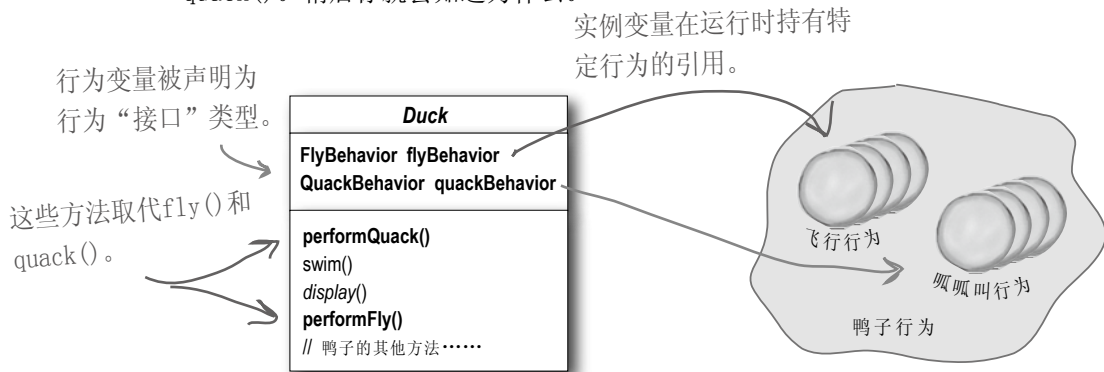
关键在于，鸭子现在会将飞行和呱呱叫的动作“委托”（delegate）别人处理，而不是使用定义在Duck类（或子类）内的呱呱叫和飞行方法。

做法是这样的：

- 1 首先，在Duck类中“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），每个鸭子对象都会动态地设置这些变量以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()删除，因为这些行为已经被搬到FlyBehavior与QuackBehavior类中了。

我们用两个相似的方法performFly()和performQuack()取代Duck类中的fly()与quack()。稍后你就会知道为什么。



- 2 现在，我们来实现performQuack()：

```
public class Duck {
    QuackBehavior quackBehavior;
    // 还有更多

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

每只鸭子都会引用实现QuackBehavior接口的对象。

鸭子对象不亲自处理呱呱叫行为，而是委托给quackBehavior引用的对象。

很容易，是吧？想进行呱呱叫的动作，Duck对象只要叫quackBehavior对象去呱呱叫就可以了。在这部分的代码中，我们不在乎quackBehavior接口的对象到底是什么，我们只关心该对象知道如何进行呱呱叫就够了。



更多的整合……

- ❸ 好吧！现在来关心“如何设定flyBehavior与quackBehavior的实例变量”。看看MallardDuck类：

```
public class MallardDuck extends Duck {  
  
    public MallardDuck() {  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
  
}
```

别忘了，因为MallardDuck继承Duck类，所以具有flyBehavior 与 quackBehavior 实例变量。

绿头鸭使用Quack类处理呱呱叫，所以当performQuack()被调用时，叫的职责被委托给Quack对象，而我们得到了真正的呱呱叫。
使用FlyWithWings作为其FlyBehavior类型。

```
    public void display() {  
        System.out.println("I'm a real Mallard duck");  
    }  
}
```

所以，绿头鸭会真的“呱呱叫”，而不是“吱吱叫”，或“叫不出声”。这是怎么办到的？当MallardDuck实例化时，它的构造器会把继承来的quackBehavior实例变量初始化成Quack类型的新实例（Quack是QuackBehavior的具体实现类）。

同样的处理方式也可以用在飞行行为上：MallardDuck的构造器将flyBehavior实例变量初始化成FlyWithWings类型的实例（FlyWithWings是FlyBehavior的具体实现类）。



等一下，你不是说过我们将不对具体实现编程吗？但是我们在那个构造器里做什么呢？我们正在制造一个具体的Quack实现类的实例！

被你逮到了，我们的确是这么做的……“只是暂时”。

在本书的后续内容中，我们的工具箱中会有更多的模式可用，到时候就可以修正这一点了。

仍请注意，虽然我们把行为设定成具体的类（通过实例化类似Quack或FlyWithWings的行为类，并把它指定到行为引用变量中），但是还是可以在运行时“轻易地”改变它。

所以，目前的做法还是很有弹性的，只是初始化实例变量的做法不够弹性罢了。但是想一想，因为quackBehavior的实例变量是一个接口类型，我们能够在运行时，通过多态的魔力动态地给它指定不同的QuickBehavior实现类。

花一点儿时间想一想，你如何实现一个其行为可以在运行时改变的鸭子。（几页以后，你就会看到做这件事的代码。）

测试Duck的代码

- 1 输入并编译下面的Duck类（Duck.java）以及两页前的MallardDuck类（MallardDuck.java）。

```
public abstract class Duck {

    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;
    public Duck() {

    }

    public abstract void display();

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

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

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }
}
```

为行为接口类型声明两个引用变量，所有鸭子子类（在同一个package中）都继承它们。

委托给行为类

- 2 输入并编译FlyBehavior接口（FlyBehavior.java）与两个行为实现类（FlyWithWings.java与FlyNoWay.java）。

```
public interface FlyBehavior {
    public void fly();
}

public class FlyWithWings implements FlyBehavior {
    public void fly() {
        System.out.println("I'm flying!!");
    }
}

public class FlyNoWay implements FlyBehavior {
    public void fly() {
        System.out.println("I can't fly");
    }
}
```

所有飞行行为类必须实现的接口。

这是飞行行为的实现，给“真会”飞的鸭子用……

这是飞行行为的实现，给“不会”飞的鸭子用（包括橡皮鸭和诱饵鸭）。

继续测试Duck的代码

- ❸ 输入并编译QuackBehavior接口（QuackBehavior.java）及其三个实现类（Quack.java、MuteQuack.java、Squeak.java）。

```
public interface QuackBehavior {
    public void quack();
}

public class Quack implements QuackBehavior {
    public void quack() {
        System.out.println("Quack");
    }
}

public class MuteQuack implements QuackBehavior {
    public void quack() {
        System.out.println("<< Silence >>");
    }
}

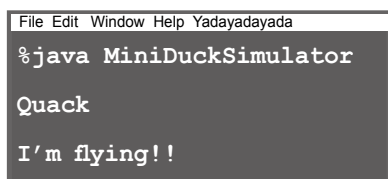
public class Squeak implements QuackBehavior {
    public void quack() {
        System.out.println("Squeak");
    }
}
```

- ❹ 输入并编译测试类（MiniDuckSimulator.java）

```
public class MiniDuckSimulator {
    public static void main(String[] args) {
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();
    }
}
```

这会调用MallardDuck继承来的performQuack()方法，进而委托给该对象的QuackBehavior对象处理（也就是说，调用继承来的quackBehavior引用对象的quack()）。至于performFly()，也是一样的道理。

- ❺ 运行代码！



```
File Edit Window Help Yadayadayada
%java MiniDuckSimulator
Quack
I'm flying!!
```

动态设定行为

在鸭子里建立了一堆动态的功能没有用到，就太可惜了！假设我们想在鸭子的子类中通过“设定方法（setter method）”来设定鸭子的行为，而不是在鸭子的构造器内实例化。

- 1 在Duck类中，加入两个新方法：

```
public void setFlyBehavior(FlyBehavior fb) {  
    flyBehavior = fb;  
}  
  
public void setQuackBehavior(QuackBehavior qb) {  
    quackBehavior = qb;  
}
```

Duck
FlyBehavior flyBehavior; QuackBehavior quackBehavior;
swim() display() performQuack() performFly() setFlyBehavior() setQuackBehavior() // 鸭子的其他方法

从此以后，我们可以“随时”调用这两个方法改变鸭子的行为。

- 2 制造一个新的鸭子类型：模型鸭（ModelDuck.java）

```
public class ModelDuck extends Duck {  
    public ModelDuck() {  
        flyBehavior = new FlyNoWay();  
        quackBehavior = new Quack();  
    }  
  
    public void display() {  
        System.out.println("I'm a model duck");  
    }  
}
```

一开始，我们的模型鸭是不会飞的。

- 3 建立一个新的FlyBehavior 类型
(FlyRocketPowered.java)

```
public class FlyRocketPowered implements FlyBehavior {  
    public void fly() {  
        System.out.println("I'm flying with a rocket!");  
    }  
}
```

没关系，我们建立一个利用火箭动力的飞行行为。



- 4 改变测试类 (MiniDuckSimulator.java)，加上模型鸭，并使模型鸭具有火箭动力。

```
public class MiniDuckSimulator {
    public static void main(String[] args) {
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();
```

```
        Duck model = new ModelDuck();
        model.performFly();
        model.setFlyBehavior(new FlyRocketPowered());
        model.performFly();
    }
}
```

如果成功了，就意味着模型鸭可以动态地改变它的飞行行为。如果把行为的实现绑定在鸭子类中，可就无法做到这样了。

- 5 运行！

```
File Edit Window Help Yabadabadoo
%java MiniDuckSimulator
Quack
I'm flying!!
I can't fly
I'm flying with a rocket!
```

改变前



第一次调用performFly() 会被委托给 flyBehavior对象（也就是FlyNoWay实例），该对象是在模型鸭构造器中设置的。

这会调用继承来的setter方法，把火箭动力飞行的行为设定到模型鸭中。哇！模型鸭突然具有了火箭动力飞行能力！



改变后

在运行时想改变鸭子的行为，只需调用鸭子的 setter方法就可以。