1　介绍设计模式

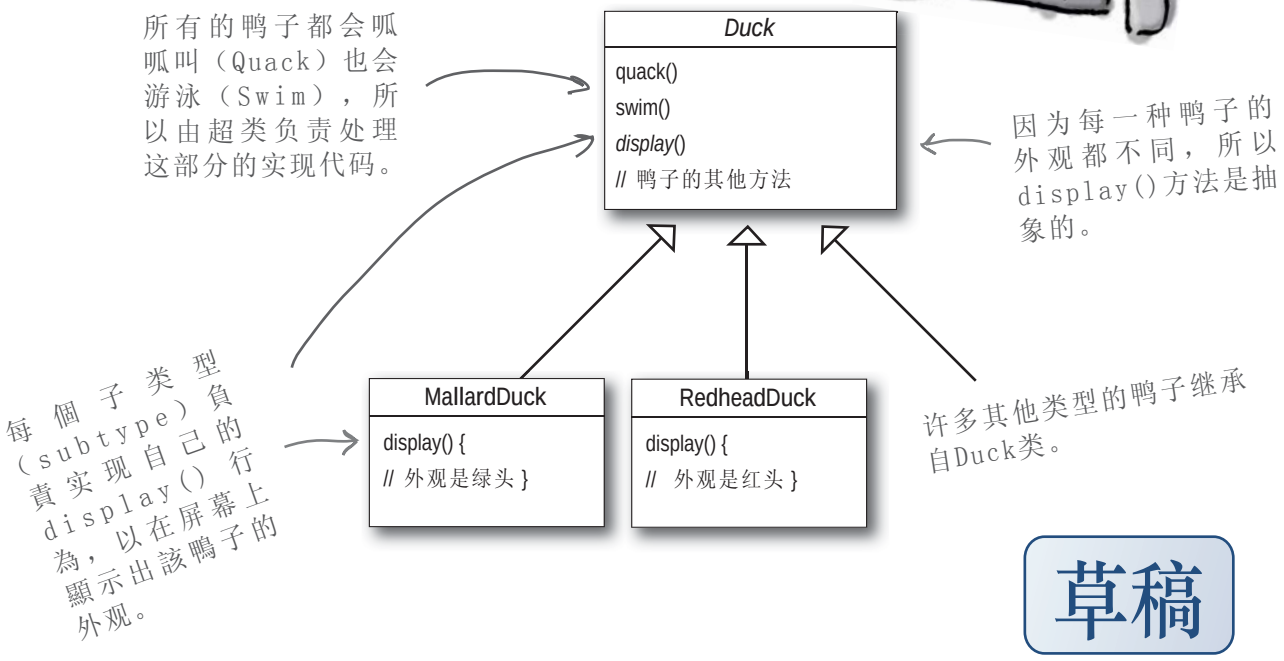
欢迎来到设计模式

草稿

有些人已经解决你的问题了。在本章，你将学习到为何（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，我们会看到设计模式的使用与优点，看看某些关键的00 设计原则，并透过一个范例来了解模式如何运作。使用模式最好的方式是：「把模式装进脑子中，然后在你的设计和已有的应用中，寻找何处可以使用这些模式。」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00　　技术，设计了一个鸭子超类（Superclass），并让各种鸭子继承自此超类。



草稿

去年，公司的竞争压力加剧。在一个星期长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」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。

草稿

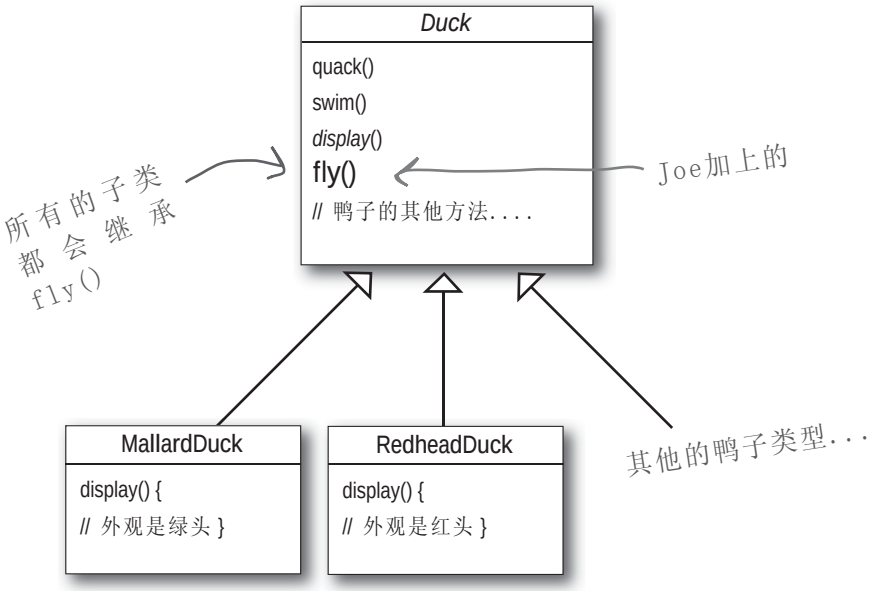
现在我们得让鸭子能飞

主管认为，此模拟程序需要会飞的鸭子，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？」

我們想要的

Joe

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。



草稿

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，有一只「橡皮鸭子」在屏幕上飞来飞去，这是你开的玩笑吗？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。当Joe在Duck超类中加上新的行为，这会使得某些子类也具有这个不恰当的行为。现在可好了！SimUDuck程序中有一个会飞的非动物。

对代码所做的局部修改，影响层面可能不只局部（会飞的橡皮鸭）！

好吧！我承认设计中有一点小疏失。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..

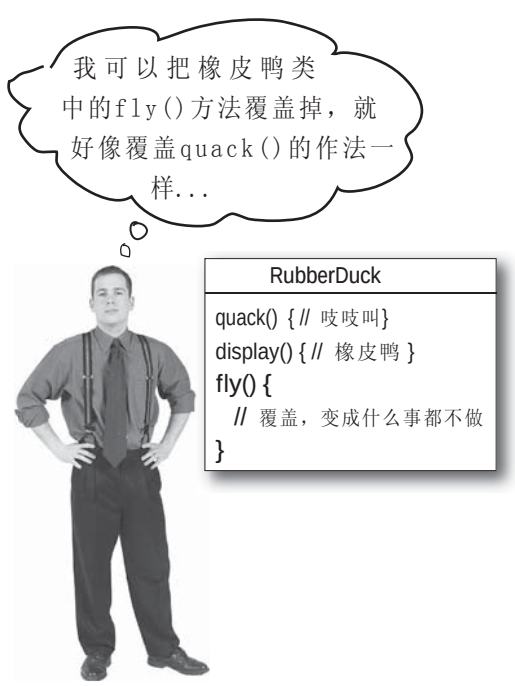
他体会到了一件事：当涉及「维护」时，为了「复用」（reuse）目的而使用继承，结局并不完美。

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）。

草稿

Joe想到继承



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。

削尖你的鉛筆

利用繼承提供鴨子行為，會導致下列哪些缺點？（多選）

- ❑ 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. 难以得知所有鸭子的全部行为。
- ❑ E. 鸭子不能同时又飞又叫。
- ❑ F. 改变会牵一发动全身，造成其他鸭子不想要的改变。

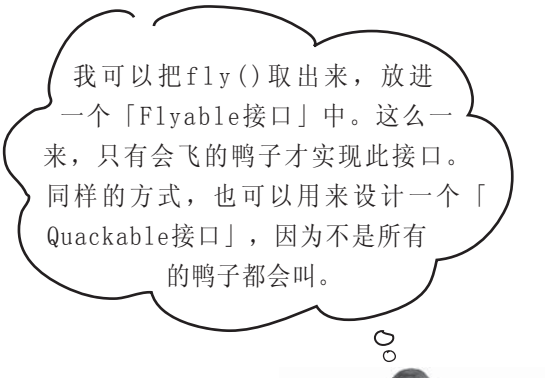
草稿

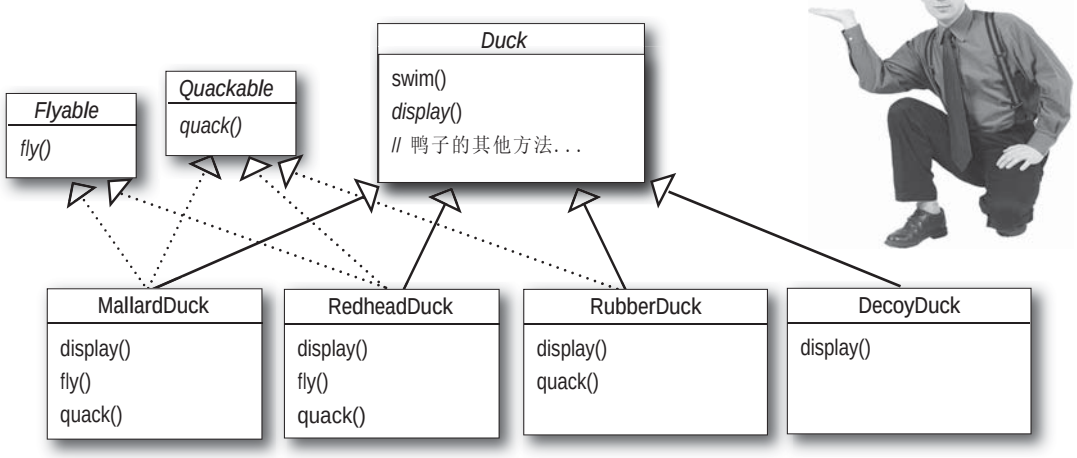
繼承並非答案

利用接口如何？

Joe认识到继承可能不是一个好的解决方法，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，他就要被迫检视并可能需要覆盖fly()　和quark()... 这简直是无穷尽的恶梦。

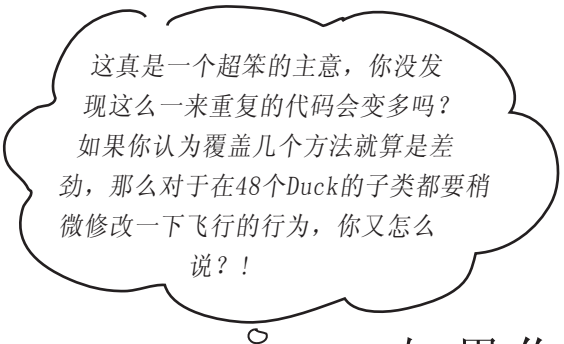
所以，他需要一个更清晰的方法，让「某些」（而不是全部）鸭子类型可飞或可叫。





你觉得这个设计如何？

草稿



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」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」。

草稿

不变的是变化

软件开发的一个不变真理

好吧！在软件开发上，有什么是你可以深信不疑的？

不管你在何处工作，建造些什么，用何种程序语言，在软件开发上，有没有一个不变的真理？

ƎGИAHƆ

（用镜子来看答案）

不管当初软件设计得多好，一阵子之后，总是需要成长与改变，
否则软件就会「死亡」。

驱使改变的因素很多。找出你的应用中需要改变代码的地方，一一列出来。（我们先起个头，好让你有个方向。）

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，也从另一家厂商买了不同的数据，
这造成数据格式不兼容。唉！

草稿

把问题归零

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Java 的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。

幸运地，有一个设计原则，正适用于此状况。

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，都会变化到某方面的代码，那么你就可以确定，这部分的代码需要被抽出来，和其他闻风不动的代码有所区隔。

下面是这个原则的另一个思考方式：「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？代码变化之后，出其不意的部分变得很少，系统变得更有弹性。

草稿

抽出變動的部分

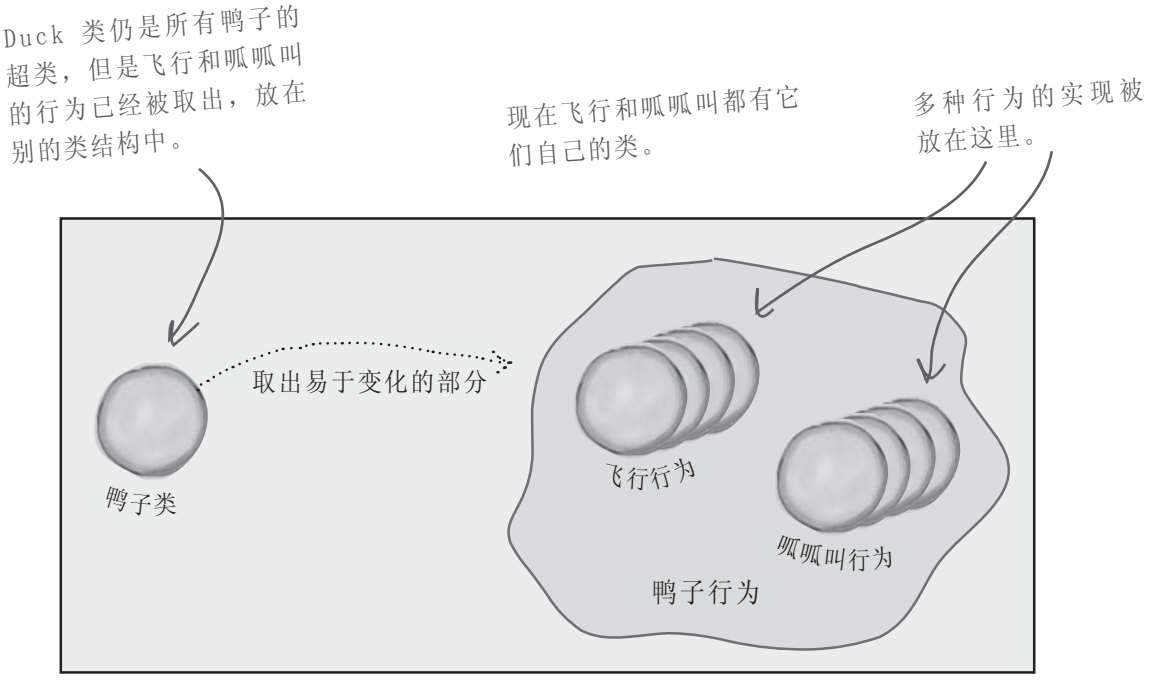
分开变化和不会变化的部分

如何开始？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」，另一个类实现「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，我们将把它们自Duck类中取出，建立一组新类代表每个行为。



设计鸭子的行为

如何设计类实现飞行和呱呱叫的行为？

我们希望一切能有弹性，毕竟，正是因为一开始的鸭子行为没有弹性，才让我们走上现在这条路。我们还想能够「指定」行为到鸭子的实例，比方说，想要产生绿头鸭实例，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，就可以在「运行时」动态地「改变」绿头鸭的飞行行为。

有了这些目标要达成，接着看看第二个设计原则：

设计原则

针对接口编程，而不是针对实现编程。

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必须实现这些接口之一。

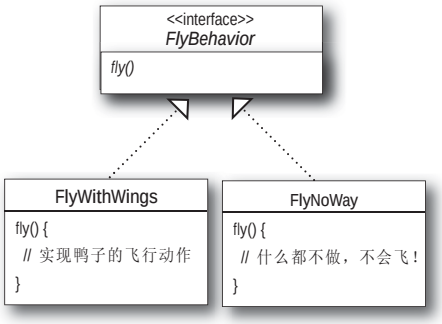
所以这次鸭子类不会负责实现Flying与Quacking接口，反而是由其他类专门实现FlyBehavior 与QuackBehavior，这就称为「行为」类。由行为类实现行为接口，而不是由Duck类实现行为接口。

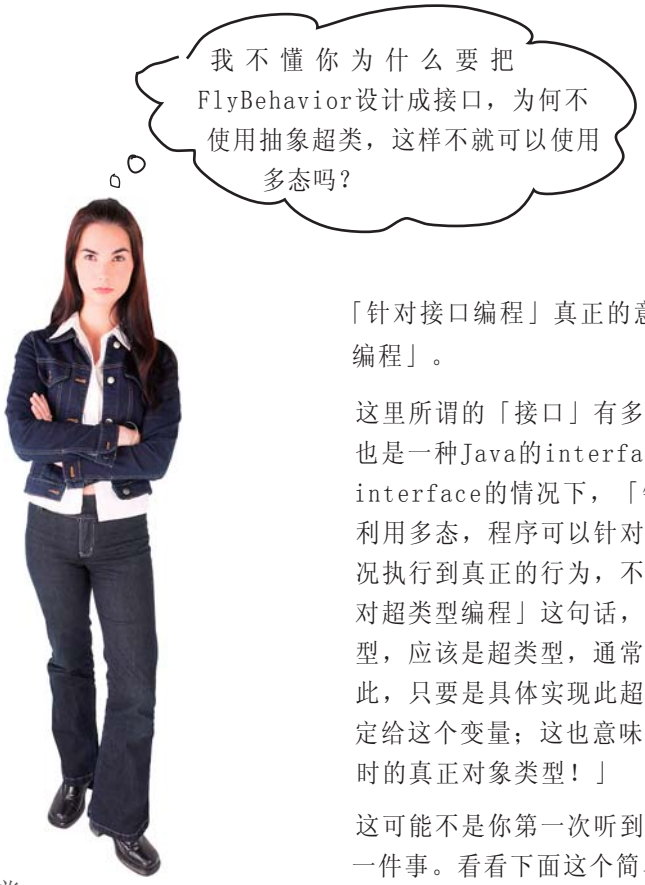
这样的作法迥异于以往，以前的作法是：行为是继承自Duck 超类的具体实现而来，或是继承某个接口并由子类自行实现而来。这两种作法都是依赖于「实现」，我们被实现绑得死死的，没办法更改行为（除非写更多代码）。

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」不会被绑死在鸭子的子类中。（换句话说，特定的实现代码，是位于实现FlyBehavior 与QuakcBehavior 的特定类中）。

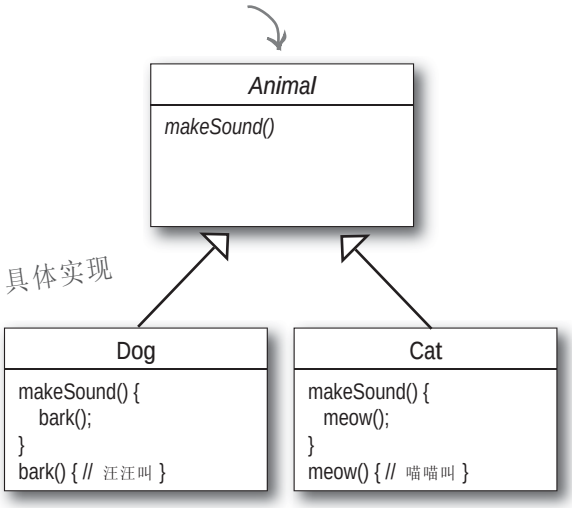
从现在开始，鸭子的行为将被放在分开的类中，此类专门提供某行为的实现。

如此，鸭子类就不再需要知道行为的实现细节。





抽象超类型可以是抽象类[或]接口。



「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」有多个含意，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想的是同一件事。看看下面这个简单的多态范例：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」，作法如下：

```
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实现编码。

但是「针对接口/超类型编程」，作法会如同下面：

```
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，子类型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」。

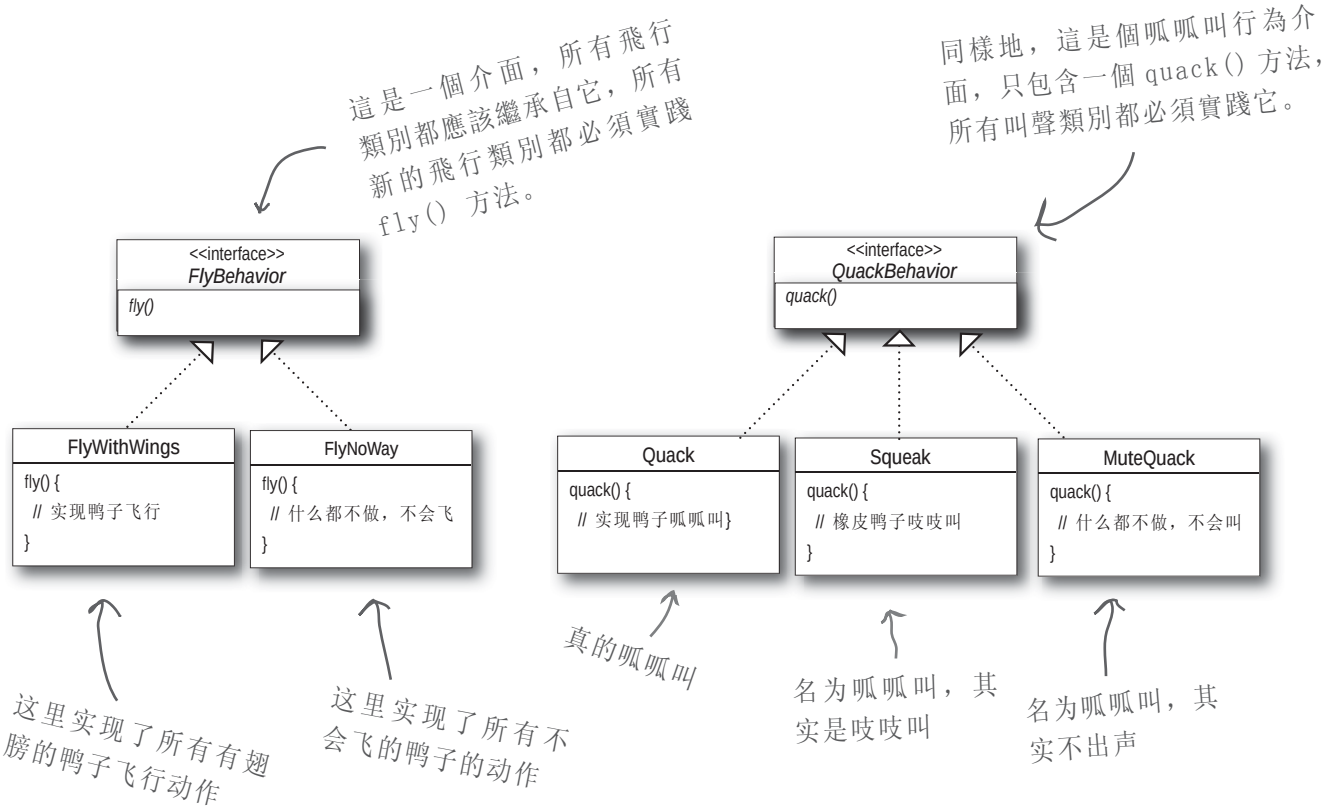
```
a = getAnimal();
a.makeSound();
```

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。

草稿

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，也不会影响有「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。

草稿

类中的行为

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，于是提前在设计上加入这些弹性。你会发现，原则与模式适合使用在软件开发过程中的任何阶段。

问：Duck是不是也该设计成一个接口？

答：在本范例中，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，如此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。我们已经将变化之处移到Duck的外面，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？

答：在OO系统中，是的，类代表的是东西，有状态（实例变量），也有方法。只是在本范例中，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，纪录飞行行为的属性（每秒翅膀拍动几下、最大高度、速度...等）。

削尖你的鉛筆

1. 在我们的新设计中，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叫声行为？

答案：

1）建立一个FlyRocketPowered 类，实现FlyBehavior 接口。

2）例如：鸭哨器（DuckCall）。（一种会产生鸭叫声的装置）

草稿

整合鸭子的行为

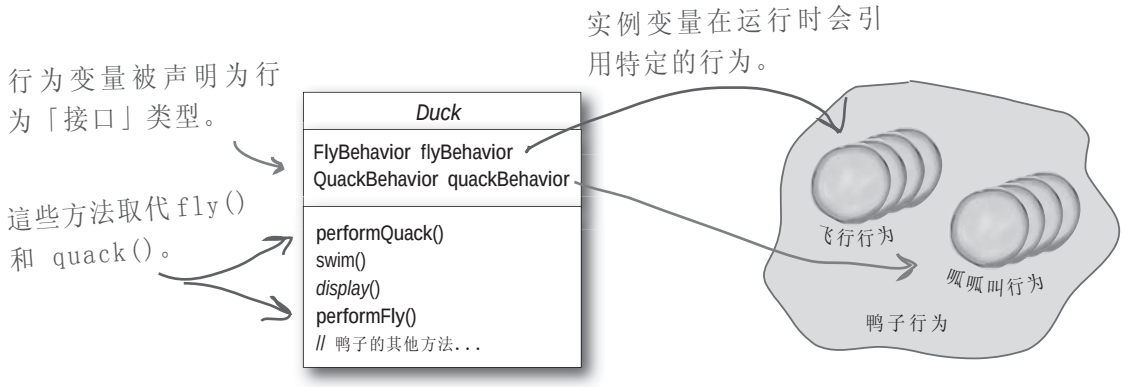
关键在于，鸭子现在会将飞行和呱呱叫的动作，「委托」（delegate）别人处理，而不是使用定义在自己类（或子类）内的方法。

作法是这样的：

1. 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Quackehavior 类中了。

   我们用performFly() 和performQuack() 取代Duck类中的fly() 与quack()。稍后你就知道为什么。



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。

不亲自处理呱呱叫行为，而是委托quackBehavior对象帮Duck呱呱叫。

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。

草稿

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();
   }

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

绿头鸭使用Quack类处理呱呱叫，所以当 performQuack()被调用，就把责任交给Quack对象进行真正的呱呱叫。

使用 FlyWithWings作为其FlyBehavior类型。

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

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对象产生时，它的构造器会把继承来的quackBehavior 实例变量予以初始化成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）。

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

给你逮到了，我们的确是这么做...「只是暂时」。

稍后在书中，我们的工具箱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』改变该行为。

所以，目前的作法还是很有弹性的，只是初始化实例变量的作法不够弹性罢了。但是想一想，因为quackBehavior 实例变量是一个接口类型，能够在运行时，透过多态的魔法动态地指定不同的QuickBehavior 实现类给它。

想想，如何实现鸭子，好让其行为可以在运行时改变。（再几页以后，你就会看到进行这件事的代码。）

测试鸭子的行为

测试Duck的代码

1. 输入下面的Duck类（Duck.java）以及两页前的MallardDuck类（MallardDuck.java），并编译之。

```
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）都继承它们。

委托给行为类

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

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

所有飞行行为类必须实现的接口

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

这是飞行行为的实现，给「真会」飞的鸭子用 ...

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

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）

草稿

继续测试Duck的代码 ...

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

```
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()，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!!
```

草稿

具有动态行为的鸭子

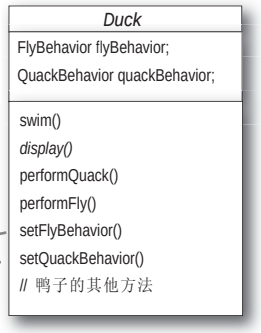
动态地设定行为

在鸭子里建立了一堆动态的功能没有用到，就太可惜了！假设我们想在鸭子子类透过「设定方法（setter method）」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;
}
```



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）

没关系，我们建立一个利用火箭动力的飞行行为。

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

草稿

❹ 改变测试类（MiniDuckSimulator.java），加上模型鸭，并改变前让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();
    }
}
```

改变前

第一次呼叫 performFly() 會把動作委由 flyBehavior 物件（也就是 FlyNoWay 物件）進行，該物件是在建構式中設定的。

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

改变后

如果成功了，就意味着模型鸭动态地改变行为。如果把行为的实现绑死在鸭子类中，可就无法做到这样。

❺ 运行！

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

在运行时想改变鸭子的行为，只要调用鸭子的 setter 方法就可以。

草稿

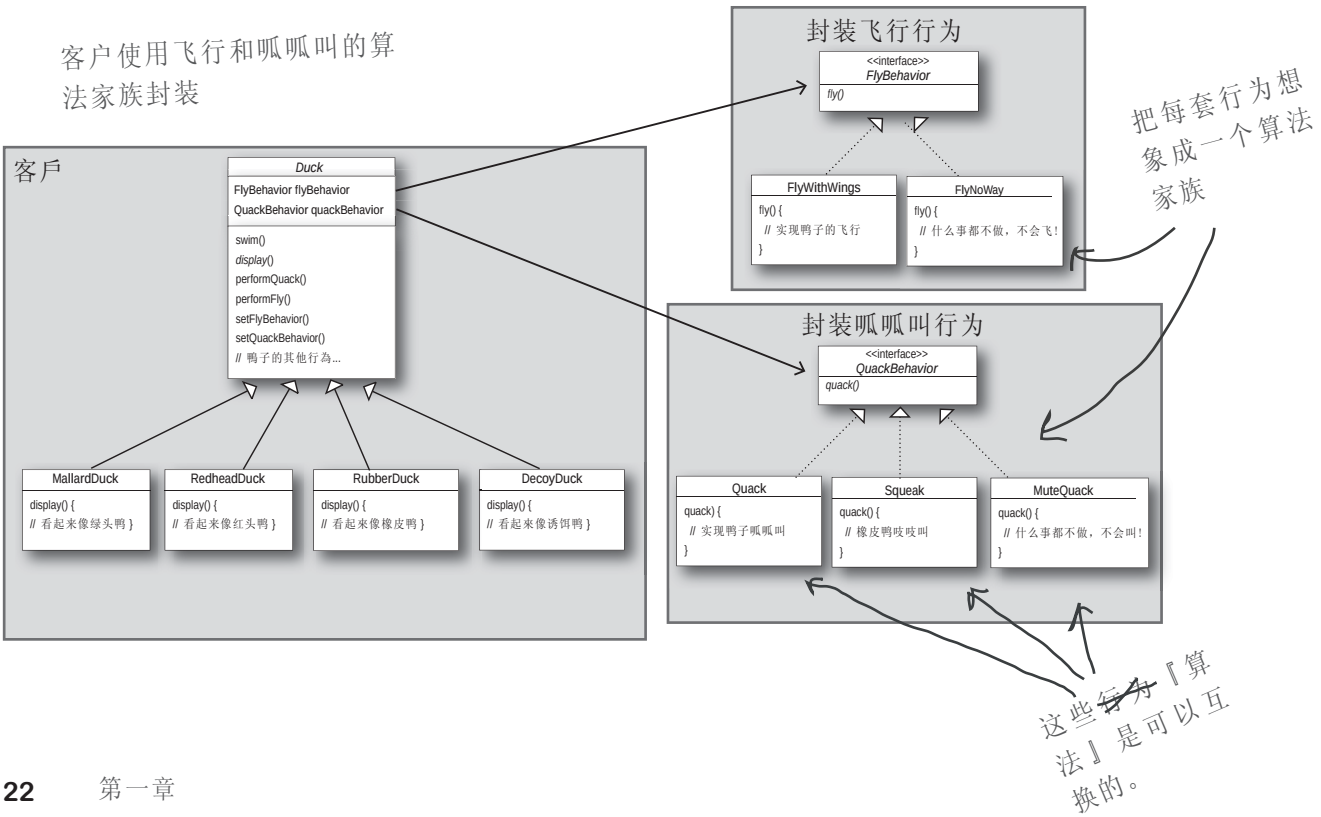
行为封装的大局观

好，我们已经深入鸭子模拟器的设计，该是将头探出水面，呼吸空气的时候了。现在就来看看整体的格局。

下面是整个重新设计后的类架构，你所期望的一切都有：鸭子继承自Duck，飞行行为实现FlyBehavior 接口，呱呱叫行为实现QuackBehavior 接口。

也请注意，我们叙述事情的方式也稍有改变。不再把鸭子的行为说成「一组行为」，我们开始把行为想成是「一群算法」。想想看，在SimUDuck 的设计中，算法代表鸭子能做的事（不同的叫法和飞行法），这样的作法，似乎也能套用在他处，例如：用一群实现相同接口的类，每个类实现不同的营业税计算公式，就可以算出不同的税金。

请特别注意类之间的『关系』。拿一枝笔，把下面图形中的每个箭头标上适当的关系，关系可以是IS -A（是一个）、HAS-A（有一个）、IMPLEMENTS（实现）。



草稿

『有一个』可能比『是一个』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。

当你将两个类结合起来使用，如同本例一般，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。

这是一个很重要的技巧。其实是使用了我们的第三个设计原则：

设计原则

多用组合，少用继承。

如你所见，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』，只要组合的行为对象，符合正确的接口标准即可。

组合用在『许多』设计模式中，在本书中，你也会看到它的诸多优点和缺点。

鸭鸣器（duckcall）是一种装置，猎人用鸭鸣器制造出鸭叫声，以引诱野鸭。你如何实现你自己的鸭鸣器，而不继承自Duck类？

大师与门徒...

大师：蚱蜢，告诉我，在面向对象的道路上，你学到了什么？

门徒：大师，我学到了，面向对象之路，可以『复用』。

大师：继续说..

门徒：大师，藉由继承，好东西可以一再被利用，所以程序开发时间就会大幅减少，就好像在林中很快地把竹子截短一样。

大师：蚱蜢呀！软件开发完成『前』以及完成『后』，何者需要花费更多时间呢？

门徒：答案是『后』，大师。我们总是需要花许多时间在系统的维护和变化上，比原先开发花的时间更多。

大师，我说蚱蜢，这就对啦！那么我们是不是应该致力于提高可维护性和可扩充性上的复用程度呀？

门徒：是的，大师，的确是如此。

大师：我觉得你还有很多东西要学，希望你再深入研究继承。你会发现，继承有它的问题，还有一些其他的方式可以达到复用。

草稿

讲到设计模式…

恭喜你，学会第一个模式了！

你刚刚用了你的第一个设计模式：也就是策略模式（Strategy Pattern）。不要怀疑，你正是使用策略模式改写SimUDuck 程序的。感谢这个模式，现在系统不担心遇到任何改变，主管们可以到赌城狂欢。

为了介绍这个模式，我们走了很长的一段路。下面是此模式的正式定义：

『策略模式』定义了算法家族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化，不会影响到使用算法的客户。

当你需要朋友对你印象深刻，或是想影响关键主管的决策时，请使用『这个』定义。

草稿

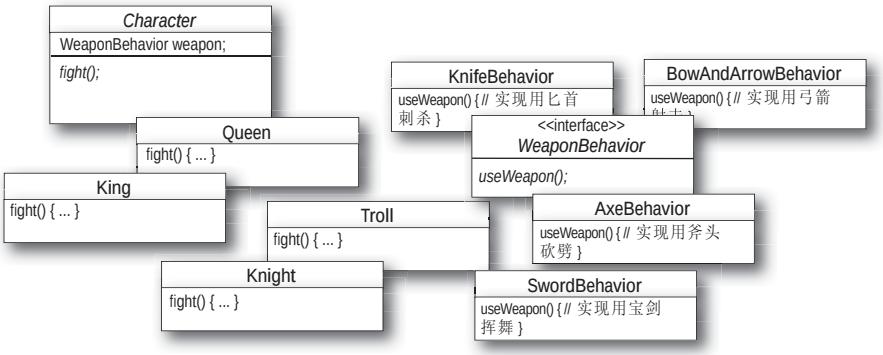
设计谜题

在下面，你将看到一堆杂乱的类与接口，这是取自一个交互式的冒险游戏。你将看到代表游戏角色的类，以及武器行为的类。每个角色一次只能使用一个武器，但是可以在游戏的过程中换武器。你的工作是要弄清楚这一切...

（答案在本章末）

你的任务：

1. 安排类。
2. 找出一个抽象类、一个接口、以及八个类。
3. 在类之间画箭头。
   a. 继承就画成这样（『extend』）。
   b. 实现接口就画成这样（『implement』）。
   c. 『有一个』关系就画成这样。
4. 把setWeapon() 方法放到正确的类中。




```
setWeapon(WeaponBehavior w) {
    this.weapon = w;
}
```

草稿

在附近餐厅中无意间听到...

这两人点的餐有何不同？其实没有差异，其实都是一份单，只是Alice讲话的长度多了一倍，而且快餐店的厨师已经感到不耐烦了。

什么是Flo有的，而Alice没有？答案是，Flo和厨师之间有『共享的词汇』，Alice却不懂这些词汇。共享的词汇不仅方便顾客点餐，也让厨师不用记太多事，毕竟这些餐点模式都已经在他的脑海中了呀！

设计模式让你和其他开发人员之间有共享的词汇，一旦懂这些词汇，和其他开发人员之间沟通这些观念就很容易，也会激起那些不懂的程序员想开始学习设计模式。设计模式也可以帮助你提升思考架构的层次到模式层面，而不是停留在琐碎的对象上。

草稿

在办公室隔间中无意间听到...

我建立了这一个广播类，能够追踪所有的对象，只要有新资料进来，就会通知每个倾听者。最棒的是，倾听者可以随时加入此广播系统，甚至可以随时将自己从系统中移除。如此的设计方式相当的动态，对象之间的依赖程度也会降低。

動動腦

除了面向对象设计和在餐厅点餐之外，你还能够想到有那些例子是需要共享的词汇？（暗示：想一想汽修、木工、料理、航管）利用这些专业术语进行沟通的质量如何？

你能否想到有什么OO设计上的东西，能够和模式名称匹配的？利用「策略模式」这个名字是否传神？

Rick，你只要说使用了「观察者模式」我们就懂了。

没错，如果你用模式名称和大家沟通，其他开发人员能够马上且清楚地知道你在说些什么。但是也请不要从此染上『模式疯』...以后连写一个『HelloWorld』都能够扯上模式，那就代表你已经病了...

草稿

共通词汇

共享模式词汇的威力

你使用模式和他人沟通，其实『不只是』和他人共享『术语』而已。

共享的模式词汇『威力强大』。

当你使用模式名称和其他开发人员或者开发团队沟通时，你们之间交流的不只是模式名称，而是一整套模式背后所象征的质量、特性、约束。

模式能够让你用更少的词汇做更充分的沟通。

当你用模式描述的时候，其他开发人员便很容易地知道你对设计的想法。

『我们使用策略模式实现鸭子的各种行为。』这句话也就是告诉我们，鸭子的行为被封装进入一组类中，可以被轻易地扩充与改变。如果有需要，甚至在运行时也可以改变行为。

将说话的方式保持在模式层次，可让你待在『设计圈子』久一点。

使用模式谈论软件系统，可以让你保持在设计层次，不会被压低到对象与类这种琐碎的事情上面。

想想看，有多少次的设计会议中，你们一不小心就进入了琐碎的实现细节的讨论上。

共享词汇可帮你的开发团队快速充电。

对于设计模式有深入了解的团队，彼此之间对于设计的看法不容易产生误解。

当你的团队开始利用模式分享设计想法与经验，你等于是建立了一个模式使用者的群落。

共享词汇能帮助初级开发人员迅速成长。

初级开发人员向有经验的开发人员看齐。当高级开发人员使用设计模式，初级开发人员也会有样学样。把你的组织建立成一个模式用户的社区。

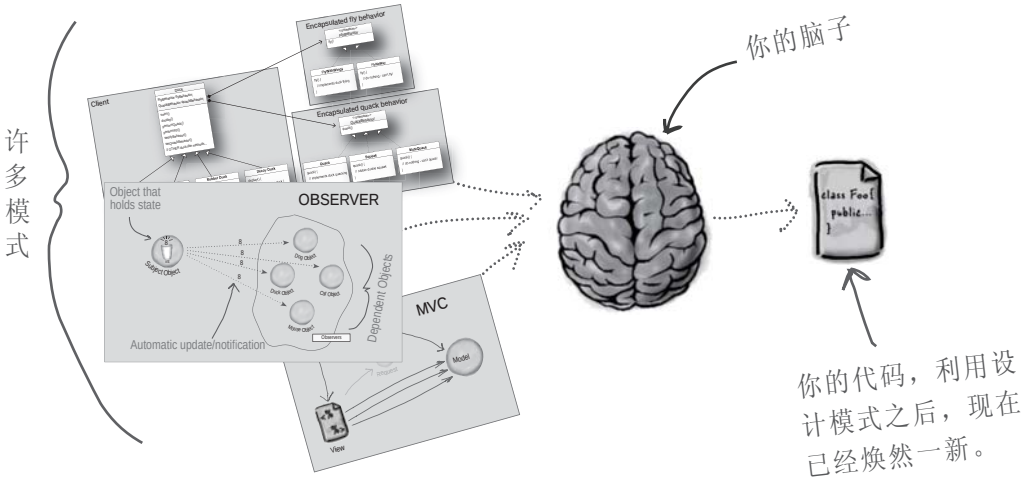
考虑在你的组织内发起一个设计模式研讨会，说不定在学习的过程中，就开始得到回报了...

草稿

我如何使用设计模式？

我们全都使用别人设计好的库与框架。我们讨论库与框架、利用它们的API 编程、编译成我们的程序、享受运用别人的代码所带来的优点。看看Java API 以及它所带来的功能：网络、GUI、IO，等等。库与框架长久以来，一直扮演着软件开发过程的重要角色，我们从中挑选所要的组件，把它们放进正确的地方。但是...库与框架无法帮助我们将应用组织成容易了解、容易维护、具有弹性的架构，所以需要设计模式。

设计模式不会直接进入你的代码中，而是必须先进入你的『脑袋』中。一旦你先在脑海中装入许多模式的知识，就能够开始在新设计中采用它们，以及当你的旧代码变得如同意大利面一样搅和成一团没弹性时，可用它们改写代码。



问：如果设计模式这么棒，为何没有人建立相关的库，我们就不必自己动手了？

答：设计模式比库的等级更高。设计模式告诉我们如何组织类和对象，以解决某类型的问题。采纳这些设计并使它们适合我们自己的应用，是我们责无旁贷的事。

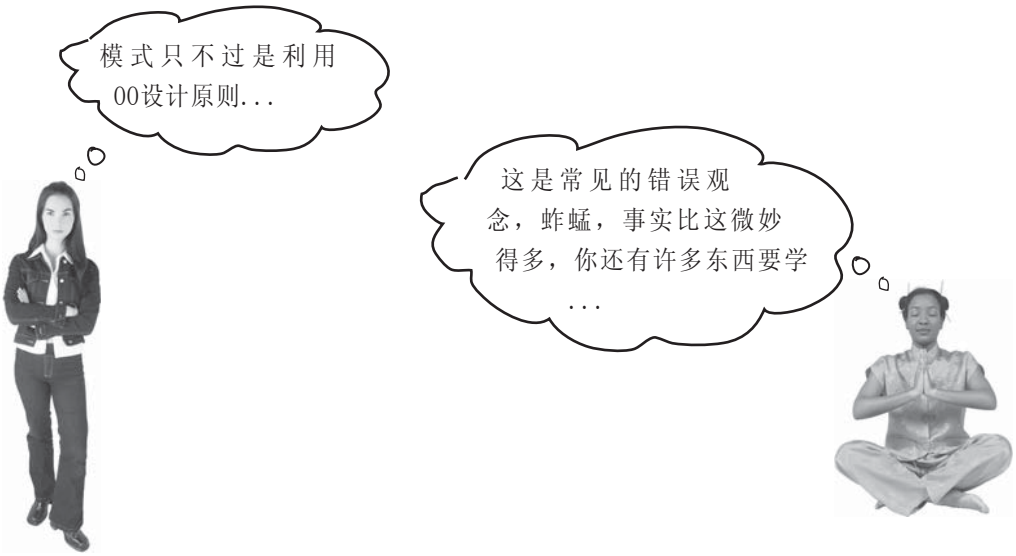
问：库和框架不也是设计模式吗？

答：库和框架提供了我们某些特定的实现，让我们的代码可以轻易地引用，但是这并不算是设计模式。有些时候，库和框架本身会用到设计模式，这样很好，因为一旦你了解了设计模式，会更容易了解这些API 是围绕着设计模式构造的。

问：那么，没有所谓设计模式的库？

答：没错，但是稍后你会看到设计模式分类列表。你可以在应用中利用这些设计模式。

为何要用设计模式？



开发人员：好吧！但是不都只是好的面向对象设计吗？我是说，我懂得运用封装、抽象、继承、多态，我真的还有必要用设计模式思考吗？运用00，一切不是都很直觉吗？这不正是我过去上了一堆00 课程的原因吗？我认为设计模式只对那些不懂好的00 设计的人有用。

大师：这是面向对象开发常有的谬误：以为知道00 基础概念，就能自动设计出弹性的、可复用的、可维护的系统。

开发人员：不是这样吗？

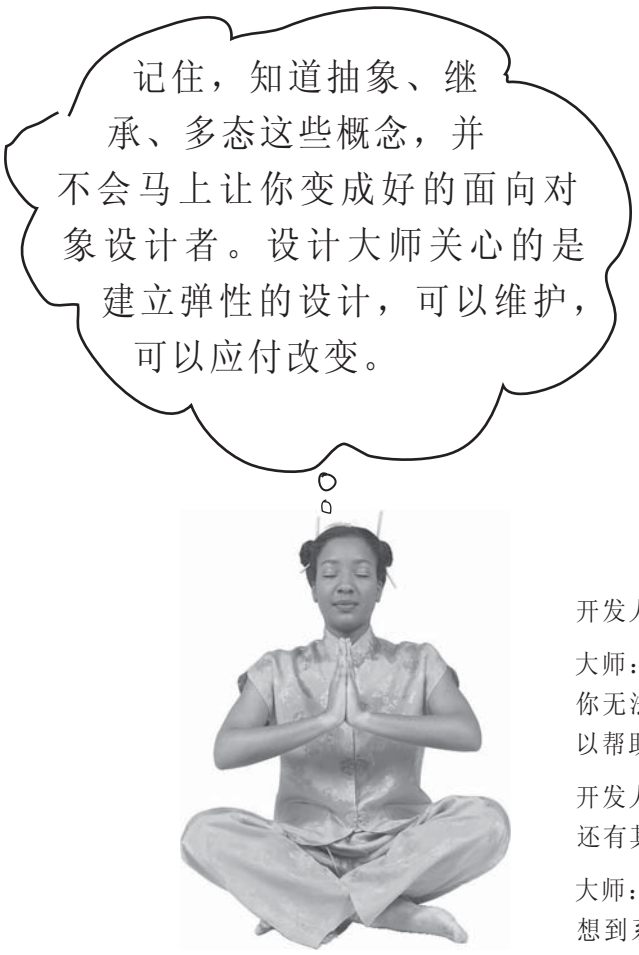
大师：不是！要构造00 系统不光只有懂这些观念就可以，事实证明只有透过不断地艰苦实践，才能成功。

开发人员：我想我开始了解了，这些构造00 系统的经验于是被整理出来...

大师：...是的，被整理成了一群『设计模式』。

开发人员：那么，如果知道了这些模式，我就可以减少许多体力劳动，直接采用可行的模式。

大师：对的，可以这么说。不过要记得，设计是一种艺术，总是有许多取舍的地方。但是如果你能采用这些经过深思熟虑，且通过时间考验的设计模式，你就领先别人了。



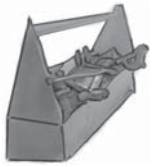
开发人员：如果我找不到模式，怎么办？

大师：有一些面向对象原则，适用于所有的模式。当你无法找到适当的模式解决问题时，采用这些原则可以帮助你。

开发人员：原则？你是说除了抽象、封装... 之外，还有其他？

大师：是的，建立可维护的00 系统，要诀就在于随时想到系统以后可能需要的变化，现在要如何设计，以应付以后的变化。

草稿



设计工具箱内的工具

你几乎快要读完第一章了！你已经在你的设计工具箱内放进了几样工具，在我们进入第二章之前，先将这些工具一一列出。

00基础
- 抽象
- 封装
- 多态
- 继承

我们假设你知道00基础，包括了多态的用法、继承就像按契约设计、封装如何运作。如果你觉得脑袋有一点生锈了，快快拿出你的《Head First Java》复习，然后再略读过本章一次。

00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

我们会在后续的内容更详细地看看这些原则，还会再多加一些原则到清单上。

阅读本书时，时时刻刻要思考着：模式如何仰赖基础与原则。

00 模式

『策略』——定义算法家族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化不会影响到使用算法的客户。

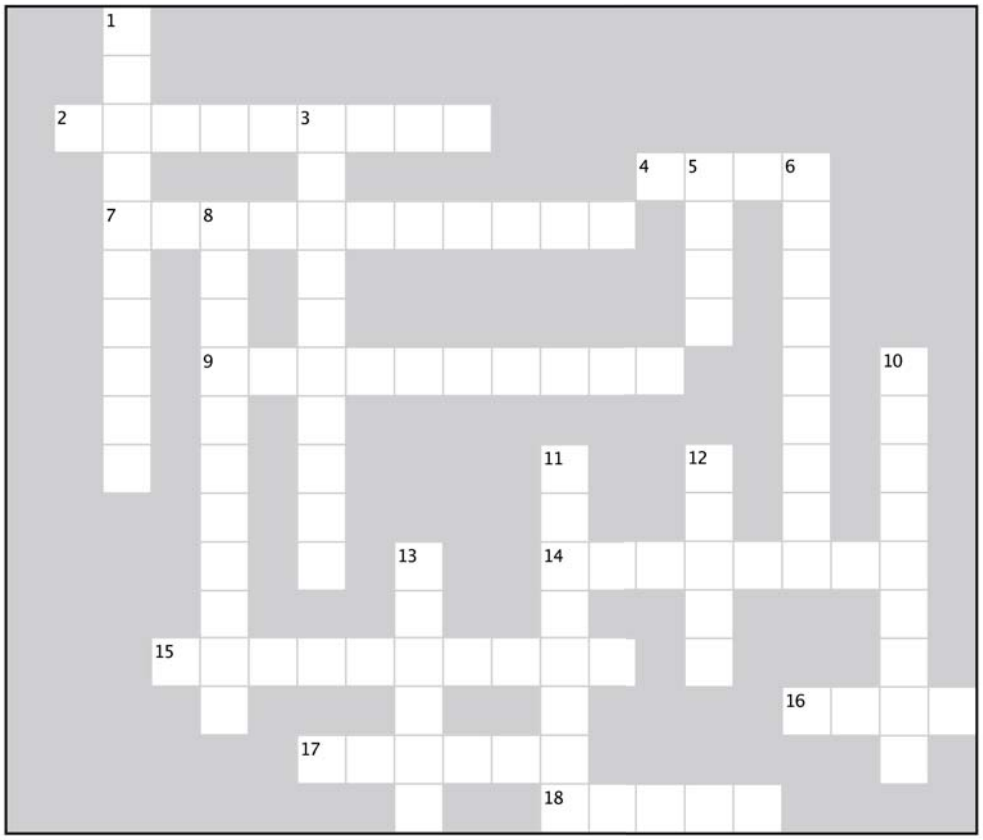
学了一个，还有更多！

要点

- 知道00 基础，并不足以让你设计出良好的00系统。
- 良好的00 设计必须具备可复用、可扩充、可维护三个特性。
- 模式可以让我们构造出具有良好00 设计质量的系统。
- 模式被认为是00 设计经验的精华所在。
- 模式不是代码，而是针对设计问题的通用解决方案。你把它们应用到特定的应用中。
- 模式不是被发明，而是被发现。.
- 大多数的模式和原则，都着眼于软件改变的主题上。
- 大多数的模式都允许系统局部改变，而不影响其他部分。
- 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语言，最大化沟通的价值。

草稿

让标准填字游戏，动动你的右脑。



横排提示：

2. Grilled cheese with bacon
4. Duck demo was located where
7. _____ what varies
9. Most patterns follow from OO _____
14. Pattern that fixed the simulator
15. Patterns give us a shared _____
16. Design patterns _____
17. Development constant
18. Patterns ____ in many applications

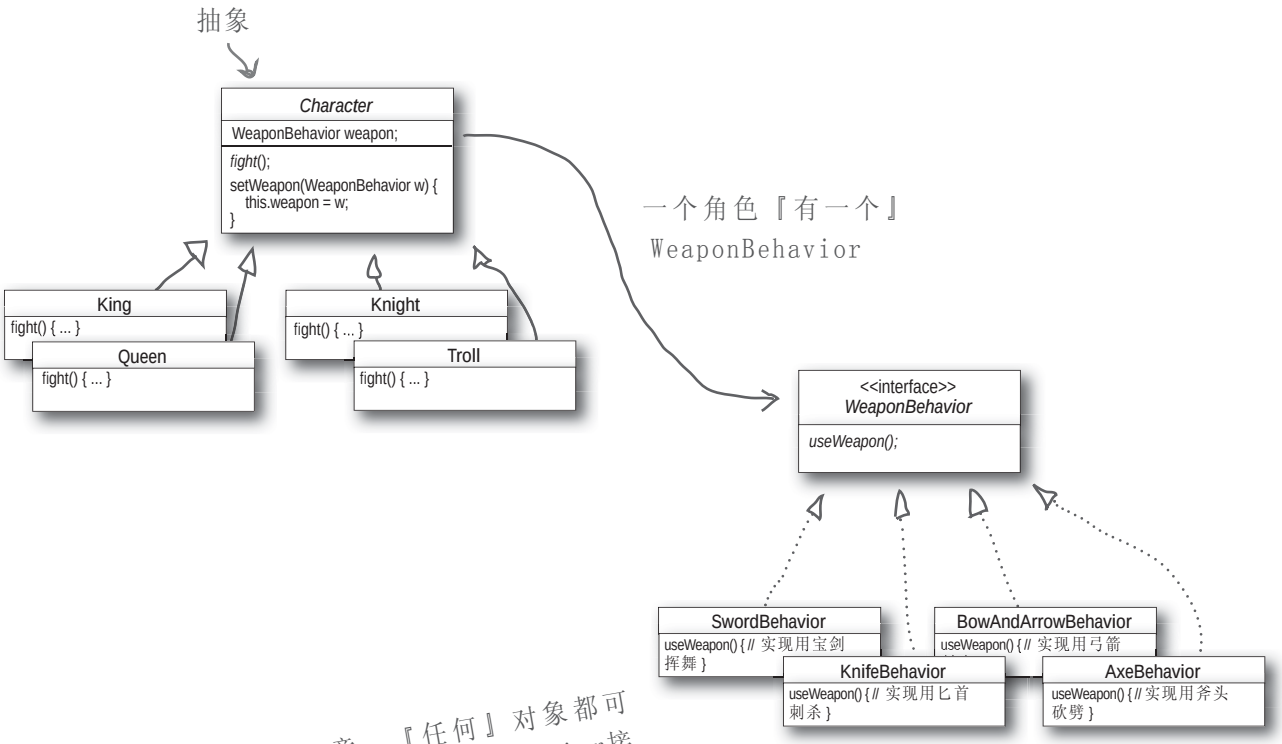
直排提示：

1. High level libraries
3. Learn from the other guy's _____
5. Java IO, Networking, Sound
6. Program to this, not an implementation
8. Favor over inheritance
10. Duck that can't quack
11. Rick was thrilled with this pattern
12. Patterns go into your _____
13. Rubberducks make a _____

设计谜题解答

Character（角色）是抽象类，让具体的角色继承之。具体的角色包括：国王（King）、皇后（Queen）、骑士（Knight）、妖怪（Troll）。而Weapon（武器）是接口，让具体的武器继承之。所有实际的角色和武器都是具体类。

任何角色如果想换武器，可以调用setWeapon() 方法，此方法定义在Character超类中。在打斗（flight）过程中，会调用到目前武器的useWeapon() 方法，攻击其他角色。

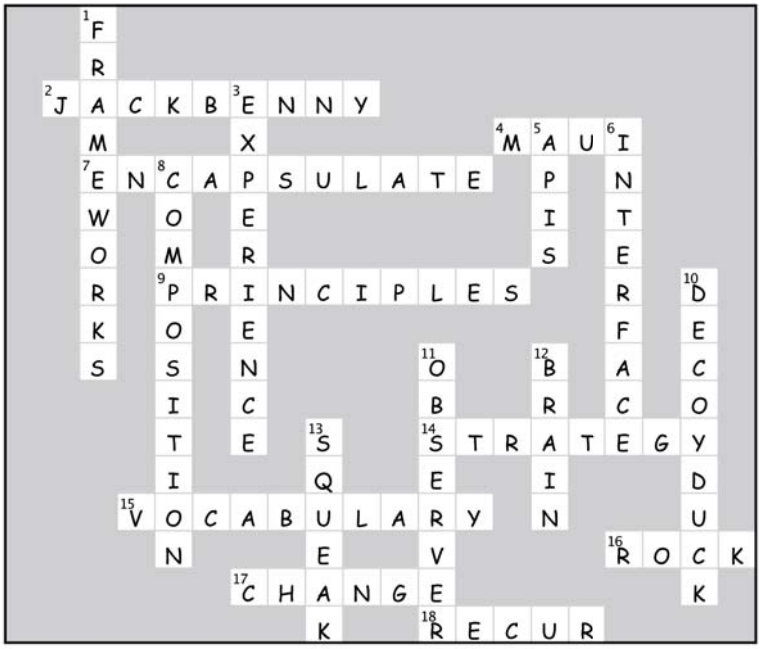


答案

削尖你的鉛筆

利用继承来提供Duck的行为，这会导致下列哪些缺点？（多选）

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



削尖你的鉛筆

驱使改变的因素很多。找出你的软件中需要改变代码的地方，一一列出来。下面是我们的答案，你的答案可能和我们不一样。

我们的顾客或用户决定要别的做法，或者想要新功能。

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

唔！应对技术改变，我们必须更新代码，适用于新协议。

我们学到了足够的构造系统的知识，希望回头去把事情做得更好。