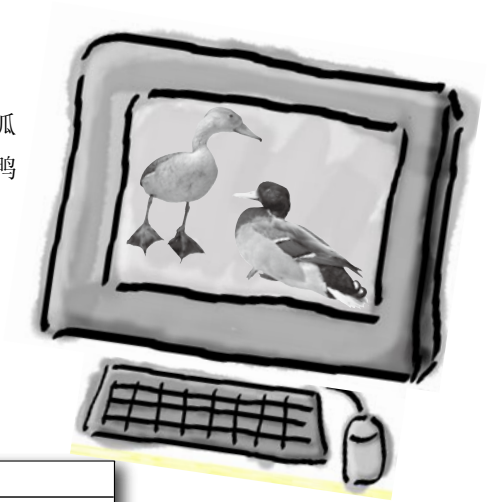
1 设计模式入门

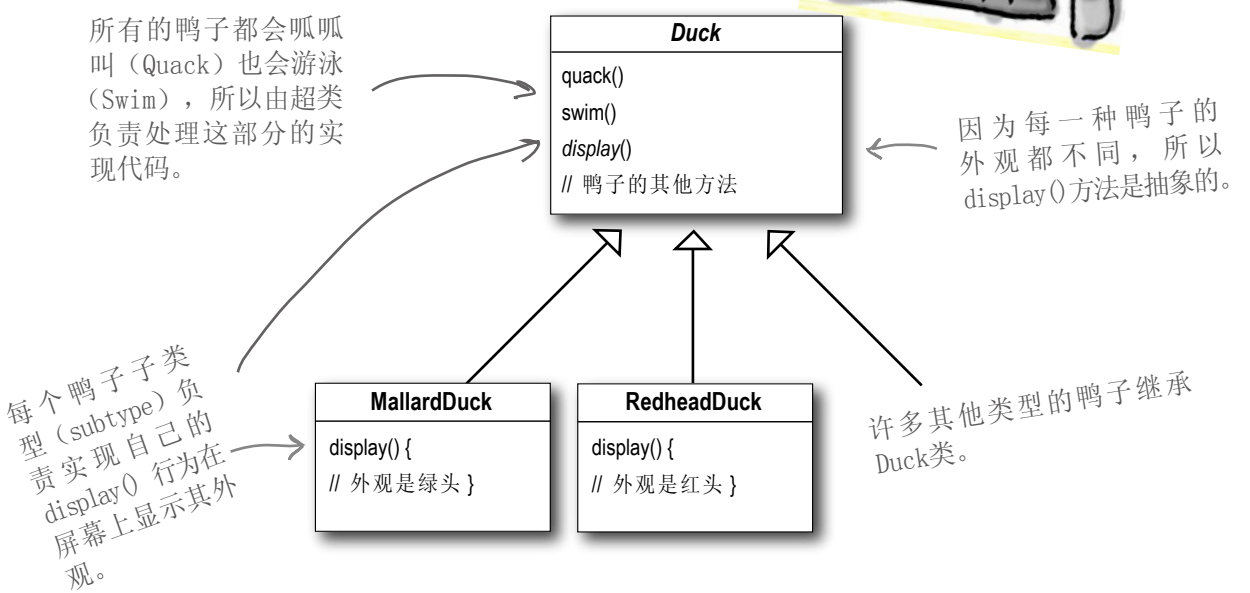
欢迎来到设计模式世界

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。

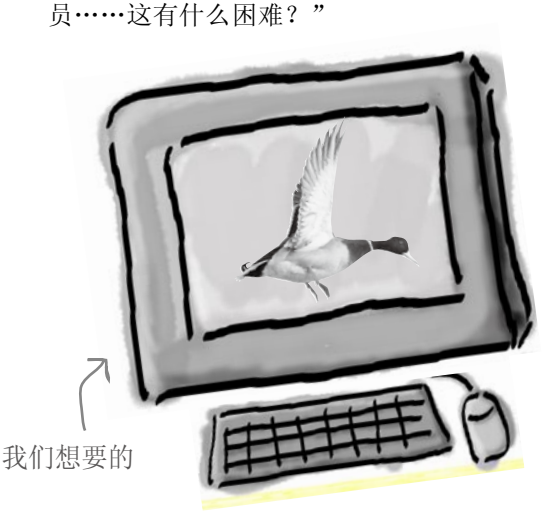




去年，公司的竞争压力加剧。在为期一周的高尔夫假期兼头脑风暴会议之后，公司主管认为该是创新的时候了，他们需要在“下周”毛伊岛股东会议上展示一些“真正”让人印象深刻的东西来振奋人心。

现在我们得让鸭子能飞

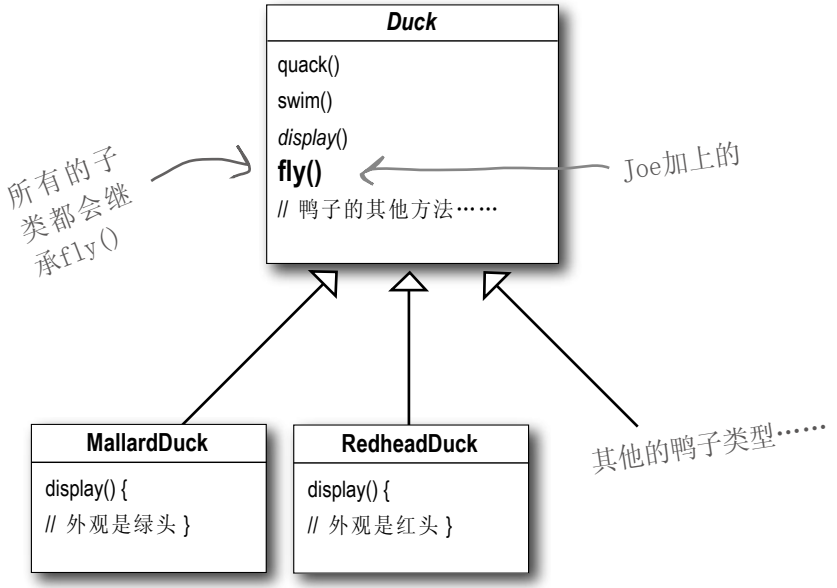
主管们确定，此模拟程序需要会飞的鸭子来将竞争者抛在后头。当然，在这个时候，Joe的经理拍胸脯告诉主管们，Joe只需要一个星期就可以搞定。“毕竟，Joe是一个OO程序员……这有什么困难？”



我们想要的

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。

Joe



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？

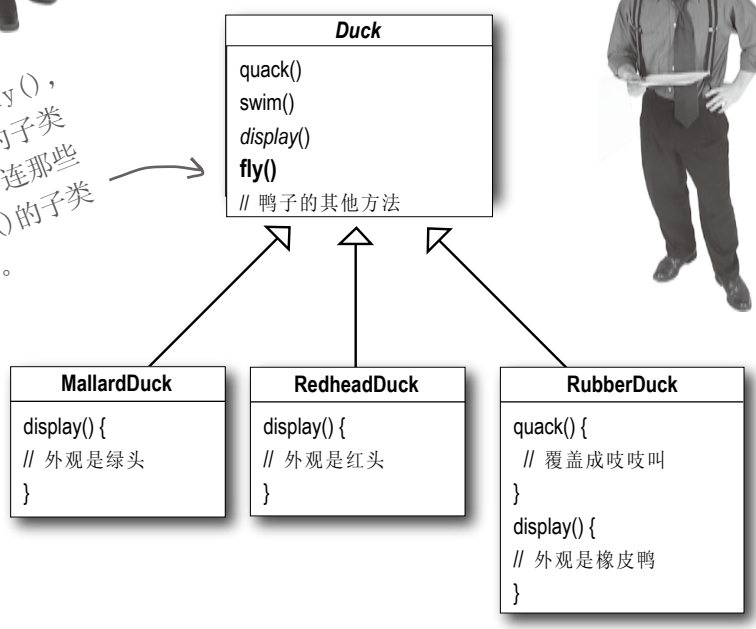
Joe忽略了一件事：并非Duck所有的子类都会飞。Joe在Duck超类中加上新的行为，会使得某些并不适合该行为的子类也具有该行为。现在可好了！SimUDuck程序中有了一个无生命的会飞的东西。

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……

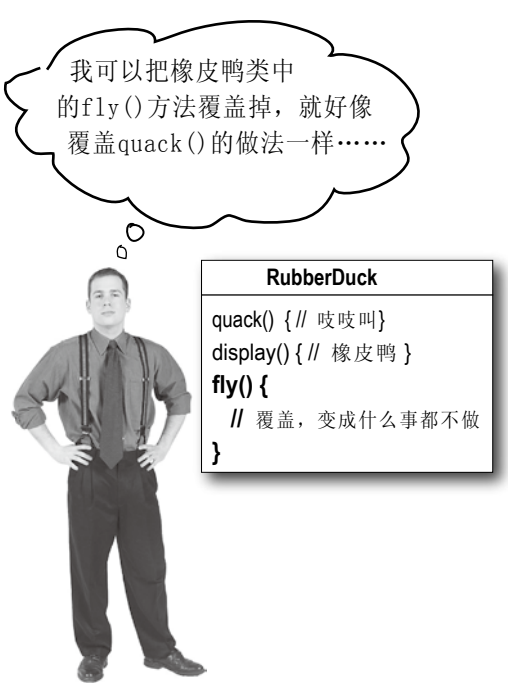
他体会到了一件事：当涉及“维护”时，为了“复用”（reuse）目的而使用继承，结局并不完美。

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）。

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。

Sharpen your pencil

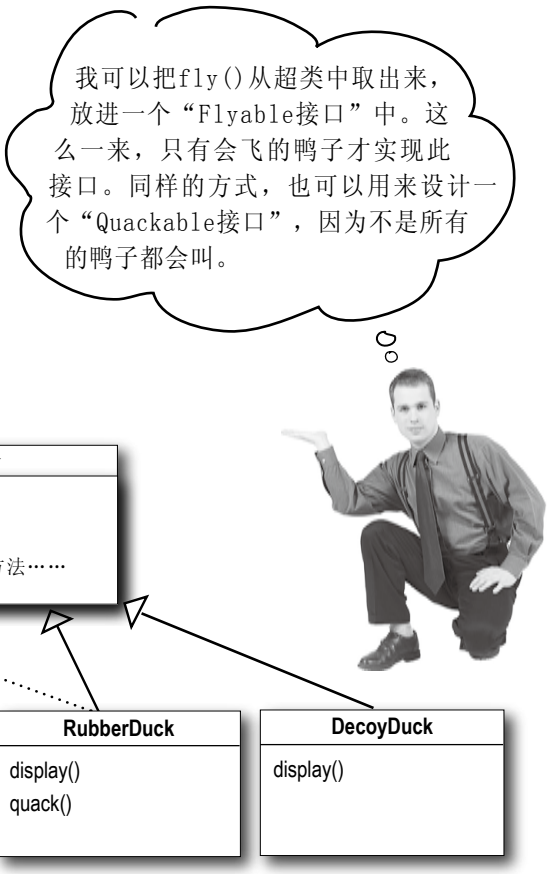
利用继承来提供Duck的行为，这会导致下列哪些缺点？（多选）

- q A. 代码在多个子类中重复。
- q B. 运行时的行为不容易改变。
- q C. 我们不能让鸭子跳舞。
- q D. 很难知道所有鸭子的全部行为。
- q E. 鸭子不能同时又飞又叫。
- q F. 改变会牵一发动全身，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quark()……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。



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，有什么是你可以深信不疑的？

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？

ƎGИAHƆ

（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。

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。

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。

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！

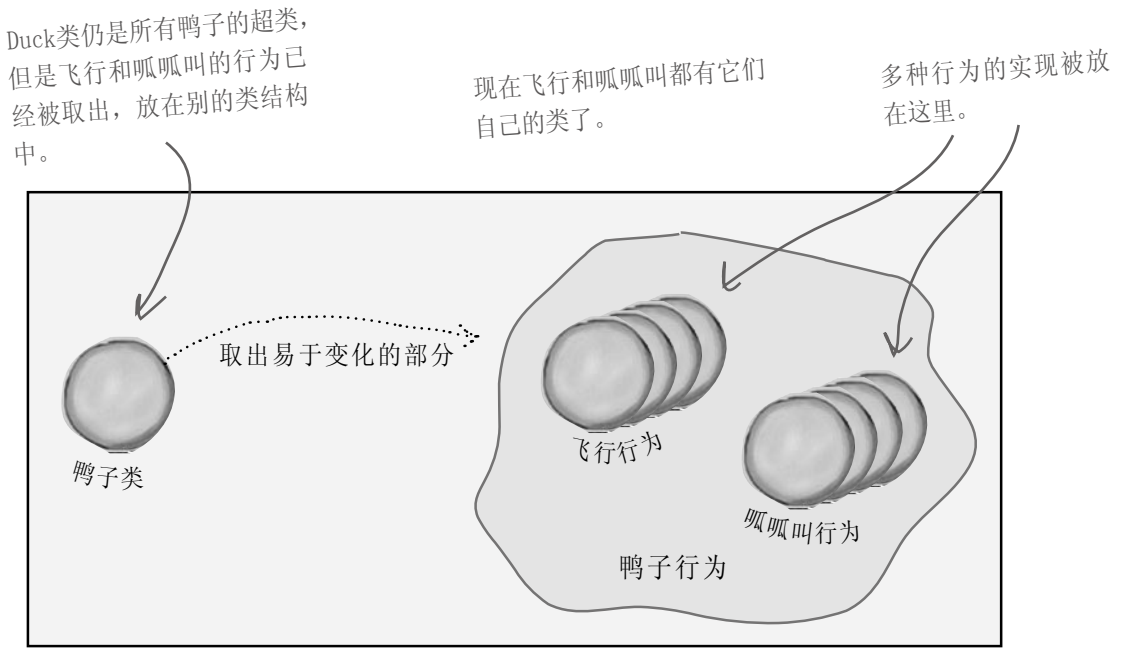
分开变化和不会变化的部分

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。



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。我们还想能够“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：

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。

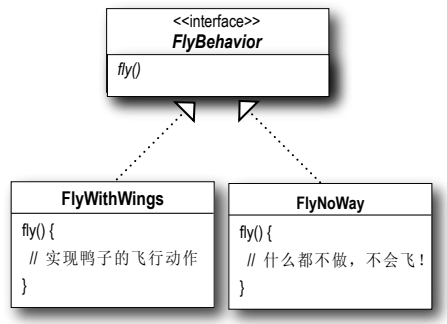
所以这次鸭子类不会负责实现Flying与Quacking接口，反而是由我们制造一组其他类专门实现FlyBehavior与QuackBehavior，这就称为“行为”类。由行为类而不是Duck类来实现行为接口。

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）。

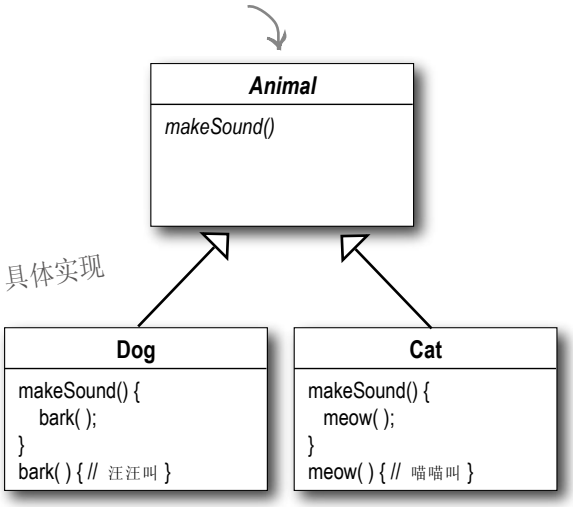
在我们的新设计中，鸭子的子类将使用接口（FlyBehavior与QuackBehavior）所表示的行为，所以实际的“实现”不会被绑死在鸭子的子类中。（换句话说，特定的具体行为编写在实现了FlyBehavior与QuakcBehavior的类中）。

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。



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。

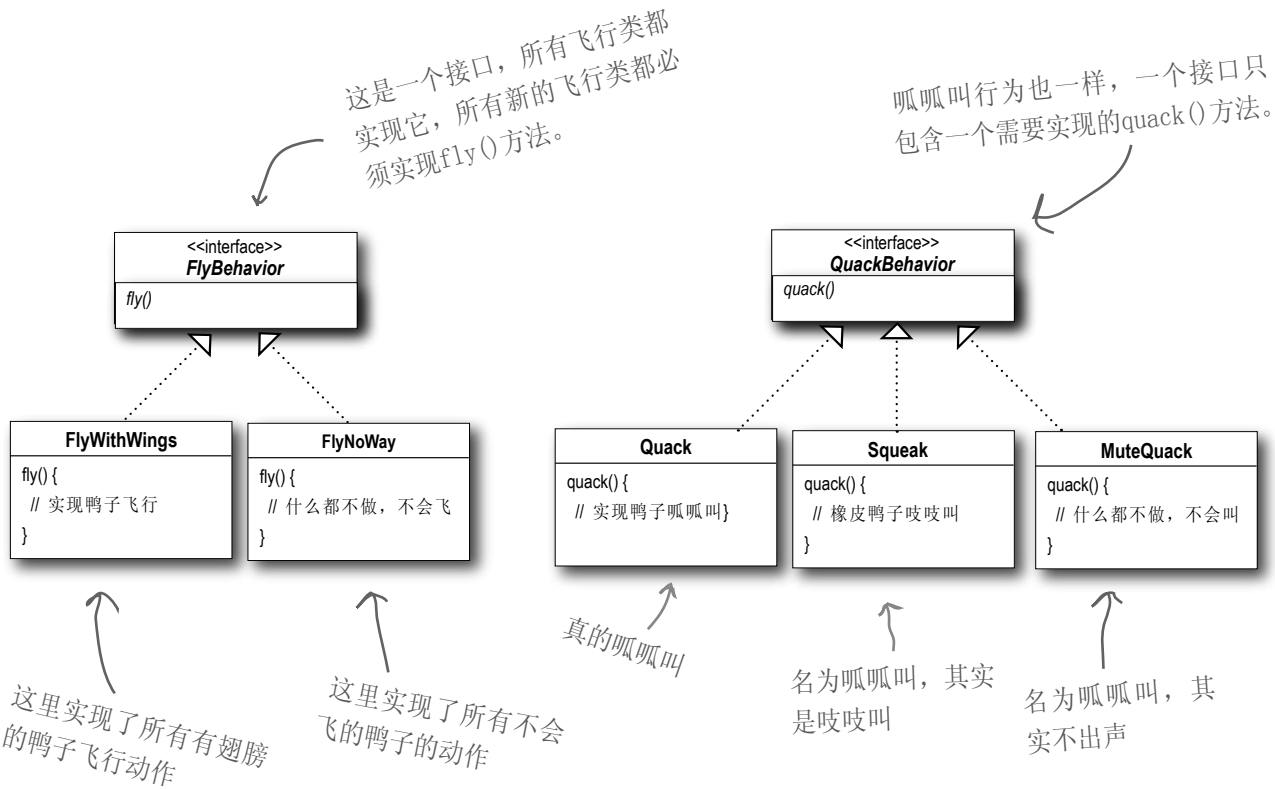
更棒的是，子类实例化的动作不再需要在代码中硬编码，例如new Dog()，而是“在运行时才指定具体实现的对象”。

```
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。

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）。

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

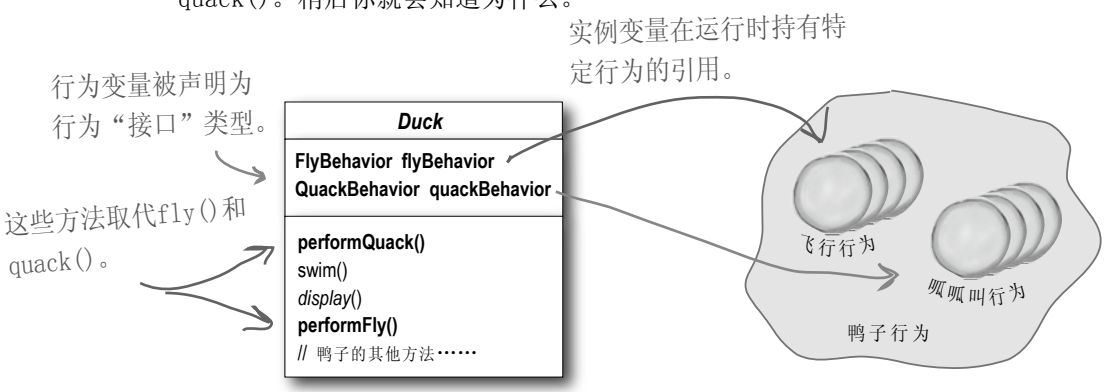
关键在于，鸭子现在会将飞行和呱呱叫的动作“委托”（delegate）别人处理，而不是使用定义在Duck类（或子类）内的呱呱叫和飞行方法。

做法是这样的：

❶ 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。



❷ 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();
   }

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

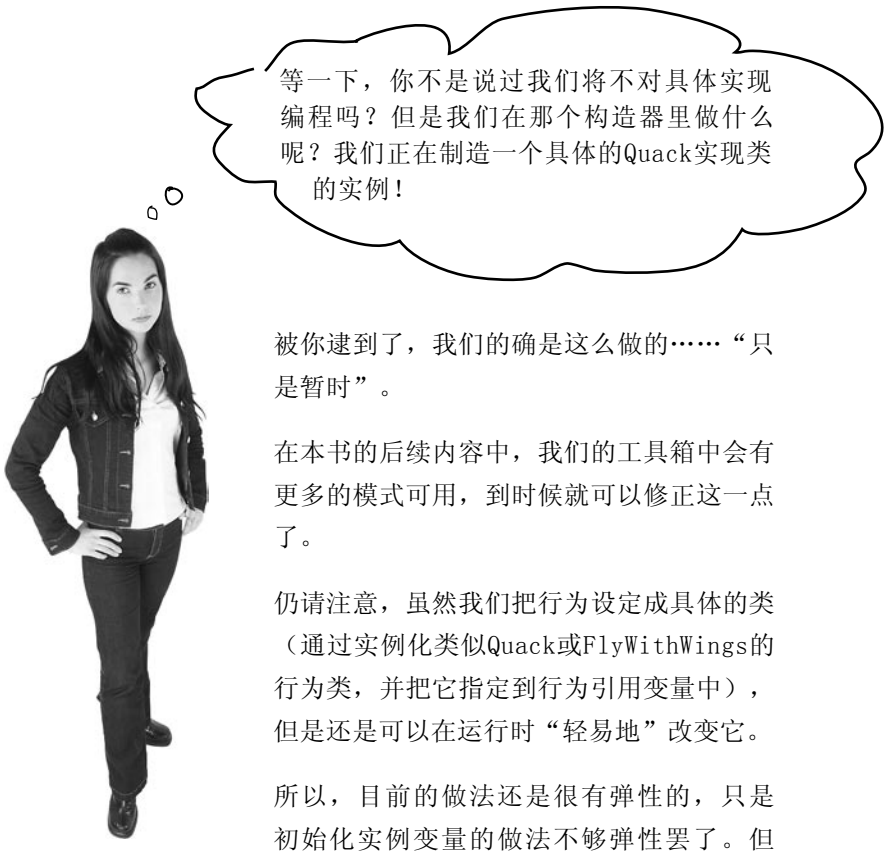
绿头鸭使用Quack类处理呱呱叫，所以当performQuack()被调用时，叫的职责被委托给Quack对象，而我们得到了真正的呱呱叫。

使用FlyWithWings作为其FlyBehavior类型。

别忘了，因为MallardDuck继承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）。



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地指定不同的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

❶ 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

❷ 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();
}
```

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）。

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！

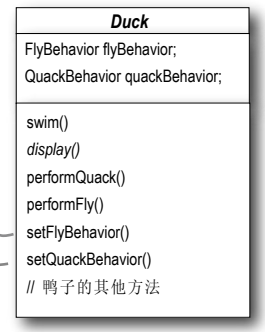
```
File Edit Window Help Yadayadayada
%java MiniDuckSimulator
Quack
I'm flying!!
```

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。

❶ 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。

❷ 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。

❸ 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!");
    }
}
```

❹ 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死在鸭子类中，可就无法做到这样了。

❺ 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。

中国互动出版网
www.china-pub.com
网上书店 独家提供样章

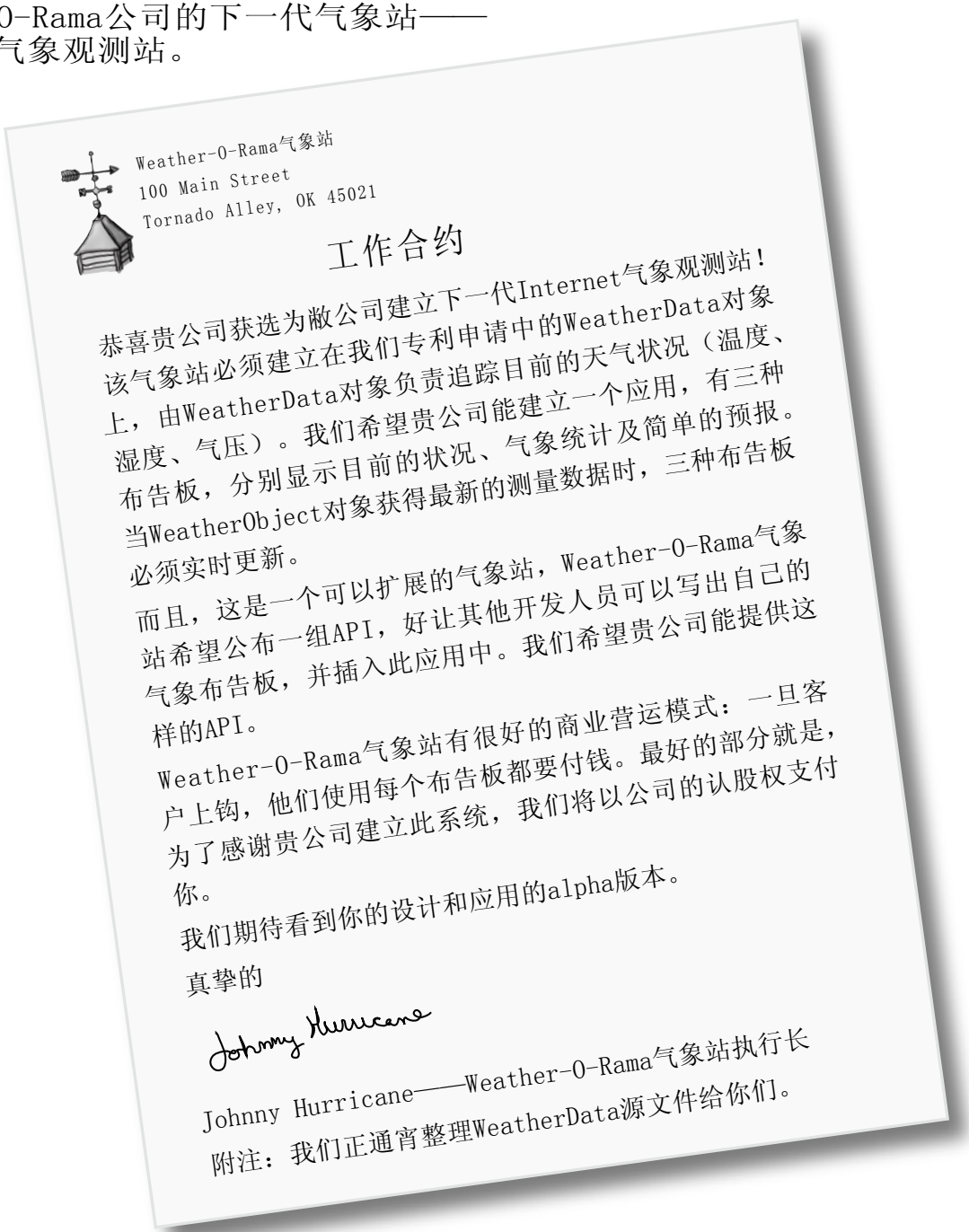
2 观察者（Observer）模式

让你的对象知悉现况

有趣的事情发生时，可千万别错过了！有一个模式可以帮你的对象知悉现况，不会错过该对象感兴趣的事。对象甚至在运行时可决定是否要继续被通知。观察者模式是JDK中使用最多的模式之一，非常有用。我们也会一并介绍一对多关系，以及松耦合（对，没错，我们说耦合）。有了观察者，你将会消息灵通。

恭喜你！

你的团队刚刚赢得一纸合约，负责建立Weather-O-Rama公司的下一代气象站——Internet气象观测站。



Weather-O-Rama气象站
100 Main Street
Tornado Alley, OK 45021

工作合约

恭喜贵公司获选为敝公司建立下一代Internet气象观测站！该气象站必须建立在我们专利申请中的WeatherData对象上，由WeatherData对象负责追踪目前的天气状况（温度、湿度、气压）。我们希望贵公司能建立一个应用，有三种布告板，分别显示目前的状况、气象统计及简单的预报。当WeatherObject对象获得最新的测量数据时，三种布告板必须实时更新。

而且，这是一个可以扩展的气象站，Weather-O-Rama气象站希望公布一组API，好让其他开发人员可以写出自己的气象布告板，并插入此应用中。我们希望贵公司能提供这样的API。

Weather-O-Rama气象站有很好的商业营运模式：一旦客户上钩，他们使用每个布告板都要付钱。最好的部分就是，为了感谢贵公司建立此系统，我们将以公司的认股权支付你。

我们期待看到你的设计和应用的alpha版本。

真挚的

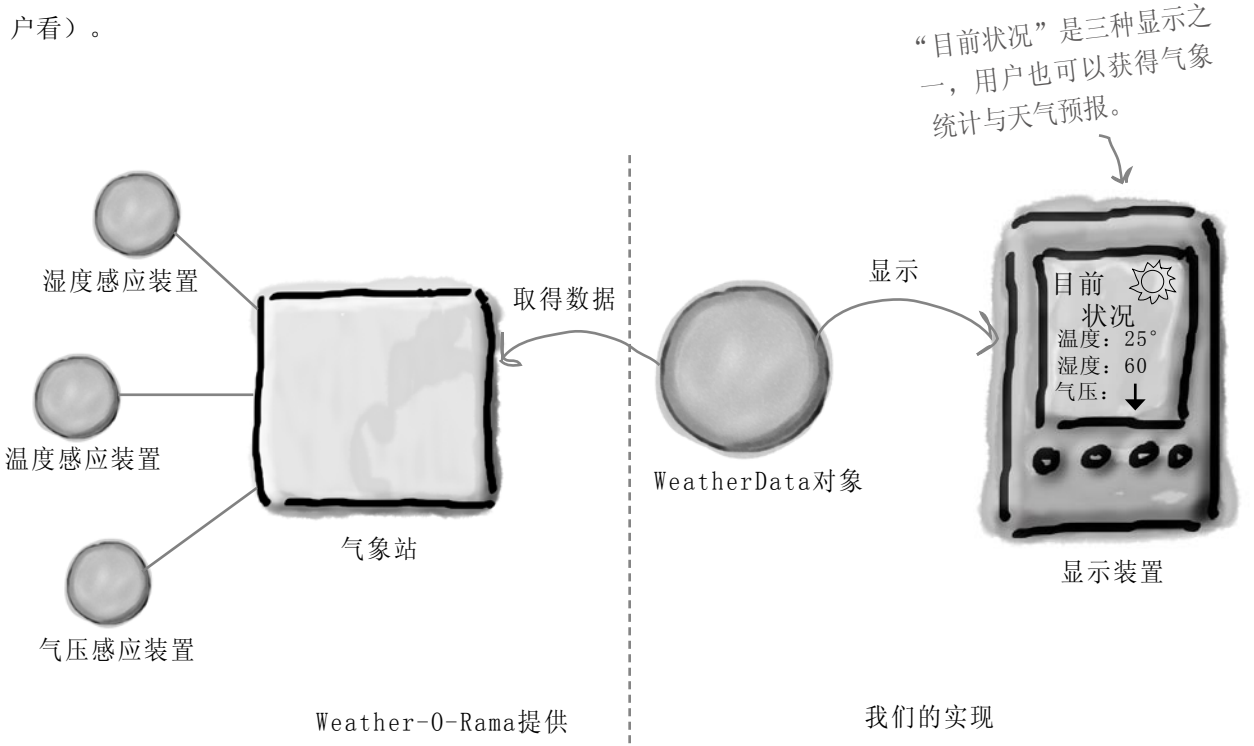
Johnny Hurricane

Johnny Hurricane——Weather-O-Rama气象站执行长

附注：我们正通宵整理WeatherData源文件给你们。

气象监测应用的概况

此系统中的三个部分是气象站（获取实际气象数据的物理装置）、WeatherData对象（追踪来自气象站的数据，并更新布告板）和布告板（显示目前天气状况给用户看）。

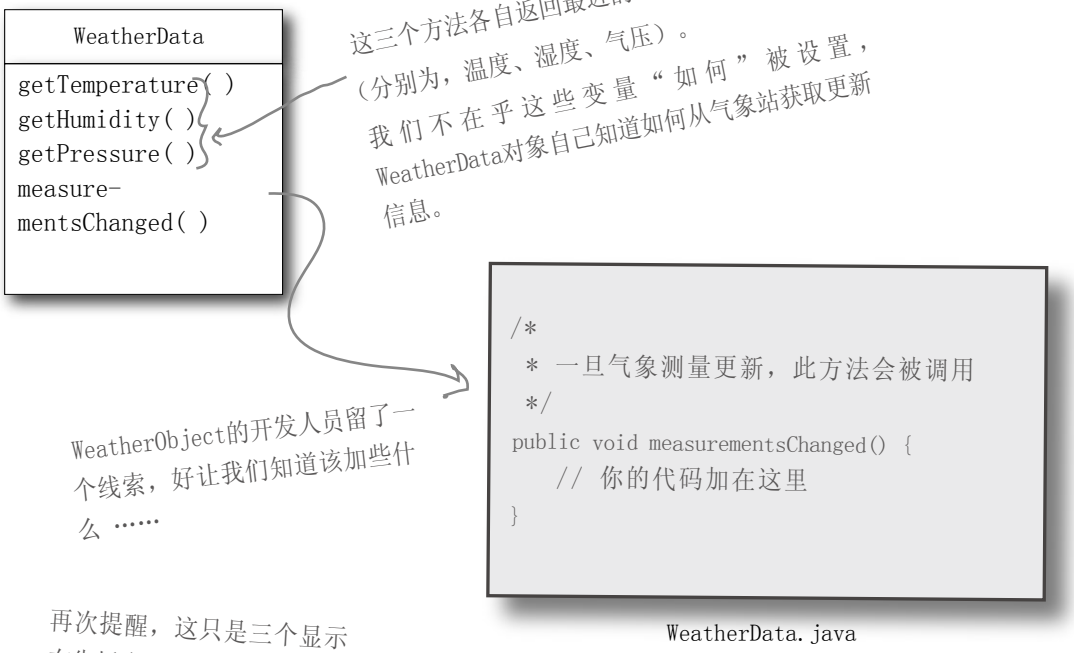


WeatherData对象知道如何跟物理气象站联系，以取得更新的数据。WeatherData对象会随即更新三个布告板的显示：目前状况（温度、湿度、气压）、气象统计和天气预报。

如果我们选择接受这个项目，我们的工作就是建立一个应用，利用WeatherData对象取得数据，并更新三个布告板：目前状况、气象统计和天气预报。

瞧一瞧刚送到的WeatherData类

如同他们所承诺的，隔天早上收到了WeatherData源文件，看了一下代码，一切都很直接：



再次提醒，这只是三个显示布告板中的一个。

显示装置

我们的工作是实现measurementsChanged()，好让它更新目前状况、气象统计、天气预报的显示布告板。

我们目前知道些什么？

Weather-O-Rama气象站的要求说明并不是很清楚，我们必须搞懂该做些什么。那么，我们目前知道些什么呢？

- WeatherData类具有getter方法，可以取得三个测量值：温度、湿度与气压。

```
getTemperature()
getHumidity()
getPressure()
```

- 当新的测量数据备妥时，measurementsChanged()方法就会被调用（我们不在乎此方法是如何被调用的，我们只在乎它被调用了）。

```
measurementsChanged()
```

- 我们需要实现三个使用天气数据的布告板：“目前状况”布告、“气象统计”布告、“天气预报”布告。一旦WeatherData有新的测量，这些布告必须马上更新。

第一号布告板

第二号布告板

第三号布告板

- 此系统必须可扩展，让其他开发人员建立定制的布告板，用户可以随心所欲地添加或删除任何布告板。目前初始的布告板有三类：“目前状况”布告、“气象统计”布告、“天气预报”布告。

将来的布告板

先看一个错误示范

这是第一个可能的实现：我们依照Weather-O-Rama气象站开发人员的暗示，在measurementsChanged()方法中添加我们的代码：

```
public class WeatherData {

    // 实例变量声明

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // 这里是其他WeatherData方法
}
```

调用 WeatherData 的三个getXxx()方法，以取得最近的测量值。这些getXxx()方法已经实现好了。

现在，更新布告板……

调用每个布告板更新显示，传入最新的测量。

在我们的第一个实现中，下列哪种说法正确？（多选）

- ❑ A. 我们是针对具体实现编程，而非针对接口。
- ❑ B. 对于每个新的布告板，我们都得修改代码。
- ❑ C. 我们无法在运行时动态地增加（或删除）布告板。
- ❑ D. 布告板没有实现一个共同的接口。
- ❑ E. 我们尚未封装改变的部分。
- ❑ F. 我们侵犯了WeatherData类的封装。

SWAG的定义:Scientific Wild A** Guess

我们的实现有什么不对？

回想第 1 章的概念和原则……

```
public void measurementsChanged( ) {

    float temp = getTemperature( );
    float humidity = getHumidity( );
    float pressure = getPressure( );

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}
```

改变的地方，需要封装起来。

针对具体实现编程，会导致我们以后在增加或删除布告板时必须修改程序。

至少，这里看起来像是一个统一的接口，布告板的方法名称都是update()，参数都是温度、湿度、气压。

我们现在就来看观察者模式，然后再回来看看如何将此模式应用到气象观测站。

认识观察者模式

我们看看报纸和杂志的订阅是怎么回事：

1. 报社的业务就是出版报纸。

2. 向某家报社订阅报纸，只要他们有新报纸出版，就会给你送来。只要你是他们的订户，你就会一直收到新报纸。

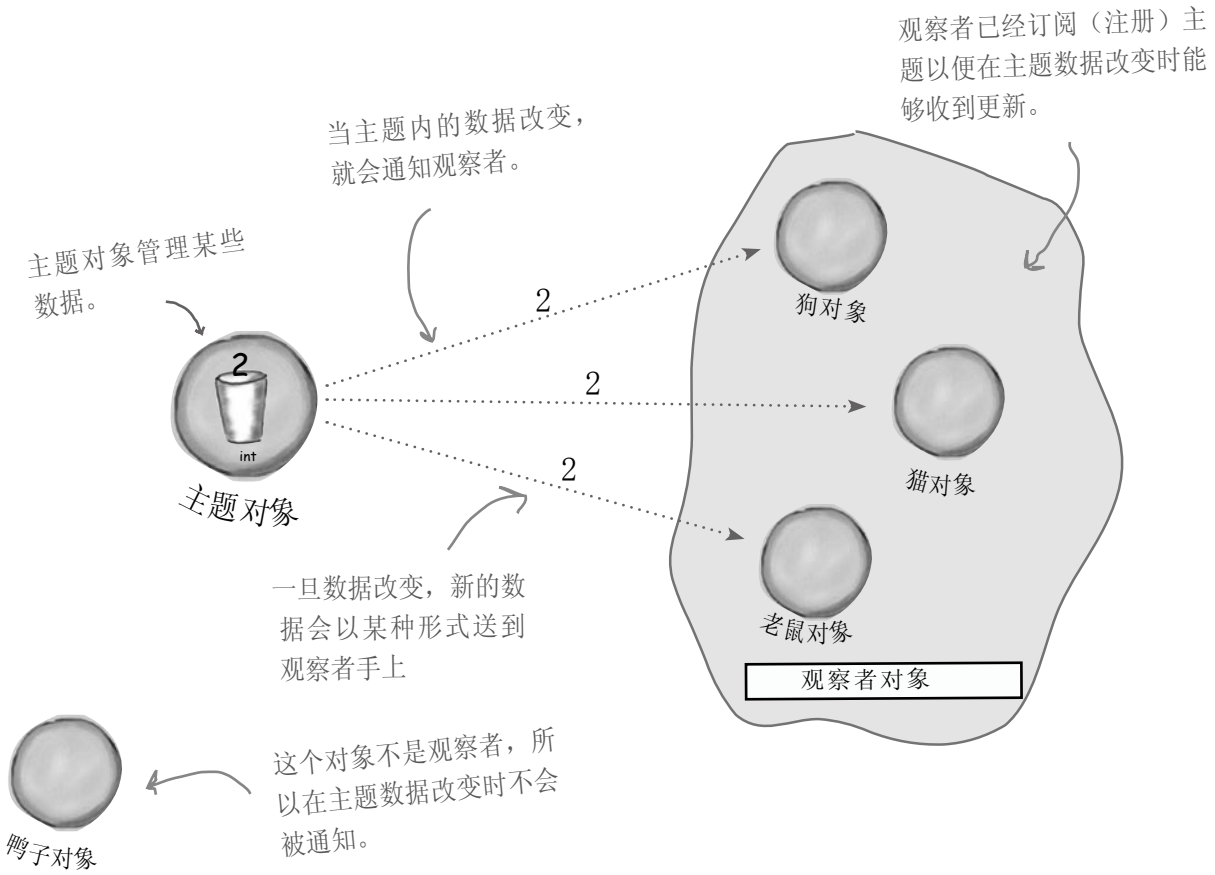
3. 当你不想再看报纸的时候，取消订阅，他们就不会再送新报纸来。

4. 只要报社还在运营，就会一直有人（或单位）向他们订阅报纸或取消订阅报纸。

出版者+订阅者=观察者模式

如果你了解报纸的订阅是怎么回事，其实就知道观察者模式是怎么回事，只是名称不太一样：出版者改称为“主题”（Subject），订阅者改称为“观察者”（Observer）。

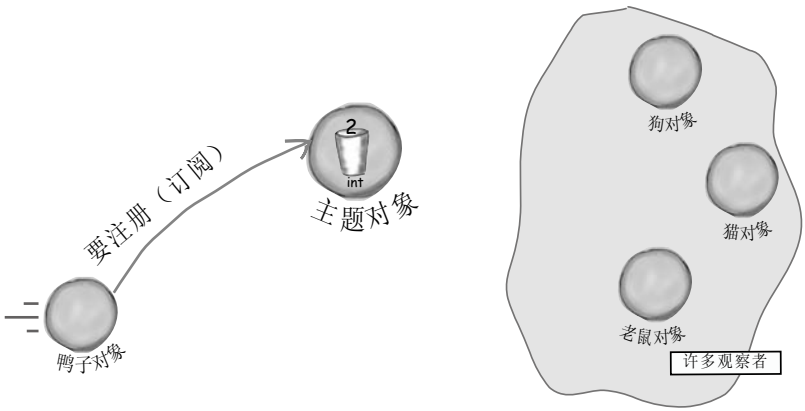
让我们来看得更仔细一点：



观察者模式的一天

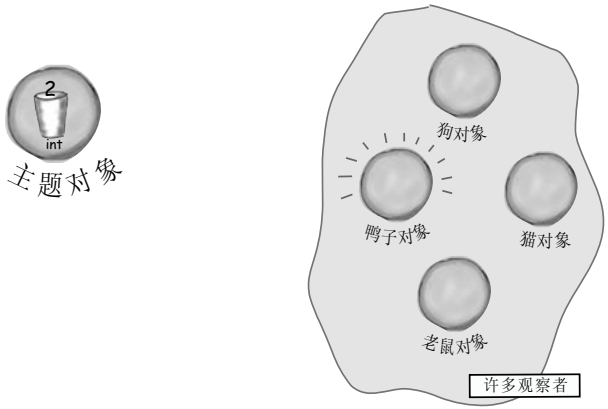
鸭子对象过来告诉主题，它想当一个观察者。

鸭子其实想说的是：我对你的数据改变感兴趣，一有变化请通知我



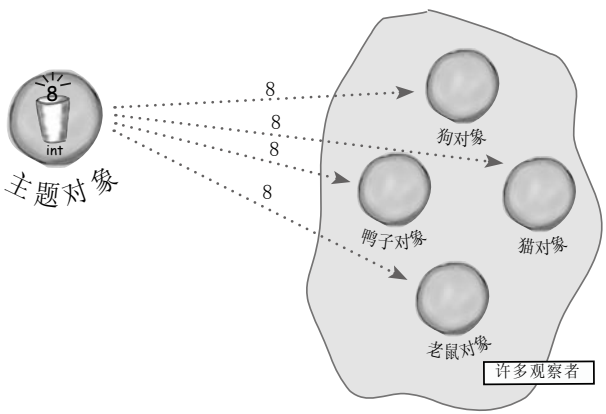
鸭子对象现在已经是正式的观察者了。

鸭子静候通知，等待参与这项伟大的事情。一旦接获通知，就会得到一个整数。



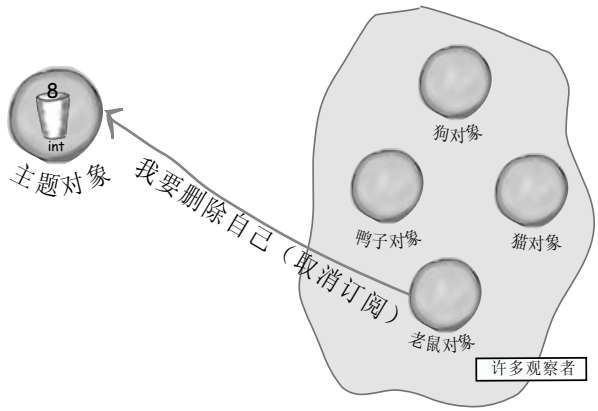
主题有了新的数据值！

现在鸭子和其他所有观察者都会收到通知：主题已经改变了。



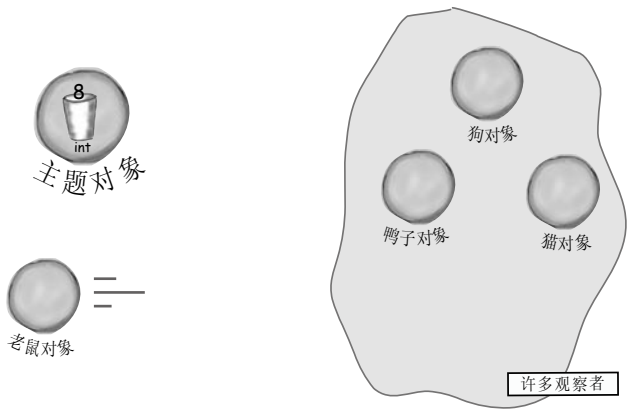
老鼠对象要求从观察者中把自己除名。

老鼠已经观察此主题太久，厌倦了，所以决定不再当个观察者。



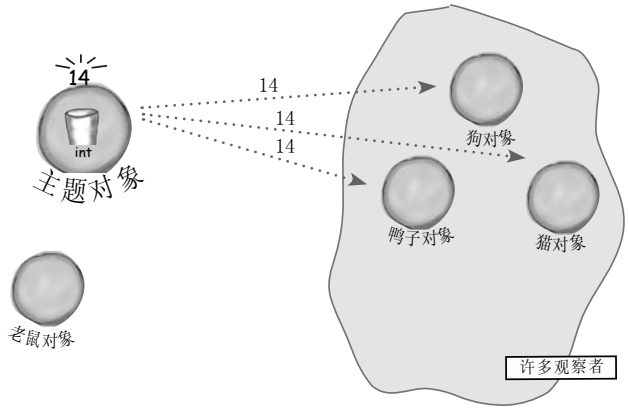
老鼠离开了！

主题知道老鼠的请求之后，把它从观察者中除名。



主题有一个新的整数

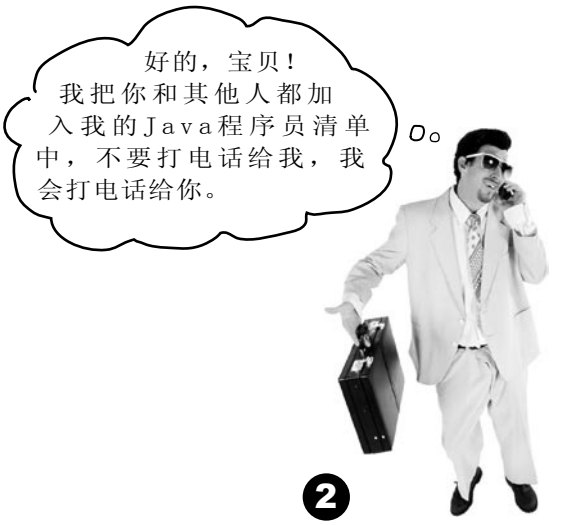
除了老鼠之外，每个观察者都会收到通知，因为它已经被除名了。嘘！不要告诉别人，老鼠其实心中暗暗地怀念这些整数，或许哪天又会再次注册，回来继续当观察者呢！



五分钟短剧：观察的主题

在今天的讽刺短剧中，有两个后泡沫时期的软件工程师，遇到一个真正的猎头……

一号软件开发人员

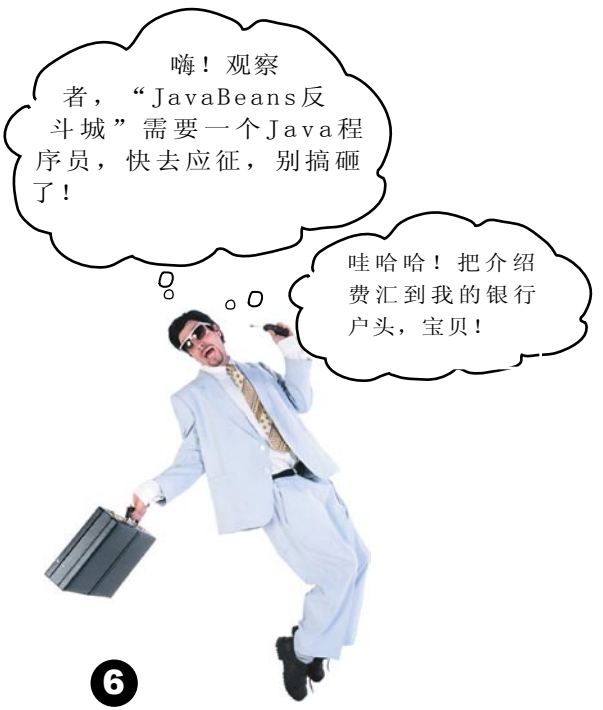


猎头/主题

二号软件开发人员

主题

5 其间，Ron和Jill继续过自己的日子，如果Java工作来了，他们会接到通知，毕竟，他们是观察者嘛！



6 主题

7 观察者

观察者

Jill自己找到工作了！

8 观察者

9 主题

两周后……

Jill热爱她的现状，她不再是观察者了。她还因为签约获得了一笔奖金，因为公司不用拨出一大笔钱给猎头。

但是，我们亲爱的Ron，又如何了？我们听说他设局把原来的猎头搞得毫无招架之力。他不只是一个观察者，也有了自己的求职者清单，只要付一笔钱给猎头，就可从其他求职者赚取更多钱。Ron既是一个主题，也是一个观察者，集两种角色于一身。

定义观察者模式

当你试图勾勒观察者模式时，可以利用报纸订阅服务，以及出版者和订阅者比拟这一切。

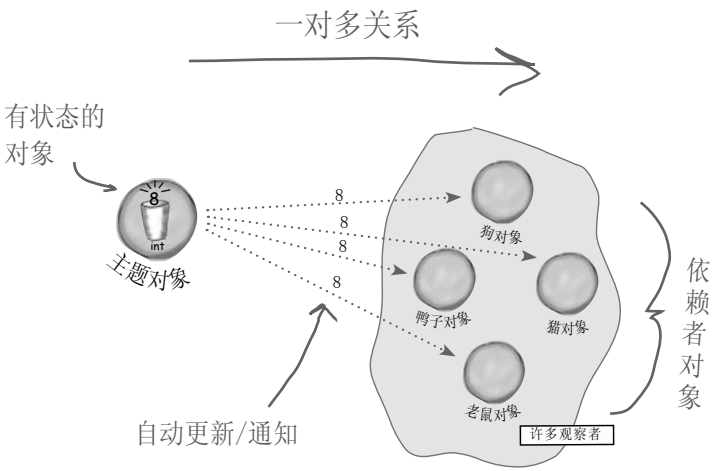
在真实的世界中，你通常会看到观察者模式被定义成：

> **观察者模式**定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

观察者模式定义了一系列对象之间的一对多关系。

当一个对象改变状态，其他依赖者都会收到通知。

让我们看看这个定义，并和之前的例子做个对照：

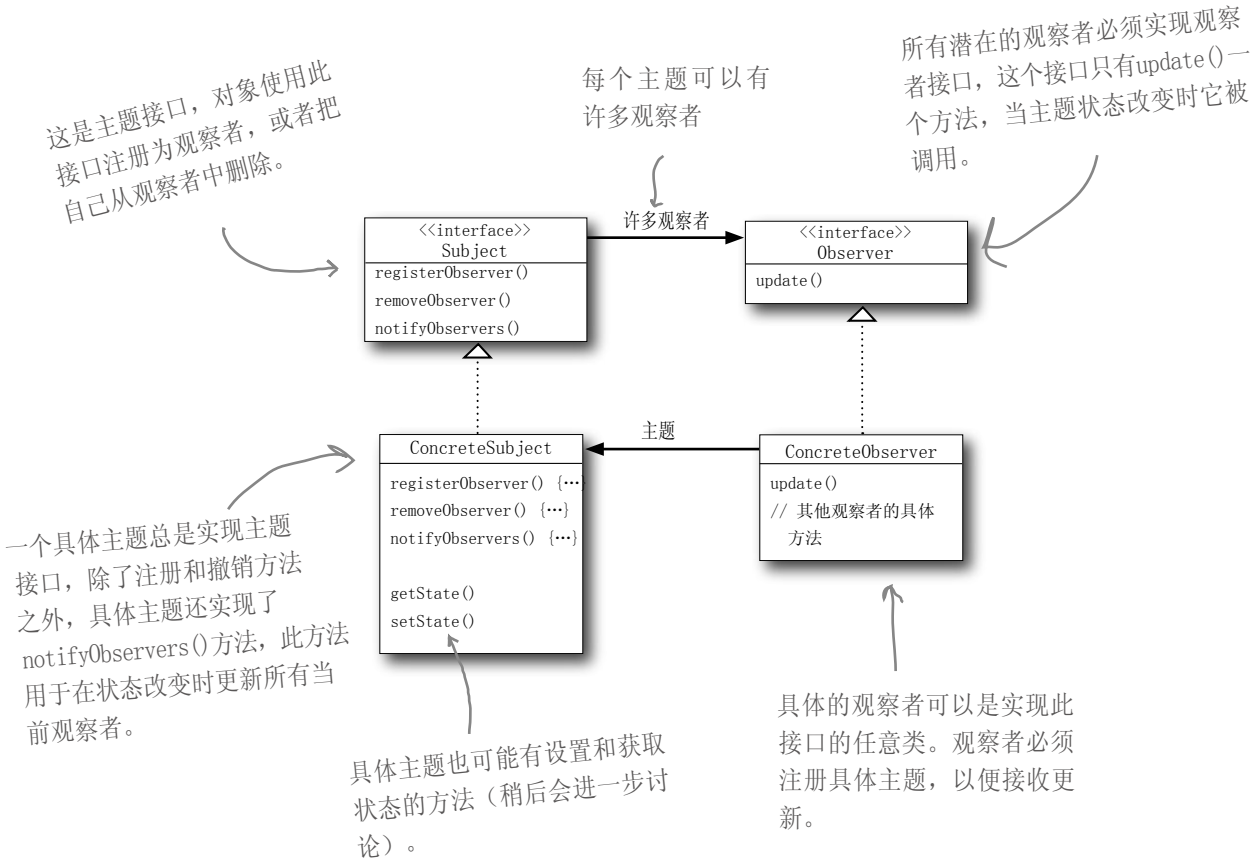


主题和观察者定义了一对多的关系。观察者依赖于此主题，只要主题状态一有变化，观察者就会被通知。根据通知的风格，观察者可能因此新值而更新。

稍后你会看到，实现观察者模式的方法不只一种，但是以包含Subject与Observer接口的类设计的做法最常见。

让我们快来看看吧……

定义观察者模式：类图



there are no Dumb Questions

问： 这和一对多的关系有何关联？

答： 利用观察者模式，主题是具有状态的对象，并且可以控制这些状态。也就是说，有“一个”具有状态的主题。另一方面，观察者使用这些状态，虽然这些状态并不属于他们。有许多的观察者，依赖主题来告诉他们状态何时改变了。这就产生一个关系：“一个”主题对“多个”观察者的关系。

问： 其间的依赖是如何产生的？

答： 因为主题是真正拥有数据的人，观察者是主题的依赖者，在数据变化时更新，这样比起让许多对象控制同一份数据来，可以得到更干净的的OO设计。

松耦合的威力

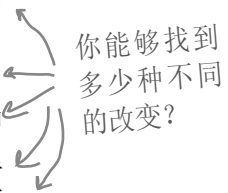
当两个对象之间松耦合，它们依然可以交互，但是不太清楚彼此的细节。

观察者模式提供了一种对象设计，让主题和观察者之间松耦合。

为什么呢？

关于观察者的一切，主题只知道观察者实现了某个接口（也就是Observer接口）。主题不需要知道观察者的具体类是谁、做了些什么或其他任何细节。

任何时候我们都可以增加新的观察者。因为主题唯一依赖的东西是一个实现Observer接口的对象列表，所以我们可以随时增加观察者。事实上，在运行时我们可以用新的观察者取代现有的观察者，主题不会受到任何影响。同样的，也可以在任何时候删除某些观察者。



有新类型的观察者出现时，主题的代码不需要修改。假如我们有个新的具体类需要当观察者，我们不需要为了兼容新类型而修改主题的代码，所有要做的就是在新的类里实现此观察者接口，然后注册为观察者即可。主题不在乎别的，它只会发送通知给所有实现了观察者接口的对象。

我们可以独立地复用主题或观察者。如果我们在其他地方需要使用主题或观察者，可以轻易地复用，因为二者并非紧耦合。

改变主题或观察者其中一方，并不会影响另一方。因为两者是松耦合的，所以只要他们之间的接口仍被遵守，我们就可以自由地改变他们。

设计原则

为了交互对象之间的松耦合设计而努力。

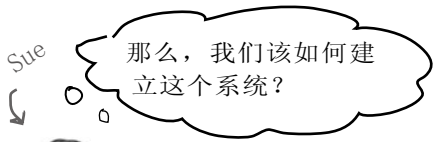
松耦合的设计之所以能让我们建立有弹性的OO系统，能够应对变化，是因为对象之间的互相依赖降到了最低。

在继续后面的内容之前，请试着画出实现气象站所需要的类，其中包括WeatherData类及布告板组件。确定你的图能够显示出各个部分如何结合起来，以及别的开发人员如何能够实现他自己的布告板组件。

如果你需要一点小帮助，请阅读下一页，你的队友正在讨论如何设计气象站。

办公室隔间对话

回到气象站项目，你的队友们已经开始全面思考这个问题了……



Mary：这个嘛！使用观察者模式啰！

Sue：是的……但是如何应用？

Mary：唔，我们再看一下定义好了：

观察者模式定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

Mary：当你思考这个定义时，你会发现很有道理。我们的WeatherData类正是此处所说的“一”，而我们的“多”正是使用天气观测的各种布告板。

Sue：没错。WeatherData对象的确是有状态，包括了温度、湿度、气压，而这些值都会改变。

Mary：对呀！而且，当这些观测值改变时，必须通知所有的布告板，好让它们各自做出处理。

Sue：好棒！我现在知道如何将观察者模式应用在气象站问题上了。

Mary：还有一些问题有待理清，我现在还不太了解它们的解决方法。

Sue：什么问题？

Mary：其中一个问题是，我们如何将气象观测值放到布告板上。

Sue：回头去看看观察者模式的图，如果我们把WeatherData对象当作主题，把布告板当作观察者，布告板为了取得信息，就必须先向WeatherData对象注册。对不对？

Mary：是的……一旦WeatherData知道有某个布告板的存在，就会适时地调用布告板的某个方法来告诉布告板观测值是多少。

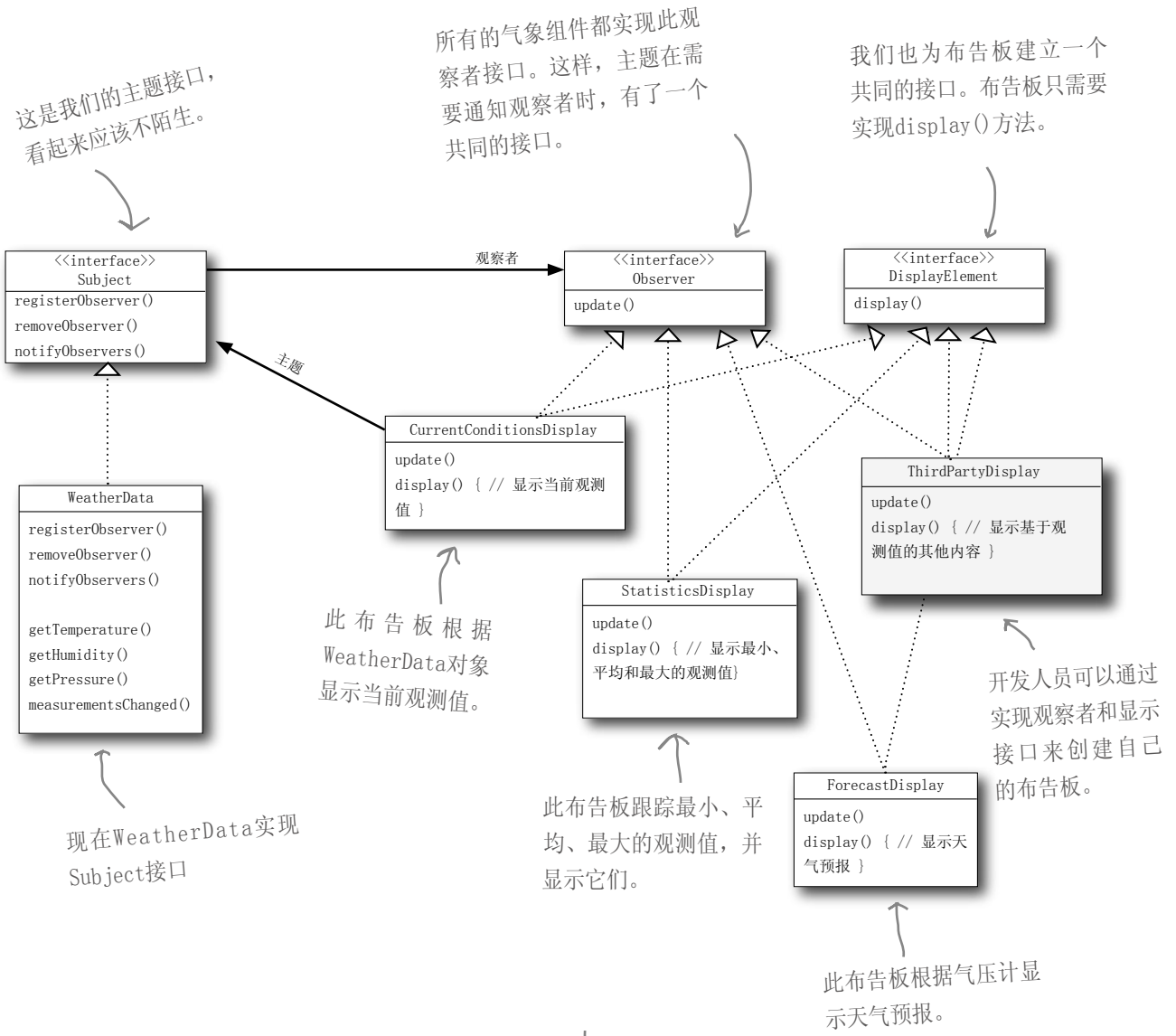
Sue：我们必须记得，每个布告板都有差异，这也就是为什么我们需要一个共同的接口的原因。尽管布告板的类都不一样，但是它们都应该实现相同的接口，好让WeatherData对象能够知道如何把观测值送给它们。

Mary：我懂你的意思。所以每个布告板都应该有一个大概名为update()的方法，以供WeatherData对象调用。

Sue：而这个update()方法应该在所有布告板都实现的共同接口里定义。

设计气象站

看看这个设计图，和你的设计图有何异同？



实现气象站

依照两页前Mary和Sue的讨论，以及上一页的类图，我们要开始实现这个系统了。稍后，你将会在本章看到Java为观察者模式提供了内置的支持，但是，我们暂时不用它，而是先自己动手。虽然，某些时候可以利用Java内置的支持，但是有许多时候，自己建立这一切会更具弹性（况且建立这一切并不是很麻烦）。所以，让我们从建立接口开始吧：

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

这两个方法都需要一个观察者作为变量，该观察者是用来注册或被删除的。

当主题状态改变时，这个方法会被调用，以通知所有的观察者。

```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

所有的观察者都必须实现update()方法，以实现观察者接口。在这里，我们按照Mary和Sue的想法把观测值传入观察者中。

当气象观测值改变时，主题会把这些状态值当作方法的参数，传送给观察者。

```
public interface DisplayElement {
    public void display();
}
```

DisplayElement接口只包含了一个方法，也就是display()。当布告板需要显示时，调用此方法。

Mary和Sue认为：把观测值直接传入观察者中是更新状态的最直接的方法。你认为这样的做法明智吗？暗示：这些观测值的种类和个数在未来有可能改变吗？如果以后会改变，这些变化是否被很好地封装？或者是需要修改许多代码才能办到？

关于将更新的状态传送给观察者，你能否想到更好的方法解决此问题？

别担心，在我们完成第一次实现后，我们会再回来探讨这个设计决策。

在WeatherData中实现主题接口

还记得我们在本章一开始的地方就试图实现WeatherData类吗？你可以去回顾一下。现在，我们要用观察者模式实现……

提醒你：为了节省篇幅，我们在代码中没有列出import和package语句。你可以到wickedlysmart网站找到完整的源代码，URL在本书的第xxxv页。

```
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    // WeatherData的其他方法
}
```

这部分是Subject接口的实现。

WeatherData现在实现了Subject接口。

我们加上一个ArrayList来纪录观察者，此ArrayList是在构造器中建立的。

当注册观察者时，我们只要把它加到ArrayList的后面即可。

同样地，当观察者想取消注册，我们把它从ArrayList中删除即可。

有趣的地方来了！在这里，我们把状态告诉每一个观察者。因为观察者都实现了update()，所以我们知道如何通知它们。

当从气象站得到更新观测值时，我们通知观察者。

我们想要每本书随书赠送一个小型气象站，但是出版社不肯。所以，和从装置中读取实际的气象数据相比，我们宁愿利用这个方法来测试布告板。或者，为了好玩，你也可以写代码从网站上抓取观测值。

现在，我们来建立布告板吧！

我们已经把WeatherData类写出来了，现在轮到布告板了。Weather-O-Rama气象站订购了三个布告板：目前状况布告板、统计布告板和预测布告板。我们先看看目前状况布告板。一旦你熟悉此布告板之后，可以在本书的代码目录中，找到另外两个布告板的源代码，你会觉得这些布告板都很类似。

此布告板实现了Observer接口，所以可以从WeatherData对象中获得改变。

它也实现了DisplayElement接口，因为我们的API规定所有的布告板都必须实现此接口。

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

构造器需要 weatherData对象（也就是主题）作为注册之用。

当update()被调用时，我们把温度和湿度保存起来，然后调用display()。

display()方法就只是把最近的温度和湿度显示出来。

there are no Dumb Questions

问： update()是最适合调用display()的地方吗？

答： 在这个简单的例子中，当值变化的时候调用display()，是很合理的。然而，你是对的，的确是有很多更好的方法来设计显示数据的方式。当我们谈到MVC（Model-View-Controller）模式时会再作说明。

问： 为什么要保存对Subject的引用呢？构造完后似乎用不着了呀？

答： 的确如此，但是以后我们可能想要取消注册，如果已经有了对Subject的引用会比较方便。

启动气象站

❶ 先建立一个测试程序

气象站已经完成得差不多了，我们还需要一些代码将这一切连接起来。这是我们的第一次尝试，本书中稍后我们会再回来确定每个组件都能通过配置文件来达到容易“插拔”。现在开始测试吧：

```
public class WeatherStation {

    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay =
            new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

首先，建立一个WeatherData对象。

建立三个布告板，并把WeatherData对象传给它们。

模拟新的气象测量。

如果你还不想下载完整的代码，可以将这两行注释掉，就能顺利执行了。

❷ 运行程序，让观察者模式表演魔术。

```
File Edit Window Help StormyWeather
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
%
```

Sharpen your pencil

Johnny Hurricane（Weather-O-Rama气象站的CEO）刚刚来电告知，他们还需要酷热指数（HeatIndex）布告板，这是不可或缺的。细节如下：

酷热指数是一个结合温度和湿度的指数，用来显示人的温度感受。可以利用温度T和相对湿度RH套用下面的公式来计算酷热指数：

$$\begin{aligned} heatindex = & \\ & 16.923 + 1.85212 * 10^{-1} * T + 5.37941 * RH - 1.00254 * 10^{-1} * T \\ & * RH + 9.41695 * 10^{-3} * T^2 + 7.28898 * 10^{-3} * RH^2 + 3.45372 * 10^{-4} \\ & * T^2 * RH - 8.14971 * 10^{-4} * T * RH^2 + 1.02102 * 10^{-5} * T^2 * RH^2 - \\ & 3.8646 * 10^{-5} * T^3 + 2.91583 * 10^{-5} * RH^3 + 1.42721 * 10^{-6} * T^3 * RH \\ & + 1.97483 * 10^{-7} * T * RH^3 - 2.18429 * 10^{-8} * T^3 * RH^2 + 8.43296 * \\ & 10^{-10} * T^2 * RH^3 - 4.81975 * 10^{-11} * T^3 * RH^3 \end{aligned}$$

开始练习打字吧！

开玩笑的啦！别担心，你不需要亲自输入此公式，只要建立你自己的HeatIndexDisplay.java文件并把公式从heatindex.txt文件中拷贝进来就可以了。

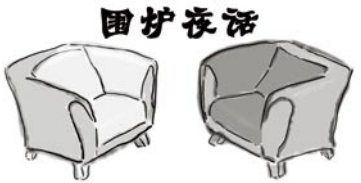
heatindex.txt文件可从wickedlysmart.com取得

这个公式是怎么回事？你可以参考《Head First气象学》，或者问问国家气象局的员工（或用Google搜索）。

当你完成后，输出结果应如下所示：

输出结果有改变的地方在这里。

```
File Edit Window Help OverdaRainbow
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Heat index is 82.95535
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Heat index is 86.90124
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
Heat index is 83.64967
%
```



今夜话题：主题和观察者就使观察者获得状态信息的正确方法发生了争吵。

主题

我很高兴，我们终于有机会面对面聊天了。

唉呀！我把该做的事都做到了，不是吗？我总是会通知你们发生什么事了……我虽然不知道你们是谁，但这不意味着我不在乎你们。况且，我知道关于你们的一件重要的事：你们实现了Observer接口。

是吗？说来听听！

拜托，我必须主动送出我的状态和通知给大家，好让你们这些懒惰的观察者知道发生什么事了。

嗯……这样或许也行，只是我必须因此门户大开，让你们全都可以进来取得你们需要的状态，这样太危险了。我不能让你们进来里面大肆挖掘我的各种数据。

观察者

是这样吗？我以为你根本不在乎我们这群观察者呢。

是呀，但这只是关于我的一小部分罢了！无论如何，我对你更了解……

嗯！你总是将你的状态传给我们，所以我们可以知道你内部的情况。有时候，这很烦人的……

咳！等等。我说主题先生，首先，我们并不懒，在你那些“很重要”通知的空档中，我们还有别的事要做。另外，为何由你主动送数据过来，而不是让我们主动去向你索取数据？

主题

是的，我可以让你们“拉”走我的状态，但是你不觉得这样对你们反而不方便吗？如果每次想要数据时都来找我，你可能要调用很多次才能收集齐全你所要的状态。这就是为什么我更喜欢“推”的原因，你们可以在一次通知中一口气得到所有东西。

是的。两种做法都有各自的优点。我注意到Java内置的Observer 模式两种做法都支持。

太好了，或许我会看到一个“拉”的好例子，因而改变我的想法。

观察者

你何不提供一些公开的getter方法，让我们“拉”走我们需要的状态？

死鸭子嘴硬！观察者种类这么多，你不可能事先料到我们每个人的需求，还是让我们直接去取得我们需要的状态比较恰当，这样一来，如果我们有人只需要一点点数据，就不会被强迫收到一堆数据。这么做同时也可以在以后比较容易修改。比方说，哪一天你决定扩展功能，新增更多的状态，如果采用我建议的方式，你就不用修改和更新对每位观察者的调用，只需改变自己来允许更多的getter方法来取得新增的状态。

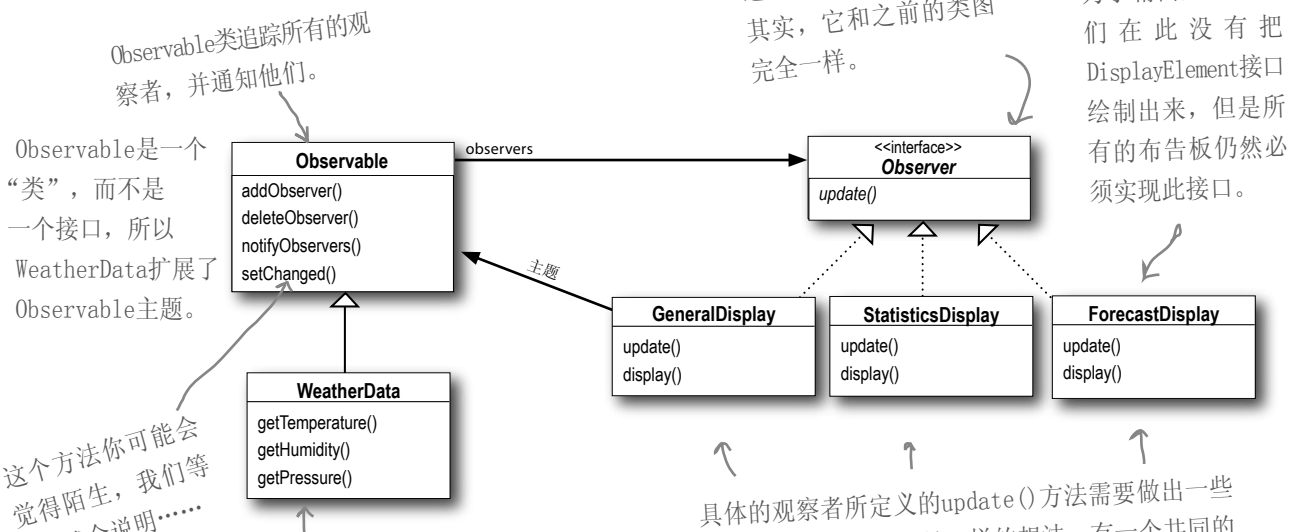
真的吗？我们得去瞧瞧……

什么？我们会有意见相同的一天？不会吧！

使用Java内置的观察者模式

到目前为止，我们已经从无到有地完成了观察者模式，但是，Java API有内置的观察者模式。java.util包（package）内包含最基本的Observer接口与Observable类，这和我们的Subject接口与Observer接口很相似。Observer接口与Observable类使用上更方便，因为许多功能都已经事先准备好了。你甚至可以使用推（push）或拉（pull）的方式传送数据，稍后就会看到这样的例子。

为了更了解java.uitl.Observer和java.util.Observable，看看下面的图，这是修改后的气象站OO设计。



Java内置的观察者模式如何运作

Java内置的观察者模式运作方式，和我们在气象站中的实现类似，但有一些小差异。最明显的差异是WeatherData（也就是我们的主题）现在扩展自Observable类，并继承到一些增加、删除、通知观察者的方法（以及其他的方法）。Java版本的用法如下：

如何把对象变成观察者……

如同以前一样，实现观察者接口（java.uitl.Observer），然后调用任何Observable对象的addObserver()方法。不想再当观察者时，调用deleteObserver()方法就可以了。

可观察者要如何送出通知……

首先，你需要利用扩展java.util.Observable接口产生“可观察者”类，然后，需要两个步骤：

1. 先调用setChanged()方法，标记状态已经改变的事实。
2. 然后调用两种notifyObservers()方法中的一个：

```
notifyObservers()   或   notifyObservers(Object arg)
```

当通知时，此版本可以传送任何的数据对象给每一个观察者。

观察者如何接收通知……

同以前一样，观察者实现了更新的方法，但是方法的签名不太一样：

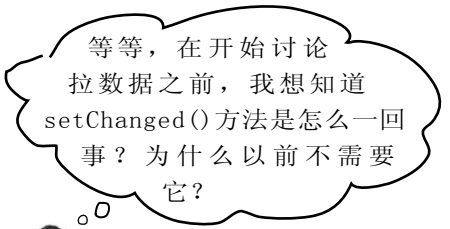
```
update(Observable o, Object arg)
```

data object

主题本身当作第一个变量，好让观察者知道是哪个主题通知它的。

这正是传入notifyObservers()的数据对象。如果没有说明则为空。

如果你想“推”（push）数据给观察者，你可以把数据当作数据对象传送给notifyObservers(arg)方法。否则，观察者就必须从可观察者对象中“拉”（pull）数据。如何拉数据？我们再做一遍气象站，你很快就会看到。



setChanged()方法用来标记状态已经改变的事实，好让notifyObservers()知道当它被调用时应该更新观察者。如果调用notifyObservers()之前没有先调用setChanged()，观察者就“不会”被通知。让我们看看Observable 内部，以了解这一切：

Observable类的伪代码

幕后花絮

```
setChanged() {
   changed = true
}

notifyObservers(Object arg) {
   if (changed) {
      for every observer on the list {
          call update (this, arg)
      }
      changed = false
   }
}

notifyObservers() {
   notifyObservers(null)
}
```

setChanged()方法把changed标志设为true。

notifyObservers() 只会在changed标为“true”时通知观察者。

在通知观察者之后，把changed标志设回false。

这样做有其必要性。setChanged()方法可以让你在更新观察者时，有更多的弹性，你可以更适当地通知观察者。比方说，如果没有setChanged()方法，我们的气象站测量是如此敏锐，以致于温度计读数每十分之一度就会更新，这会造成WeatherData对象持续不断地通知观察者，我们并不希望看到这样的事情发生。如果我们希望半度以上才更新，就可以在温度差距到达半度时，调用setChanged()，进行有效的更新。

你也许不会经常用到此功能，但是把这样的功能准备好，当需要时马上就可以使用。总之，你需要调用setChanged()，以便通知开始运转。如果此功能在某些地方对你有帮助，你可能也需要clearChanged()方法，将changed状态设置回false。另外也有一个hasChanged()方法，告诉你changed标志的当前状态。

利用内置的支持重做气象站

首先，把WeatherData改成使用java.util.Observable

❶ 记得要导入（import）正确的Observer/Observable。

❷ 我们现在正在继承Observable。

❸ 我们不再需要追踪观察者了，也不需要管理注册与删除（让超类代劳即可）。所以我们把注册、添加、通知的相关代码删除。

```
import java.util.Observable;
import java.util.Observer;

public class WeatherData extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() { }

    public void measurementsChanged() {
        setChanged();
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

    public float getPressure() {
        return pressure;
    }
}
```

❹ 我们的构造器不再需要为了记住观察者们而建立数据结构了。

★ 注意：我们没有调用notifyObservers()传送数据对象，这表示我们采用的做法是“拉”。

❺ 在调用notifuObservers()之前，要先调用setChanged()来指示状态已经改变。

❻ 这些并不是新方法，只是因为我们要使用“拉”的做法，所以才提醒你有这些方法。察者会利用这些方法取得WeatherData对象的状态。

现在，让我们重做CurrentConditionsDisplay

❶ 再说一遍，记得要导入（import）正确的Observer/Observable。

❷ 我们现在正在实现java.util.Observer接口。

```
import java.util.Observable;
import java.util.Observer;

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    Observable observable;
    private float temperature;
    private float humidity;

    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }

    public void update(Observable obs, Object arg) {
        if (obs instanceof WeatherData) {
            WeatherData weatherData = (WeatherData)obs;
            this.temperature = weatherData.getTemperature();
            this.humidity = weatherData.getHumidity();
            display();
        }
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

❸ 现在构造器需要一Observable当参数，并将CurrentCondi-tionsDisplay对象登记成为观察者。

❹ 改变update()方法，增加Observable和数据对象作为参数。

❺ 在 update()中，先确定可观察者属于WeatherData类型，然后利用 getter方法获取温度和湿度测量值，最后调用display()。

练习

代码帖

ForecastDisplay类的代码小纸片在冰箱上被弄乱了。你能够重新排列它们，好恢复原来的样子吗？有些大括号掉到地上了，因为太小捡起来不易，所以如果你觉得需要大括号时，可以自行加上。

```
public ForecastDisplay(Observable
observable) {
```

```
display();
```

```
observable.addObserver(this);
```

```
if (observable instanceof WeatherData) {
```

```
public class ForecastDisplay implements
Observer, DisplayElement {
```

```
public void display() {
    // 这里显示代码
}
```

```
lastPressure = currentPressure;
currentPressure = weatherData.getPressure();
```

```
private float currentPressure = 29.92f;
private float lastPressure;
```

```
WeatherData weatherData =
(WeatherData)observable;
```

```
public void update(Observable observable,
    Object arg) {
```

```
import java.util.Observable;
import java.util.Observer;
```

运行新的代码

让我们运行新的代码，以确定它是对的……

```
File  Edit   Window  Help  TryTihisAtHome
%java WeatherStation
Forecast: Improving weather on the way!
Avg/Max/Min temperature = 80.0/80.0/80.0
Current conditions: 80.0F degrees and 65.0% humidity
Forecast: Watch out for cooler, rainy weather
Avg/Max/Min temperature = 81.0/82.0/80.0
Current conditions: 82.0F degrees and 70.0% humidity
Forecast: More of the same
Avg/Max/Min temperature = 80.0/82.0/78.0
Current conditions: 78.0F degrees and 90.0% humidity
%
```

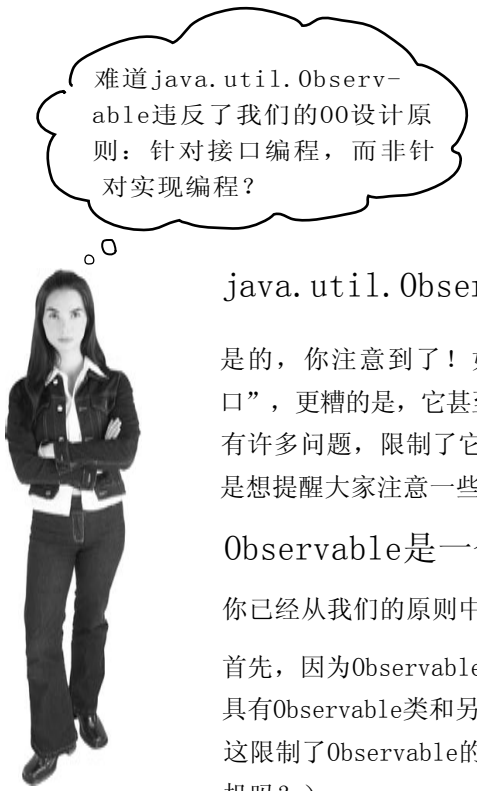
嗯！你注意到差别了吗？再看一次……

你会看到相同的计算结果，但是奇怪的地方在于，文字输出的次序不一样。怎么会这样呢？在继续之前，请花一分钟的时间思考……

不要依赖于观察者被通知的次序

java.uitl.Observable实现了它的notifyObservers()方法，这导致了通知观察者的次序不同于我们先前的次序。谁也没有错，只是双方选择不同的方式实现罢了。

但是可以确定的是，如果我们的代码依赖这样的次序，就是错的。为什么呢？因为一旦观察者/可观察者的实现有所改变，通知次序就会改变，很可能就会产生错误的结果。这绝对不是我们所认为的松耦合。



java.util.Observable的黑暗面

是的，你注意到了！如同你所发现的，可观察者是一个“类”而不是一个“接口”，更糟的是，它甚至没有实现一个接口。不幸的是，java.util.Observable的实现有许多问题，限制了它的使用和复用。这并不是说它没有提供有用的功能，我们只是想提醒大家注意一些事实。

Observable是一个类

你已经从我们的原则中得知这不是一件好事，但是，这到底会造成什么问题呢？

首先，因为Observable是一个“类”，你必须设计一个类继承它。如果某类想同时具有Observable类和另一个超类的行为，就会陷入两难，毕竟Java不支持多重继承。这限制了Observable的复用潜力（而增加复用潜力不正是我们使用模式最原始的动机吗？）。

再者，因为没有Observable接口，所以你无法建立自己的实现，和Java内置的ObserverAPI搭配使用，也无法将java.util的实现换成另一套做法的实现（比方说，

Observable将关键的方法保护起来

如果你看看ObservableAPI，你会发现setChanged()方法被保护起来了（被定义成protected）。那又怎么样呢？这意味着：除非你继承自Observable，否则你无法创建Observable实例并组合到你自己的对象中来。这个设计违反了第二个设计原则：“多用组合，少用继承”。

做什么呢？

如果你能够扩展java.util.Observable，那么Observable“可能”可以符合你的需求。否则，你可能需要像本章开头的做法那样自己实现这一整套观察者模式。不管用哪一种方法，反正你都已经熟悉观察者模式了，应该都能善用它们。

在JDK中，还有哪些地方可以找到观察者模式

在JDK中，并非只有在java.util中才能找到观察者模式，其实在JavaBeans和Swing中，也都实现了观察者模式。现在，你已经具备足够的能力来自行探索这些API，但是我们还是在此稍微提一个简单的Swing例子，让你感受一下其中的乐趣。

如果你对JavaBeans里的观察者模式感到好奇，可以查一下PropertyChangeListener接口。

背景介绍……

让我们看看一个简单的Swing API：JButton。如果你观察一下JButton的超类AbstractButton，会看到许多增加与删除倾听者（listener）的方法，这些方法可以让观察者感应到Swing组件的不同类型事件。比方说：ActionListener让你“倾听”可能发生在按钮上的动作，例如按下按钮。你可以在Swing API中找到许多不同类型的倾听者。

一个小的、改变生活的程序

我们的程序很简单，你有一个按钮，上面写着“Should I do it?”（我该做吗？）。当你按下按钮，倾听者（观察者）必须回答此问题。我们实现了两个倾听者，一个是天使（AngelListener），一个是恶魔（DevilListener）。程序的行为如下：

这是我们一个很炫的接口。

这是点按钮后所得到的输出。

```
File  Edit  Window  Help  HeMadeMeDoIt
%java SwingObserverExample
Come on, do it!
Don't do it, you might regret it!
%
```

恶魔的答复

天使的答复

代码是这样的……

这个改变生活的程序需要的代码很短。我们只需要建立一个JButton对象，把它加到JFrame，然后设置好倾听者就行了。我们打算用内部类（inner class）作为倾听者类（这样的技巧在Swing中很常见）。如果你对内部类或Swing不熟悉，可以读一读《Head First Java》中的并于“获得GUI”的章节。

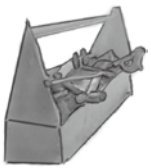
简单的Swing应用：建立一个JFrame，然后放上一个按钮。

```
public class SwingObserverExample {
    JFrame frame;

    public static void main(String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }

    public void go() {
        frame = new JFrame();

        JButton button = new JButton("Should I do it?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());
        frame.getContentPane().add(BorderLayout.CENTER, button);
        // 在这里设置frame属性
    }

    class AngelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Don't do it, you might regret it!");
        }
    }

    class DevilListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Come on, do it!");
        }
    }
}
```

制造出两个倾听者（观察者），一个天使，一个恶魔。

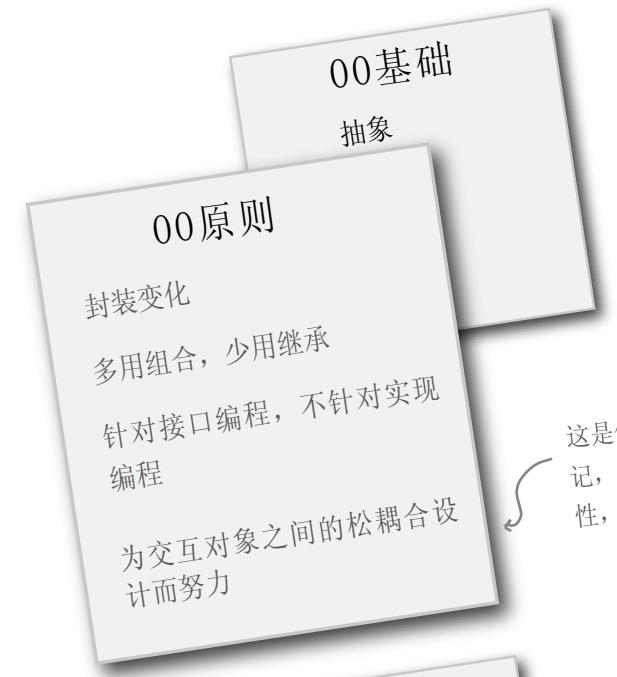
这是观察者的类定义，定义成内部类（你也可以不这么做）。

当主题（JButton）的状态改变时，在本例中，不是调用update()，而是调用actionPerformed()。



设计箱内的工具

欢迎来到第2章的结尾，你的对象工具箱内又多了一些东西……



要点

- 观察者模式定义了对象之间一对多的关系。
- 主题（也就是可观察者）用一个共同的接口来更新观察者
- 观察者和可观察者之间用松耦合方式结合（loosecoupling），可观察者不知道观察者的细节，只知道观察者实现了观察者接口。
- 使用此模式时，你可从被观察者处推（push）或拉（pull）数据（然而，推的方式被认为更“正确”）。
- 有多个观察者时，不可以依赖特定的通知次序。
- Java有多种观察者模式的实现，包括了通用的java.util.Observable。
- 要注意java.util.Observable实现上所带来的一些问题。
- 如果有必要的话，可以实现自己的Observable，这并不难，不要害怕。
- Swing大量使用观察者模式，许多GUI框架也是如此。
- 此模式也被应用在许多地方，例如：JavaBeans、RMI。

练习

挑战设计原则

对于每一个设计原则，请描述观察者模式如何遵循此原则。

设计原则

找出程序中会变化的方面，然后将其和固定不变的方面相分离。

设计原则

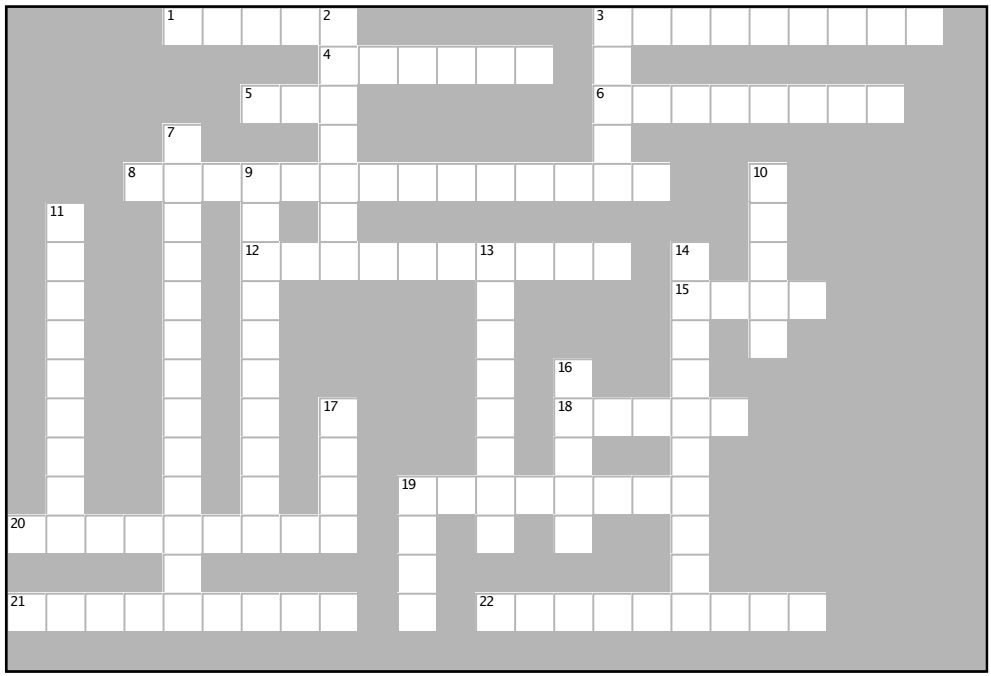
针对接口编程，不针对实现编程

设计原则

多用组合，少用继承

这一个比较难回答。给一点暗示：想想看观察者和主题是如何搭配工作的。

再次为你的右脑找些事情做吧！
这次所有的词都来自第2章。



横排提示：

1. Observable is a _____ not an interface
3. Devil and Angel are _____ to the button
4. Implement this method to get notified
5. Jill got one of her own
6. CurrentConditionsDisplay implements this interface
8. How to get yourself off the Observer list
12. You forgot this if you're not getting notified when you think you should be
15. One Subject likes to talk to _____ observers
18. Don't count on this for notification
19. Temperature, humidity and _____
20. Observers are _____ on the Subject
21. Program to an _____ not an implementation
22. A Subject is similar to a _____

竖排提示：

2. Ron was both an Observer and a _____
3. You want to keep your coupling _____
7. He says you should go for it
9. _____ can manage your observers for you
10. Java framework with lots of Observers
11. Weather-O-Rama's CEO named after this kind of storm
13. Observers like to be _____ when something new happens
14. The WeatherData class _____ the Subject interface
16. He didn't want any more ints, so he removed himself
17. CEO almost forgot the _____ index display
19. Subject initially wanted to ____ all the data to Observer

习题解答

Sharpen your pencil

在我们的第一个实现中，下列哪种说法正确？（多选）

- [x] A 我们是针对具体实现编程，而非针对接口。
- [x] B. 对于每个新的布告板，我们都得修改代码。
- [x] C. 我们无法在运行时动态地增加或删除布告板。
- [ ] D. 布告板没有实现一个共同的接口。
- [x] E. 我们尚未封装改变的部分。
- [ ] F. 我们侵犯了WeatherData类的封装。

挑战设计原则

设计原则

找出程序中会变化的方面，然后将其和固定不变的方面相分离。

在观察者模式中，会改变的是主题的状态，以及观察者的数目和类型。用这个模式，你可以改变依赖于主题状态的对象，却不必改变主题。这就叫提前规划！

设计原则

针对接口编程，不针对实现编程。

主题与观察者都使用接口：观察者利用主题的接口向主题注册，而主题利用观察者接口通知观察者。这样可以让两者之间运作正常，又同时具有松耦合的优点。

设计原则

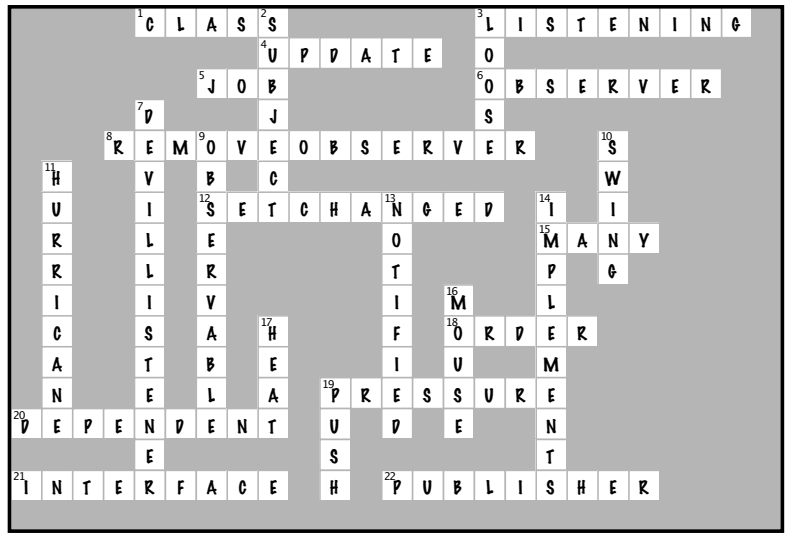
多用组合，少用继承。

观察者模式利用“组合”将许多观察者组合进主题中。对象之间的这种关系不是通过继承产生的，而是在运行时利用组合的方式而产生的。

代码帖

习题解答

```
import java.util.Observable;
import java.util.Observer;

public class ForecastDisplay implements
Observer, DisplayElement {

    private float currentPressure = 29.92f;
    private float lastPressure;

    public ForecastDisplay(Observable
    observable) {
        WeatherData weatherData =
        (WeatherData)observable;
        observable.addObserver(this);
    }
    public void update(Observable observable,
        Object arg) {
        if (observable instanceof WeatherData) {
            lastPressure = currentPressure;
            currentPressure = weatherData.getPressure();
            display();
        }
    }
    public void display() {
        // 在这里显示代码
    }
}
```



3 装饰者模式

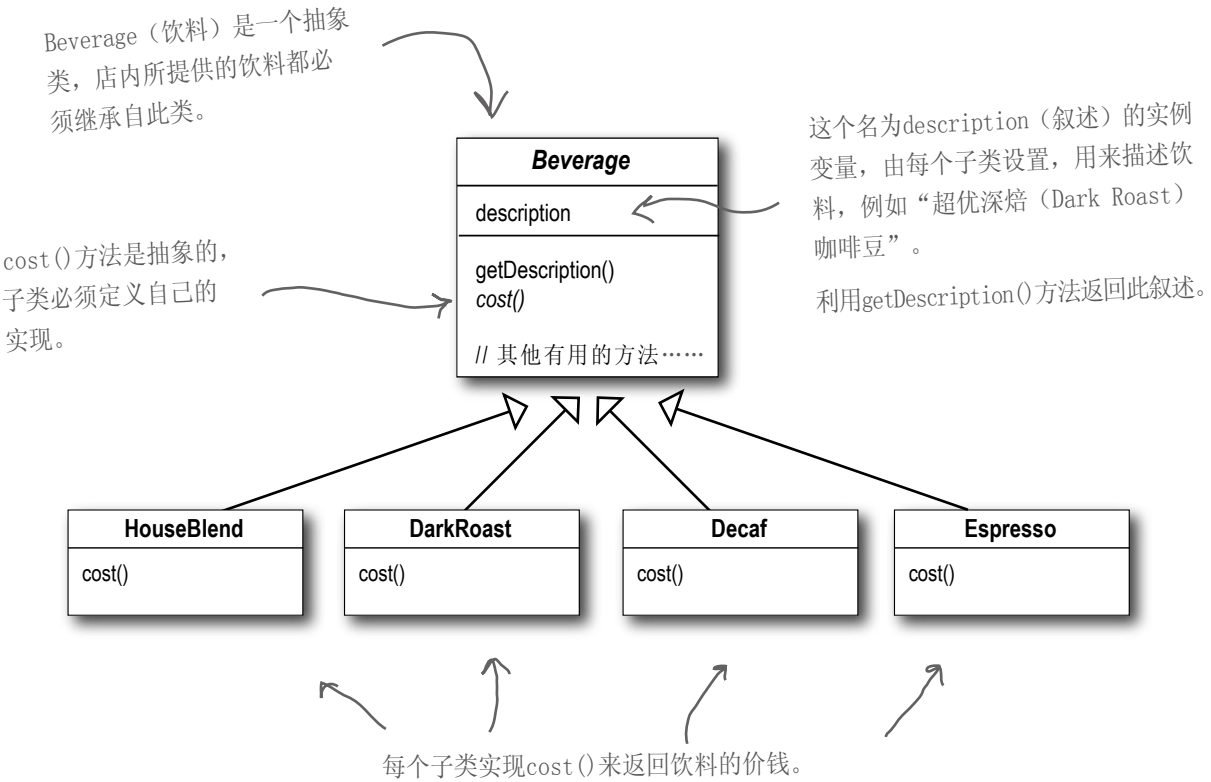
本章可以称为“给爱用继承的人一个全新的设计眼界”。我们即将再度探讨典型的继承滥用问题。你将在本章学到如何使用对象组合的方式，做到在运行时装饰类。为什么呢？一旦你熟悉了装饰的技巧，你将能够在不修改任何底层代码的情况下,给你的（或别人的）对象赋予新的职责。

欢迎来到星巴兹咖啡

星巴兹（Starbuzz）是以扩张速度最快而闻名的咖啡连锁店。如果你在街角看到它的店，在对面街上肯定还会看到另一家。

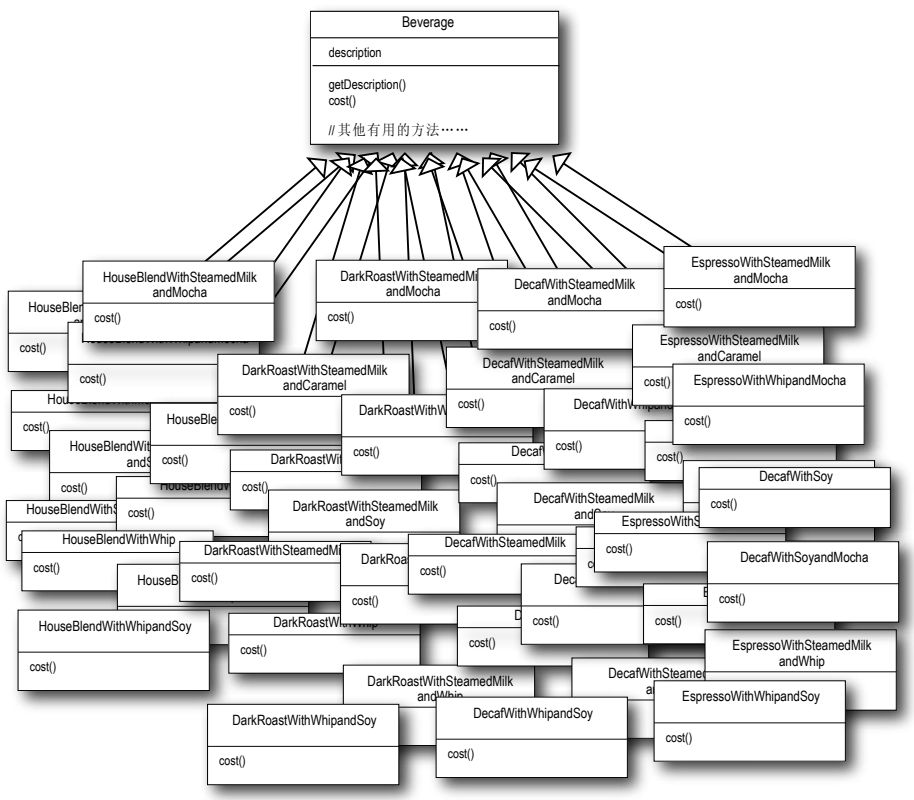
因为扩张速度实在太快了，他们准备更新订单系统，以合乎他们的饮料供应要求。

他们原先的类设计是这样的……



购买咖啡时，也可以要求在其中加入各种调料，例如：蒸奶（Steamed Milk）、豆浆（Soy）、摩卡（Mocha，也就是巧克力风味）或覆盖奶泡。星巴兹会根据所加入的调料收取不同的费用。所以订单系统必须考虑到这些调料部分。

这是他们的第一个尝试……

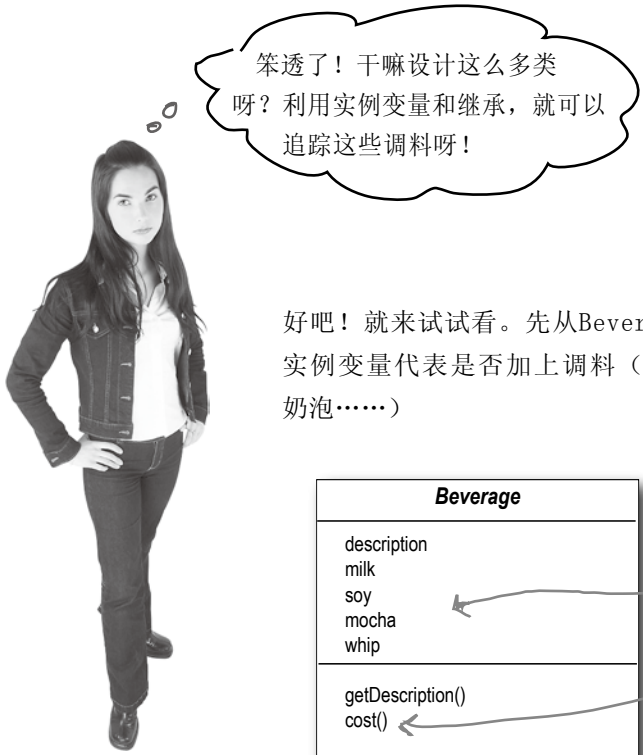


每个cost()方法将计算出咖啡加上订单上各种调料的价钱。

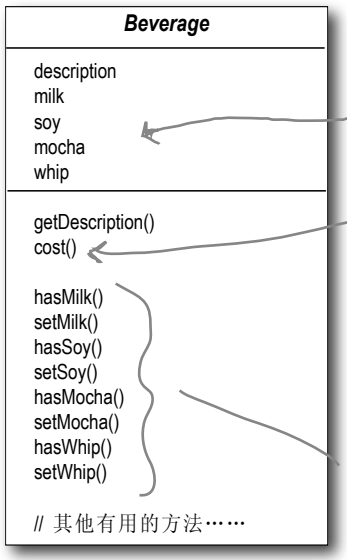
很明显，星巴兹为自己制造了一个维护恶梦。如果牛奶的价钱上扬，怎么办？新增一种焦糖调料风味时，怎么办？

造成这种维护上的困难，究竟违反了我们之前提过的哪种设计原则？

暗示：违反了两个原则，而且很严重！



好吧！就来试试看。先从Beverage基类下手，加上实例变量代表是否加上调料（牛奶、豆浆、摩卡、奶泡……）



各种调料的新的布尔值

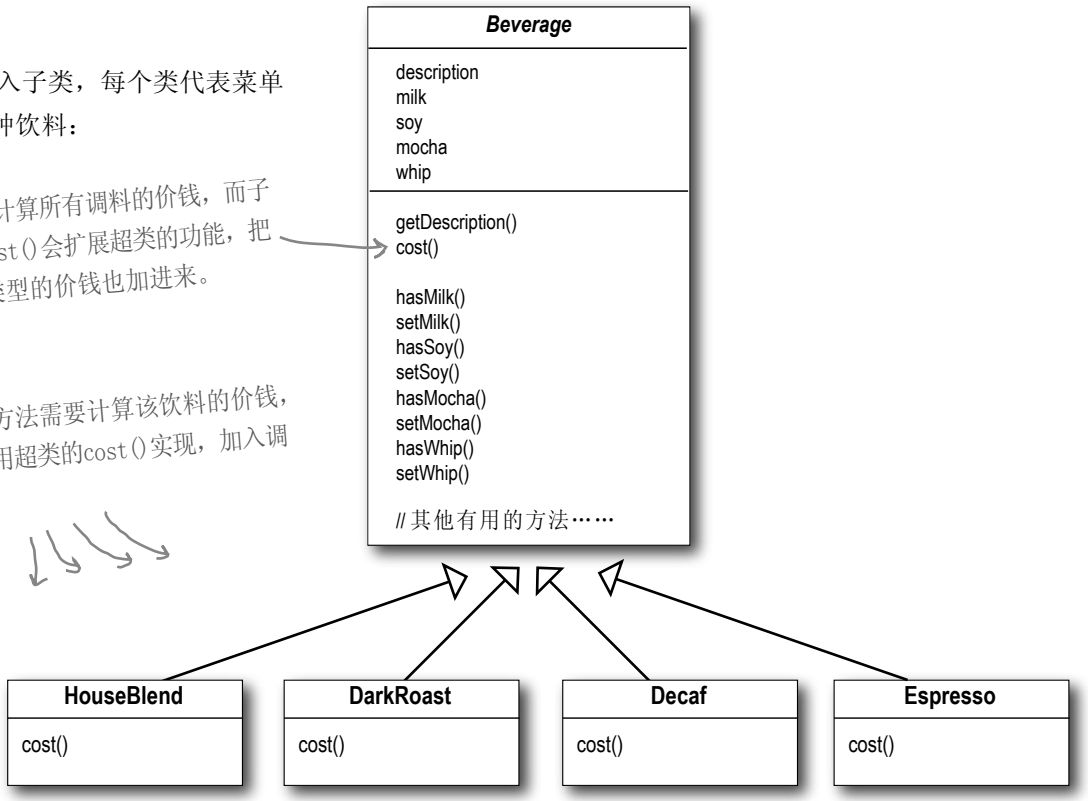
现在，Beverage类中的cost()不再是一个抽象方法，我们提供了cost()的实现，让它计算要加入各种饮料的调料价钱。子类仍将覆盖cost()，但是会调用超类的cost()，计算出基本饮料加上调料的价钱。

这些方法取得和设置调料的布尔值。

现在加入子类，每个类代表菜单上的一种饮料：

超类cost()将计算所有调料的价钱，而子类覆盖过的cost()会扩展超类的功能，把指定的饮料类型的价钱也加进来。

每个cost()方法需要计算该饮料的价钱，然后通过调用超类的cost()实现，加入调料的价钱。



Sharpen your pencil

请为下面类的cost()方法书写代码（用伪Java代码即可）。

```
public class Beverage {
   public double cost() {




   }
}
```

```
public class DarkRoast extends Beverage {

   public DarkRoast() {
      description = "Most Excellent Dark Roast";
   }

   public double cost() {




   }
}
```

看吧！一共只需要五个类，这正是我们要的做法。

我不确定耶！通过思考设计将来可能需要的变化，我可以看出来这种方法有一些潜在的问题。

Sharpen your pencil

当哪些需求或因素改变时会影响这个设计？

调料价钱的改变会使我们更改现有代码。

一旦出现新的调料，我们就需要加上新的方法，并改变超类中的cost()方法。

以后可能会开发出新饮料。对这些饮料而言（例如：冰茶），某些调料可能并不适合，但是在这个设计方式中，Tea（茶）子类仍将继承那些不适合的方法，例如：hasWhip()（加奶泡）。

这是很糟糕的！我们在第1章就得到了这个教训。

万一顾客想要双倍摩卡咖啡，怎么办？

轮到你了：

大师与门徒……

大师：我说蚱蜢呀！距离我们上次见面已经有些时日，你对于继承的冥想，可有精进？

门　　徒：是的，大师。尽管继承威力强大，但是我体会到它并不总是能够实现最有弹性和最好维护的设计。

大师：啊！是的，看来你已经有所长进。那么，告诉我，我的门徒，不通过继承又能如何达到复用呢？

门徒：大师，我已经了解到利用组合（composition）和委托（delegation）可以在运行时具有继承行为的效果。

大师：好，好，继续……

门徒：利用继承设计子类的行为，是在编译时静态决定的，而且所有的子类都会继承到相同的行为。然而，如果能够利用组合的做法扩展对象的行为，就可以在运行时动态地进行扩展。

大师：很好，蚱蜢，你已经开始看到组合的威力了。

门徒：是的，我可以利用此技巧把多个新职责，甚至是设计超类时还没有想到的职责加在对象上。而且，可以不用修改原来的代码。

大师：利用组合维护代码，你认为效果如何？

门徒：这正是我要说的。通过动态地组合对象，可以写新的代码添加新功能，而无须修改现有代码。既然没有改变现有代码，那么引进bug或产生意外副作用的机会将大幅度减少。

大师：非常好。蚱蜢，今天的谈话就到这里。希望你能在这个主题上更深入……牢记，代码应该如同晚霞中的莲花一样地关闭（免于改变），如同晨曦中的莲花一样地开放（能够扩展）。

开放-关闭原则

此刻，蚱蜢面临最重要的设计原则之一：

设计原则

类应该对扩展开放，对修改关闭。

请进，现在“开放”中。欢迎用任何你想要的行为来扩展我们的类。如果你的需要或需求有所改变（我们知道这一定会发生的），那就来吧！动手扩展吧！

抱歉，现在是“关闭”状态。没错。我们花了许多时间得到了正确的代码，还解决了所有的bug，所以不能让你修改现有代码。我们必须关闭代码以防止被修改。如果你不喜欢，可以找经理谈。

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。如能实现这样的目标，有什么好处呢？这样的设计具有弹性可以应对改变，可以接受新的功能来应对改变的需求。

there are no Dumb Questions

问： 对扩展开放，对修改关闭？听起来很矛盾。设计如何兼顾两者？

答： 这是一个很好的问题。乍听之下，的确感到矛盾，毕竟，越难修改的事物，就越难以扩展，不是吗？

但是，有一些聪明的OO技巧，允许系统在不修改代码的情况下，进行功能扩展。想想观察者模式（在第2章）……通过加入新的观察者，我们可以在任何时候扩展Subject（主题），而且不需向主题中添加代码。以后，你还会陆续看到更多的扩展行为的其他OO设计技巧。

问： 好吧！我了解观察者（Observ-able），但是该如何将某件东西设计成可以扩展，又禁止修改？

答： 许多模式是长期经验的实证，可通过提供扩展的方法来保护代码免于被修改。在本章，将看到使用装饰者模式的一个好例子，完全遵循开放-关闭原则。

问： 我如何让设计的每个部分都遵循开放-关闭原则？

答： 通常，你办不到。要让OO设计同时具备开放性和关闭性，又不修改现有的代码，需要花费许多时间和努力。一般来说，我们实在没有闲工夫把设计的每个部分都这么设计（而且，就算做得到，也可能只是一种浪费）。遵循开放-关闭原则，通常会引入新的抽象层次，增加代码的复杂度。你需要把注意力集中在设计中最有可能改变的地方，然后应用开放-关闭原则。

问： 我怎么知道，哪些地方的改变是更重要呢？

答： 这牵涉到设计OO系统的经验，和对你工作领域的了解。多看一些其他的例子可以帮你学习如何辨别设计中的变化区。

虽然似乎有点矛盾，但是的确有一些技术可以允许在不直接修改代码的情况下对其进行扩展。

在选择需要被扩展的代码部分时要小心。每个地方都采用开放-关闭原则，是一种浪费，也没必要，还会导致代码变得复杂且难以理解。

认识装饰者模式

好了，我们已经了解利用继承无法完全解决问题，在星巴兹遇到的问题有：类数量爆炸、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，在这里要采用不一样的做法：我们要以饮料为主体，然后在运行时以调料来“装饰”（decorate）饮料。比方说，如果顾客想要摩卡和奶泡深焙咖啡，那么，要做的是：

1. 拿一个深焙咖啡（DarkRoast）对象
2. 以摩卡（Mocha）对象装饰它
3. 以奶泡（Whip）对象装饰它
4. 调用cost()方法，并依赖委托（delegate）将调料的价钱加上去。

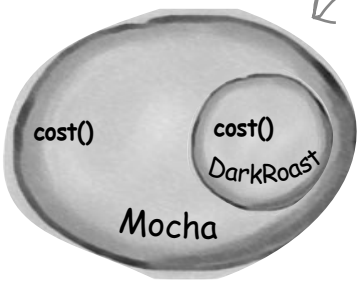
好了！但是如何“装饰”一个对象，而“委托”又要如何与此搭配使用呢？给一个暗示：把装饰者对象当成“包装者”。让我们看看这是如何工作的……

以装饰者构造饮料订单

❶ 以DarkRoast对象开始

别忘了，DarkRoast继承自Beverage，且有一个用来计算饮料价钱的cost()方法。

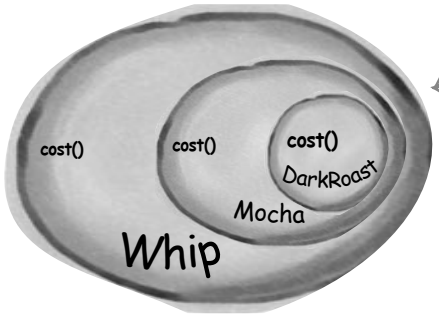
❷ 顾客想要摩卡（Mocha），所以建立一个Mocha对象，并用它将DarkRoast对象包（wrap）起来。



Mocha对象是一个装饰者，它的类型“反映”了它所装饰的对象（本例中，就是Beverage）。所谓的“反映”，指的就是两者类型一致。

所以Mocha也有一个cost()方法。通过多态，也可以把Mocha所包裹的任何Beverage当成是Beverage（因为Mocha是Beverage的子类型）。

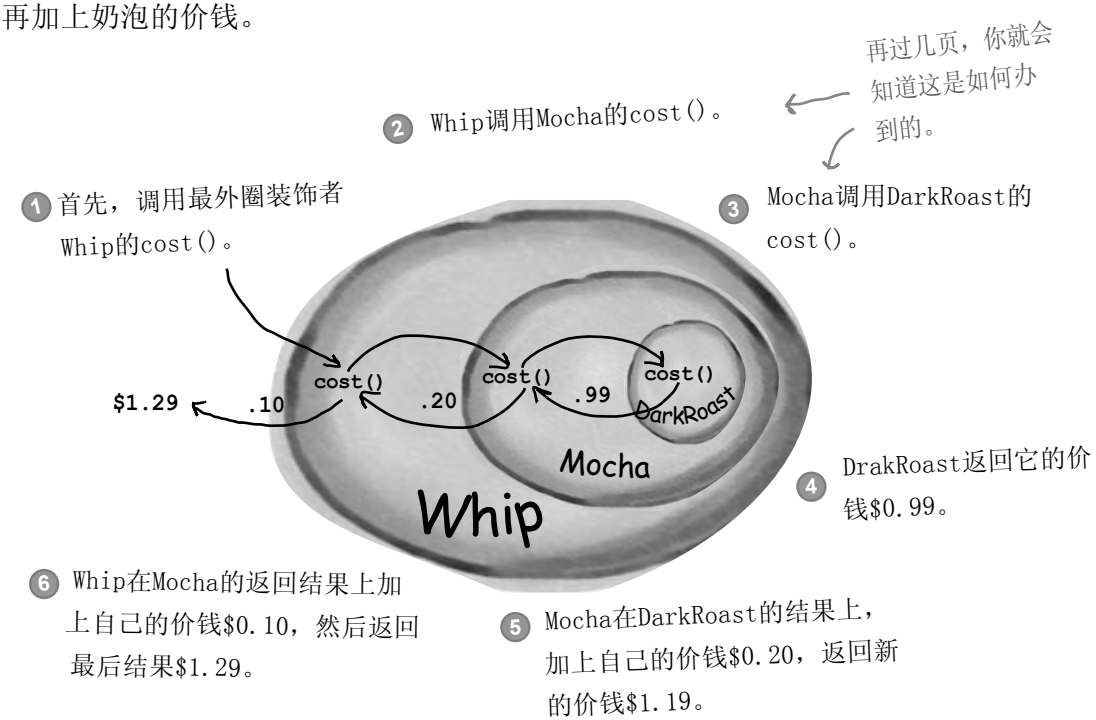
❸ 顾客也想要奶泡（Whip），所以需要建立一个Whip装饰者，并用它将Mocha对象包起来。别忘了，DarkRoast继承自Beverage，且有一个cost()方法，用来计算饮料价钱。



Whip是一个装饰者，所以它也反映了DarkRoast类型，并包括一个cost()方法。

所以，被Mocha和Whip包起来的DarkRoast对象仍然是一个Beverage，仍然可以具有DarkRoast的一切行为，包括调用它的cost()方法。

❹ 现在，该是为顾客算钱的时候了。通过调用最外圈装饰者（Whip）的cost()就可以办得到。Whip的cost()会先委托它装饰的对象（也就是Mocha）计算出价钱，然后再加上奶泡的价钱。



好了，这是目前所知道的一切……

- 装饰者和被装饰对象有相同的超类型。
- 你可以用一个或多个装饰者包装一个对象。
- 既然装饰者和被装饰对象有相同的超类型，所以在任何需要原始对象（被包装的）的场合，可以用装饰过的对象代替它。
- 装饰者可以在所委托被装饰者的行为之前与/或之后，加上自己的行为，以达到特定的目的。 关键点！
- 对象可以在任何时候被装饰，所以可以在运行时动态地、不限量地用你喜欢的装饰者来装饰对象。

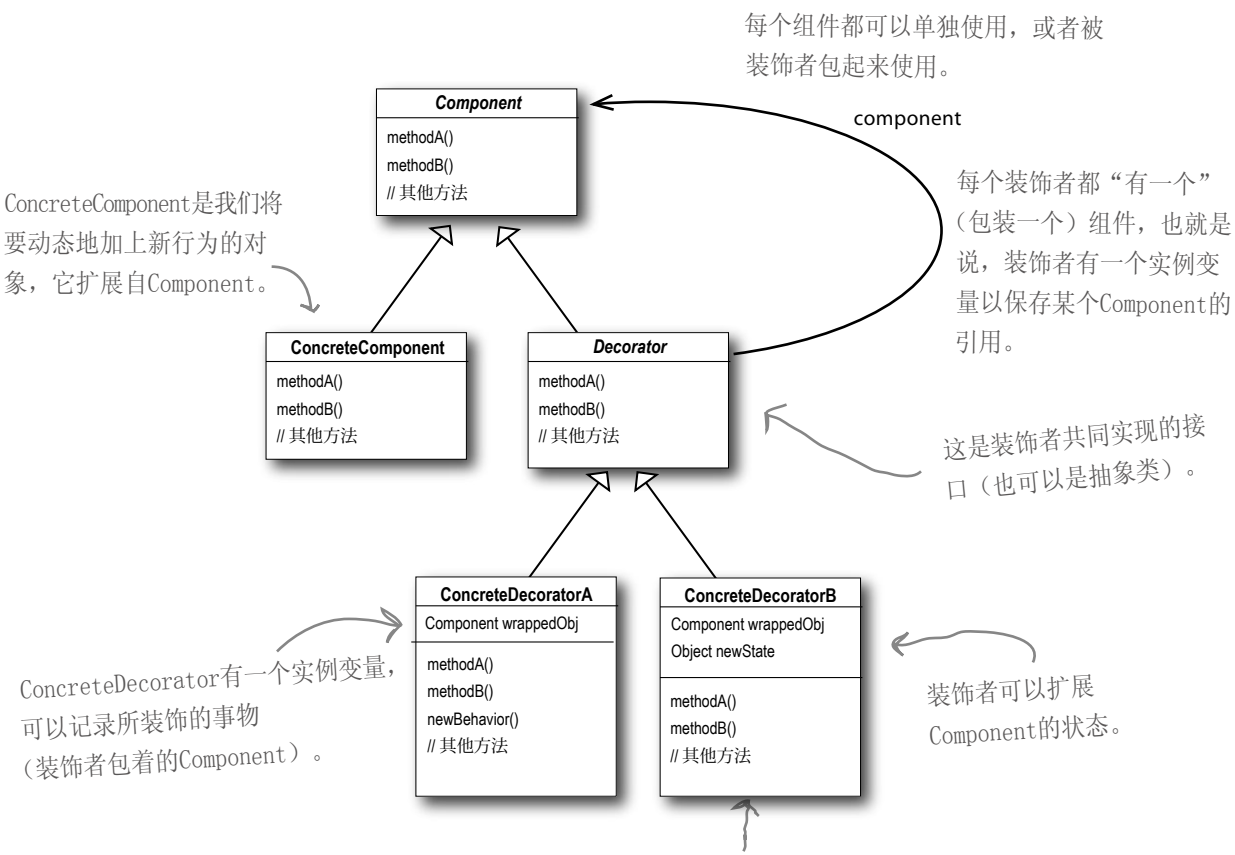
现在，就来看看装饰者模式的定义，并写一些代码，了解它到底是怎么工作的。

定义装饰者模式

让我们先来看看装饰者模式的说明：

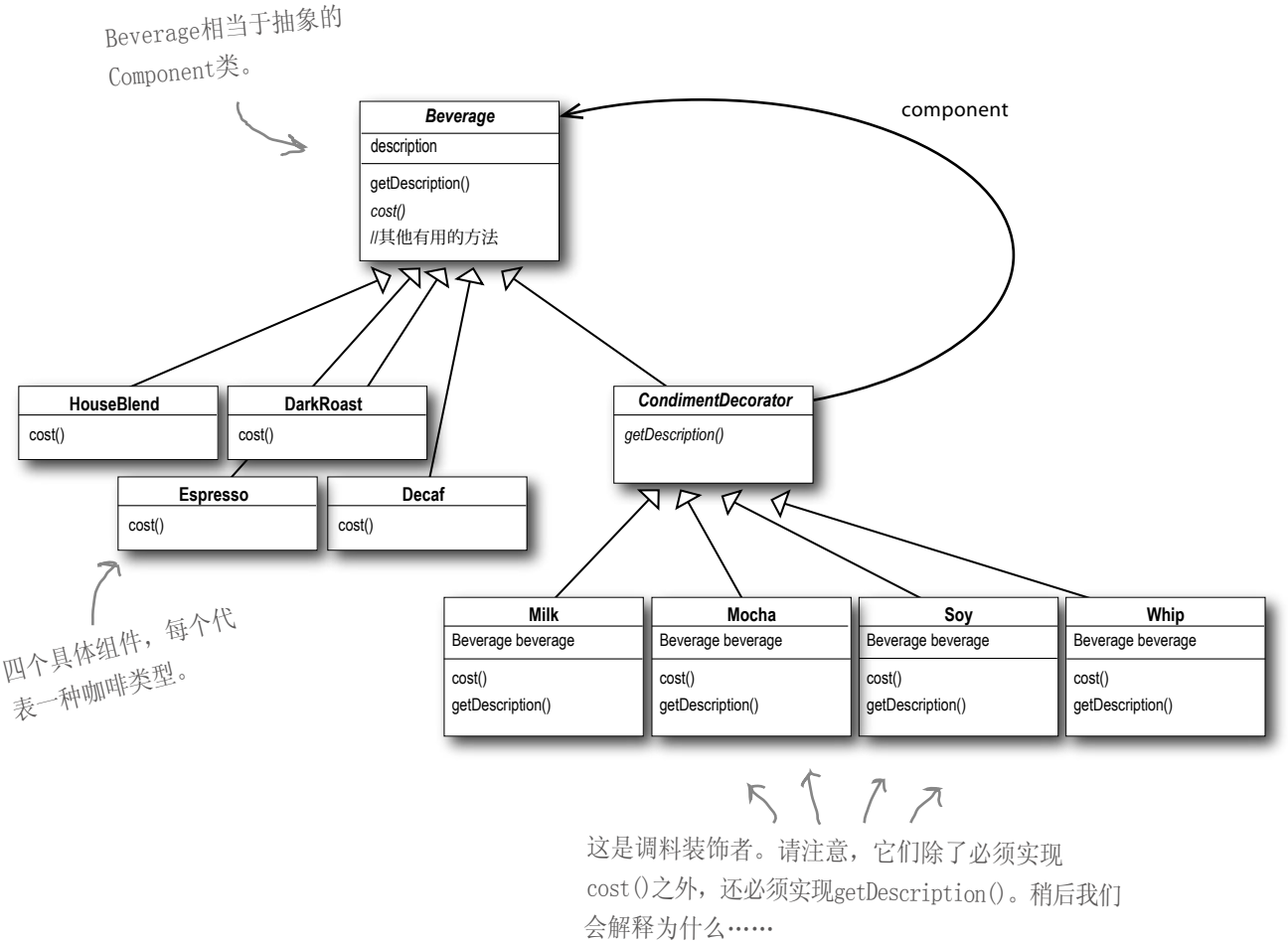
> **装饰者模式**动态地将责任附加到对象上。若要扩展功能，装饰者提供了比继承更有弹性的替代方案。

虽然这说明了装饰者模式的“角色”，但是没说明怎么在我们的实现中实际“应用”它。我们来看看类图，会有些帮助（下一页，我们会将此结构套用在饮料问题上）。



装饰我们的饮料

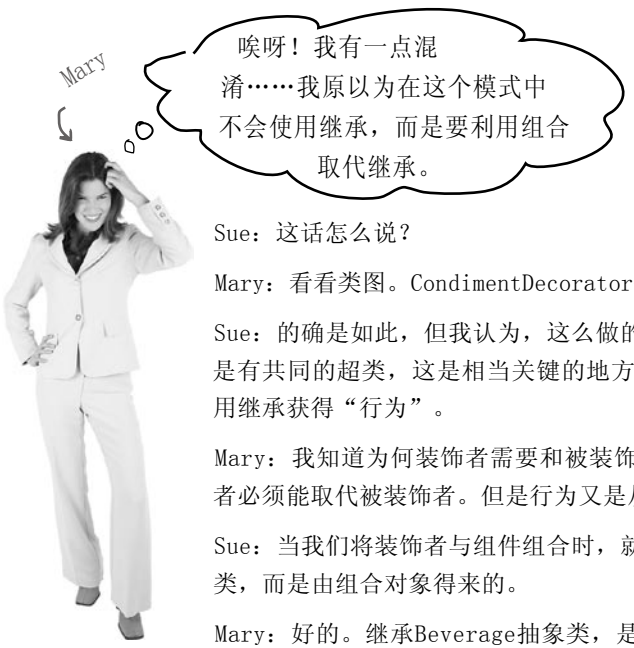
好吧！让星巴兹饮料也能符合此框架……



在往下看之前，想想如何实现咖啡和调料的cost()方法。也思考一下如何实现调料的getDescription()方法。

办公室隔间对话

在继承和组合之间，观念有一些混淆。



Sue：这话怎么说？

Mary：看看类图。CondimentDecorator扩展自Beverage类，这用到了继承，不是吗？

Sue：的确是如此，但我认为，这么做的重点在于，装饰者和被装饰者必须是一样的类型，也就是有共同的超类，这是相当关键的地方。在这里，我们利用继承达到“类型匹配”，而不是利用继承获得“行为”。

Mary：我知道为何装饰者需要和被装饰者（亦即被包装的组件）有相同的“接口”，因为装饰者必须能取代被装饰者。但是行为又是从哪里来的？

Sue：当我们将装饰者与组件组合时，就是在加入新的行为。所得到的新行为，并不是继承自超类，而是由组合对象得来的。

Mary：好的。继承Beverage抽象类，是为了有正确的类型，而不是继承它的行为。行为来自装饰者和基础组件，或与其他装饰者之间的组合关系。

Sue：正是如此。

Mary：哦！我明白了。而且因为使用对象组合，可以把所有饮料和调料更有弹性地加以混和与匹配，非常方便。

Sue：是的。如果依赖继承，那么类的行为只能在编译时静态决定。换句话说，行如果不是来自超类，就是子类覆盖后的版本。反之，利用组合，可以把装饰者混合着用……而且是在“运行时”。

Mary：而且，如我所理解的，我们可以在任何时候，实现新的装饰者增加新的行为。如果依赖继承，每当需要新行为时，还得修改现有的代码。

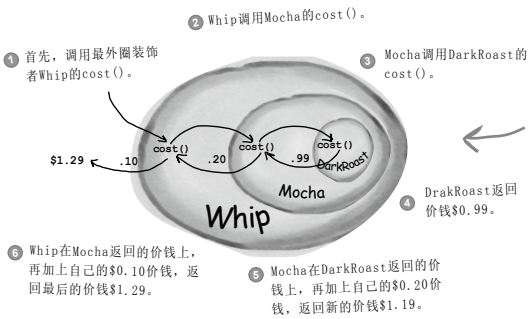
Sue：的确如此。

Mary：我还剩下一个问题，如果我们需要继承的是component类型，为什么不Beverage类设计成一个接口，而是设计成一个抽象类呢？

Sue：关于这个嘛，还记得吗？当初我们从星巴兹拿到这个程序时，Beverage“经”是一个抽象类了。通常装饰者模式是采用抽象类，但是在Java中可以使用接口。尽管如此，通常我们都努力避免修改现有的代码，所以，如果抽象类运作得好好的，还是别去修改它。

新咖啡师傅特训

如果有一张单子点的是：“双倍摩卡豆浆奶泡拿铁咖啡”，请使用菜单得到正确的价钱并画一个图来表达你的设计，采用和几页前一样的格式。



这张图是“深度烘焙摩卡奶泡”饮料。

Sharpen your pencil

把图画在这里

写下星巴兹的代码

该是把设计变成真正的代码的时候了！

先从Beverage类下手，这不需要改变星巴兹原始的设计。如下所示：

```
public abstract class Beverage {
    String description = "Unknown Beverage";

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

Beverage是一个抽象类，有两个方法：getDescrip-tion()及cost()。

getDescription()已经在此实现了，但是cost()必须在子类中实现。

Beverage很简单。让我们也来实现Condiment（调料）抽象类，也就是装饰者类吧：

首先，必须让Condiment Decorator能够取代Beverage，所以将Condiment Decorator扩展自 Beverage 类。

```
public abstract class CondimentDecorator extends Beverage {
    public abstract String getDescription();
}
```

所有的调料装饰者都必须重新实现getDescription()方法。稍后我们会解释为什么 ……

写饮料的代码

现在，已经有了基类，让我们开始开始实现一些饮料吧！先从浓缩咖啡（Espresso）开始。别忘了，我们需要为具体的饮料设置描述，而且还必须实现cost()方法。

首先，让Espresso扩展自Beverage类，因为Espresso是一种饮料。

```
public class Espresso extends Beverage {

    public Espresso() {
        description = "Espresso";
    }

    public double cost() {
        return 1.99;
    }
}
```

为了要设置饮料的描述，我们写了一个构造器。记住，description实例变量继承自Beverage。

最后，需要计算Espresso的价钱，现在不需要管调料的价钱，直接把Espresso的价格$1.99返回即可。

```
public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = "House Blend Coffee";
    }

    public double cost() {
        return .89;
    }
}
```

这是另一种饮料，做法和Espresso一样，只是把Espresso名称改为"HouseBlend Coffee"，并返回正确的价钱$0.89。

你可以自行建立另外两种饮料类（DarkRoast和Decaf），做法都一样。

写调料代码

如果你回头去看看装饰者模式的类图，将发现我们已经完成了抽象组件（Beverage），有了具体组件（HouseBlend），也有了抽象装饰者（CondimentDecorator）。现在，我们就来实现具体装饰者。先从摩卡下手：

摩卡是一个装饰者，所以让它扩展自CondimentDecorator。

别忘了，CondimentDecorator扩展自Beverage。

```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }

    public double cost() {
        return .20 + beverage.cost();
    }
}
```

要让Mocha能够引用一个Beverage，做法如下：

(1)用一个实例变量记录饮料，也就是被装饰者。

(2)想办法让被装饰者（饮料）被记录到实例变量中。这里的做法是：把饮料当作构造器的参数，再由构造器将此饮料记录在实例变量中。

我们希望叙述不只是描述饮料（例如“DarkRoast”），而是完整地连调料都描述出来（例如“DarkRoast, Mocha”）。所以首先利用委托的做法，得到一个叙述，然后在其后加上附加的叙述（例如“Mocha”）。

要计算带Mocha饮料的价钱。首先把调用委托给被装饰对象，以计算价钱，然后再加上Mocha的价钱，得到最后结果。

在下一页，我们会实际实例化一个饮料对象，然后用各种调料（装饰者）包装它。但是，在这么做之前，首先……

写下Soy和Whip调料的代码，并完成编译。你需要它们，否则将无法进行下一页的程序。

供应咖啡

恭喜你，是时候舒服地坐下来，点一些咖啡，看看你利用装饰者模式设计出的灵活系统是多么神奇了。

这是用来下订单的一些测试代码★：

```
public class StarbuzzCoffee {

    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()
                + " $" + beverage.cost());

        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription()
                + " $" + beverage2.cost());

        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription()
                + " $" + beverage3.cost());
    }
}
```

订一杯Espresso，不需要调料，打印出它的描述与价钱。

制造出一个DarkRoast对象。

用Mocha装饰它。

用第二个Mocha装饰它。

用Whip装饰它。

最后，再来一杯调料为豆浆、摩卡、奶泡的HouseBlend咖啡。

★ 当我们介绍到“工厂”和“生成器”设计模式时，将有更好的方式建立被装饰者对象。注意，关于“生成器模式”请参考本书附录。

现在，来看看实验结果：

```
File  Edit  Window  Help  CloudsInMyCoffee
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
%
```

there are no Dumb Questions

问： 如果我将代码针对特定种类的具体组件（例如HouseBlend），做一些特殊的事（例如，打折），我担心这样的设计是否恰当。因为一旦用装饰者包装HouseBlend，就会造成类型改变。

答： 的确是这样。如果你把代码写成依赖于具体的组件类型，那么装饰者就会导致程序出问题。只有在针对抽象组件类型编程时，才不会因为装饰者而受到影响。但是，如果的确针对特定的具体组件编程，就应该重新思考你的应用架构，以及装饰者是否适合。

问： 对于使用到饮料的某些客户来说，会不会容易不使用最外圈的装饰者呢？比方说，如果我有深焙咖啡，以摩卡、豆浆、奶泡来装饰，引用到豆浆而不是奶泡，代码会好写一些，这意味着订单里没有奶泡了。

答： 你当然可以争辩说，使用装饰者模式，你必须管理更多的对象，所以犯下你所说的编码错误的机会会增加。但是，装饰者通常是用其他类似于工厂或生成器这样的模式创建的。一旦我们讲到这两个模式，你就会明白具体的组件及其装饰者的创建过程，它们会“封装得很好”，所以不会有这种问题。

问： 装饰者知道这一连串装饰链条中其他装饰者的存在吗？比方说，我想要让getDescription()列出“Whip,Double Mocha”而不是“Mocha,Whip,Mocha”，这需要最外圈的装饰者知道有哪些装饰者牵涉其中了。

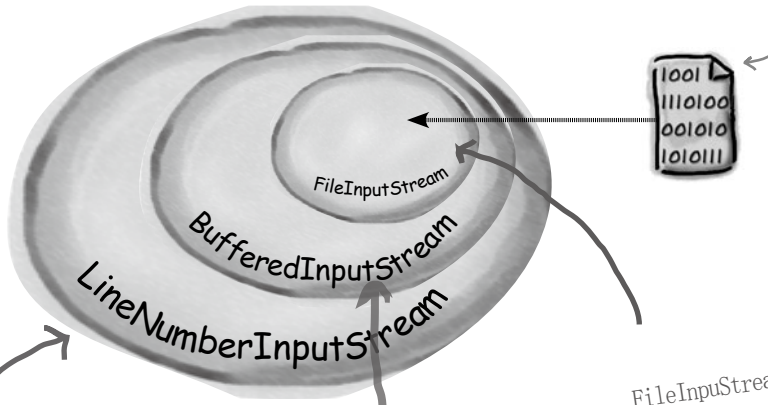
答： 装饰者该做的事，就是增加行为到被包装对象上。当需要窥视装饰者链中的每一个装饰者时，这就超出他们的天赋了。但是，并不是做不到。可以写一个CondimentPrettyPrint装饰者，解析出最后的描述字符串，然后把“Mocha,Whip, Mocha”变成“Whip,Double Mocha”。如果能把getDescription()的返回值变成ArrayList类型，让每个调料名称独立开来，那么CondimentPrettyPrint方法会更容易编写。

我们在星巴兹的朋友决定开始在菜单上加上咖啡的容量大小，供顾客可以选择小杯（tall）、中杯（grande）、大杯（venti）。星巴兹认为这是任何咖啡都必须具备的，所以在Beverage类中加上了getSize()与setSize()。他们也希望调料根据咖啡容量收费，例如：小中大杯的咖啡加上豆浆，分别加收0.10、0.15、0.20美金。

如何改变装饰者类应对这样的需求？

真实世界的装饰者：Java I/O

java.io包内的类太多了，简直是……“排山倒海”。你第一次（还有第二次和第三次）看到这些API发出“哇”的惊叹时，放心，你不是唯一受到惊吓的人。现在，你已经知道装饰者模式，这些I/O的相关类对你来说应该更有意义了，因为其中许多类都是装饰者。下面是一个典型的对象集合，用装饰者来将功能结合起来，以读取文件数据：



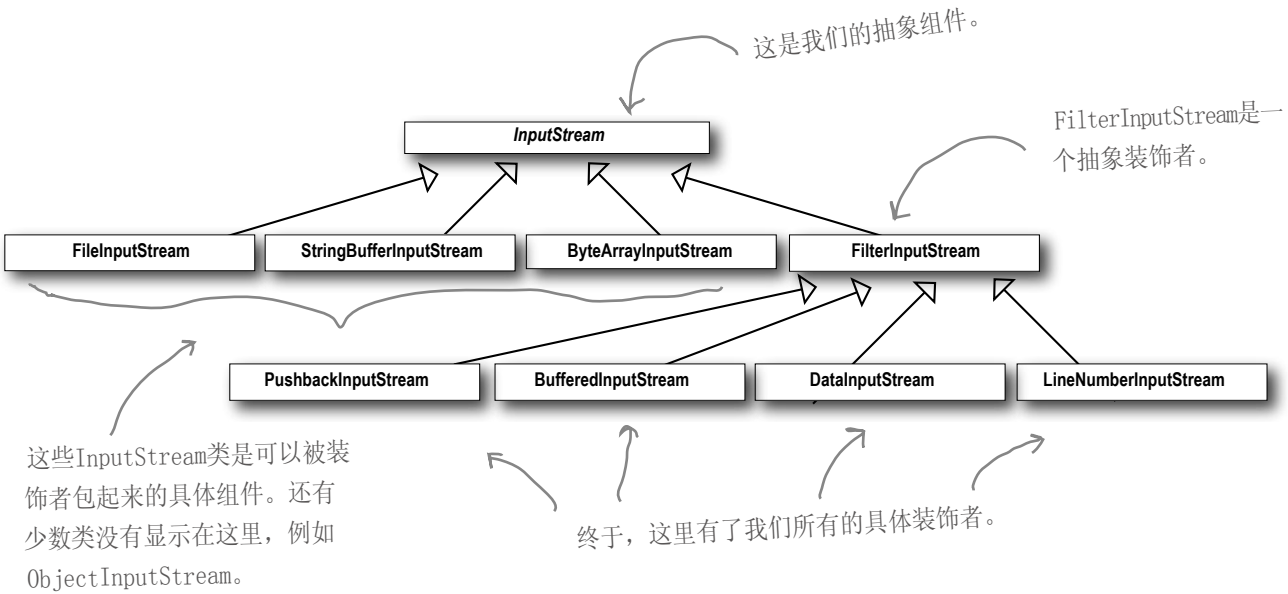
FileInpuStream是被装饰的“组件”。Java I/O程序库提供了几个组件，包括了FileInputStream、StringBufferInputStream、ByteArrayInputStream……等。这些类都提供了最基本的字节读取功能。

BufferedInputStream是一个具体的“装饰者”，它加入两种行为：利用缓冲输入来改进性能；用一个readLine()方法（用来一次读取一行文本输入数据）来增强接口。

LineNumberInputStream也是一个具体的“装饰者”。它加上了计算行数的能力。

BufferedInputStream及LineNumberInputStream都扩展自FilterInputStream，而FilterInputStream是一个抽象的装饰类。

装饰java.io类



你可以发现，和星巴兹的设计相比，java.io其实没有多大的差异。我们把java.io API范围缩小，让你容易查看它的文件，并组合各种“输入”流装饰者来符合你的用途。

你会发现“输出”流的设计方式也是一样的。你可能还会发现Reader/Writer流（作为基于字符数据的输入输出）和输入流/输出流的类相当类似（虽然有一些小差异和不一致之处，但是相当雷同，所以你应该可以了解这些类）。

但是Java I/O也引出装饰者模式的一个“缺点”：利用装饰者模式，常常造成设计中有大量的小类，数量实在太多，可能会造成使用此API程序员的困扰。但是，现在你已经了解了装饰者的工作原理，以后当使用别人的大量装饰的API时，就可以很容易地辨别出他们的装饰者类是如何组织的，以方便用包装方式取得想要的行为。

编写自己的Java I/O装饰者

你已经知道装饰者模式，也看过Java I/O类图，应该已经准备好编写自己的输入装饰者了。

这个想法怎么样：编写一个装饰者，把输入流内的所有大写字符转成小写。举例：当读取“I know the Decorator Pattern therefore I RULE!”，装饰者会将它转成“i know the decorator pattern therefore i rule!”

不要忘了导入java.io……（这里省略了）

首先，扩展FilterInputStream，这是所有InputStream的抽象装饰者。

```
public class LowerCaseInputStream extends FilterInputStream {
    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = super.read();
        return (c == -1 ? c : Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int offset, int len) throws IOException {
        int result = super.read(b, offset, len);
        for (int i = offset; i < offset+result; i++) {
            b[i] = (byte)Character.toLowerCase((char)b[i]);
        }
        return result;
    }
}
```

现在，必须实现两个read()方法，一个针对字节，一个针对字节数组，把每个是大写字符的字节（每个代表一个字符）转成小写。

别忘了：我们在代码中没有列出package与import语句。如果想取得完整的源代码，可以到第xxxv页中列出的wickedlysmart网站URL下载。

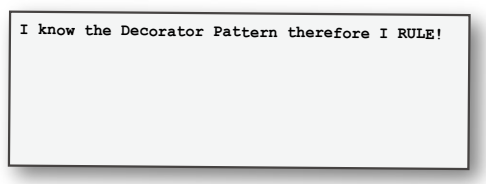
测试你的新Java I/O装饰者

写个小程序，来测试刚写好的I/O 装饰者：

```
public class InputTest {
    public static void main(String[] args) throws IOException {
        int c;
        try {
            InputStream in =
                new LowerCaseInputStream(
                    new BufferedInputStream(
                        new FileInputStream("test.txt")));

            while((c = in.read()) >= 0) {
                System.out.print((char)c);
            }

            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

设置FileInputStream，先用BufferedInputStream装饰它，再用我们崭新的LowerCaseInputStream过滤器装饰它。

只用流来读取字符，一直到文件尾端。每读一个字符，就马上将它显示出来。



test.txt file

你需要做出这个文件。

运行看看：

```
File Edit Window Help DecoratorsRule
% java InputTest
i know the decorator pattern therefore i rule!
%
```

模式访谈

本周访问：
装饰者的告白

HeadFirst：欢迎装饰者模式，听说你最近情绪有点差？

装饰者：是的，我知道大家都认为我是一个有魅力的设计模式，但是，你知道吗？我也有自己的困扰，就和大家一样。

HeadFirst：愿意让我们分担一些你的困扰吗？

装饰者：当然可以。你知道我有能力为设计注入弹性，这是毋庸置疑的，但我也有“黑暗面”。有时候我会在设计中加入大量的小类，这偶尔会导致别人不容易了解我的设计方式。

HeadFirst：你能够举个例子吗？

装饰者：以Java I/O库来说，人们第一次接触到这个库时，往往无法轻易地理解它。但是如果他们能认识到这些类都是用来包装InputStream的，一切都会变得简单多了。

HeadFirst：听起来并不严重。你还是一个很好的模式，只需要一点点的教育，让大家知道怎么用，问题就解决了。

装饰者：恐怕不只这些，我还有类型问题。有些时候，人们在客户代码中依赖某种特殊类型，然后忽然导入装饰者，却又没有周详地考虑一切。现在，我的一个优点是,你通常可以透明地插入装饰者，客户程序甚至不需知道它是在和装饰者打交道。但是，如我刚刚所说的，有些代码会依赖特定的类型，而这样的代码一导入装饰者，嘭！出状况了！

HeadFirst：这个嘛,我相信每个人都必须了解到，在插入装饰者时，必须要小心谨慎。我不认为这是你的错！

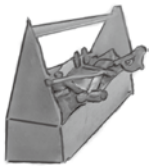
装饰者：我知道，我也试着不这么想。我还有一个问题，就是采用装饰者在实例化组件时，将增加代码的复杂度。一旦使用装饰者模式，不只需要实例化组件，还要把此组件包装进装饰者中，天晓得有几个。

HeadFirst：我下周会访谈工厂（Factory）模式和生成器（Builder）模式，我听说他们对这个问题有很大的帮助。

装饰者：那倒是真的。我应该常和这些家伙聊聊。

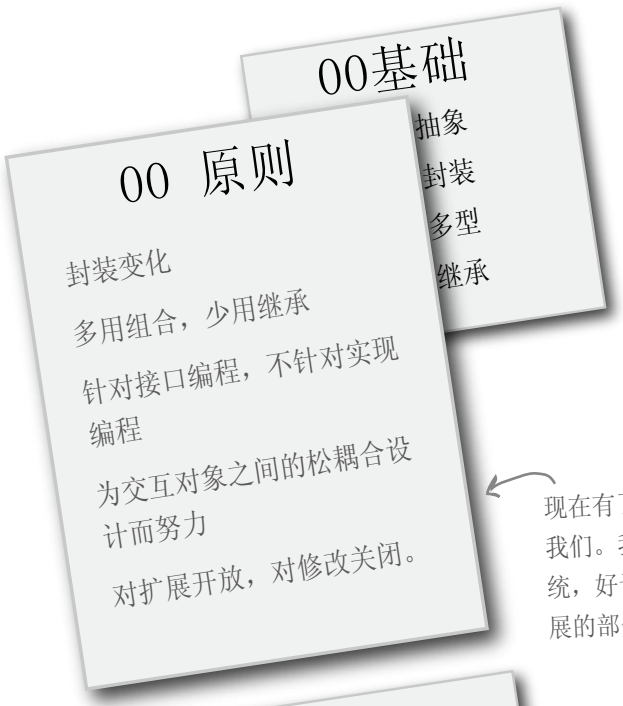
HeadFirst：我们都认为你是一个好的模式，适合用来建立有弹性的设计，维持开放－关闭原则。你要开心一点，别负面思考。

装饰者：我尽量吧，谢谢你！



设计箱内的工具

本章已经接近尾声，你的工具箱内又多了一个新的原则和一个新的模式。



要点

- 继承属于扩展形式之一，但不见得是达到弹性设计的最佳方式。
- 在我们的设计中，应该允许行为可以被扩展，而无须修改现有的代码。
- 组合和委托可用于在运行时动态地加上新的行为。
- 除了继承，装饰者模式也可以让我们扩展行为。
- 装饰者模式意味着一群装饰者类，这些类用来包装具体组件。
- 装饰者类反映出被装饰的组件类型（事实上，他们具有相同的类型，都经过接口或继承实现）。
- 装饰者可以在被装饰者的行为前面与/或后面加上自己的行为，甚至将被装饰者的行为整个取代掉，而达到特定的目的。
- 你可以用无数个装饰者包装一个组件。
- 装饰者一般对组件的客户是透明的，除非客户程序依赖于组件的具体类型。
- 装饰者会导致设计中出现许多小对象，如果过度使用，会让程序变得很复杂。

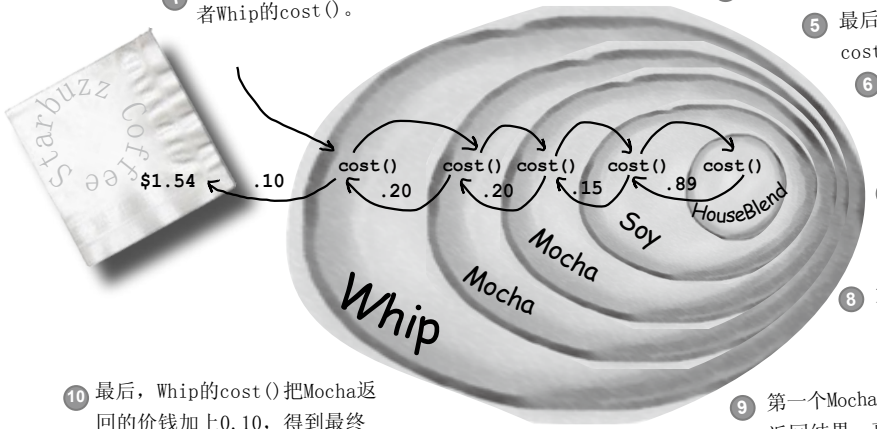
习题解答

```
public class Beverage {

  //为milkCost、soyCost、mochaCost
  //和whipCost声明实例变量。
  //为milk、soy、mocha和whip
  //声明getter与setter方法。


public double cost() {
      float condimentCost = 0.0;
      if (hasMilk()) {
         condimentCost += milkCost;
      }
      if (hasSoy()) {
         condimentCost += soyCost;
      }
      if (hasMocha()) {
         condimentCost += mochaCost;
      }
      if (hasWhip()) {
         condimentCost += whipCost;
      }
      return condimentCost;
   }
}
```

```
public class DarkRoast extends Beverage {

   public DarkRoast() {
      description = "Most Excellent Dark Roast";
   }

   public double cost() {

       return 1.99 + super.cost();

   }
}
```

新咖啡师傅特训

“有双摩卡、豆浆、奶泡的House Blend咖啡”



1. 首先，调用最外圈装饰者Whip的cost()。
2. Whip调用Mocha的cost()。
3. Mocha调用另一个Mocha的cost()。
4. 接着，Mocha调用Soy的cost()。
5. 最后，Soy调用HouseBlend的cost()。
6. HouseBlend的cost()返回0.89给Soy后，离开本层。
7. Soy的cost()把HouseBlend返回的结果加上0.15，返回给Mocha后，离开本层。
8. 第二个Mocha的cost()加上0.20，返回结果，离开本层。
9. 第一个Mocha的cost()加上0.20，返回结果，离开本层。
10. 最后，Whip的cost()把Mocha返回的价钱加上0.10，得到最终价钱为$1.54。

习题解答

我们在星巴兹的朋友决定开始在菜单上加上咖啡的容量大小，供顾客可以选择小杯（tall）、中杯（grande）、大杯（venti）。星巴兹认为这是任何咖啡都必须具备的，所以在Beverage类中加上了getSize()与setSize()。他们也希望调料根据咖啡容量收费，例如：小中大杯的咖啡加上豆浆，分别加收0.10、0.15、0.20美金。

如何改变装饰者类应对这样的需求？

```
public class Soy extends CondimentDecorator {
    Beverage beverage;

    public Soy(Beverage beverage) {
        this.beverage = beverage;
    }

    public int getSize() {
        return beverage.getSize();
    }

    public String getDescription() {
        return beverage.getDescription() + ", Soy";
    }

    public double cost() {
        double cost = beverage.cost();
        if (getSize() == Beverage.TALL) {
            cost += .10;
        } else if (getSize() == Beverage.GRANDE) {
            cost += .15;
        }  else if (getSize() == Beverage.VENTI) {
            cost += .20;
        }
        return cost;
    }
}
```

现在要把getSize()传播到被包装的饮料。因为所有的调料装饰者都会用到这个方法，所以也应该把它移到抽象类中。

在这里取得容量大小（全都传播到具体的饮料），然后加上适当的价钱。

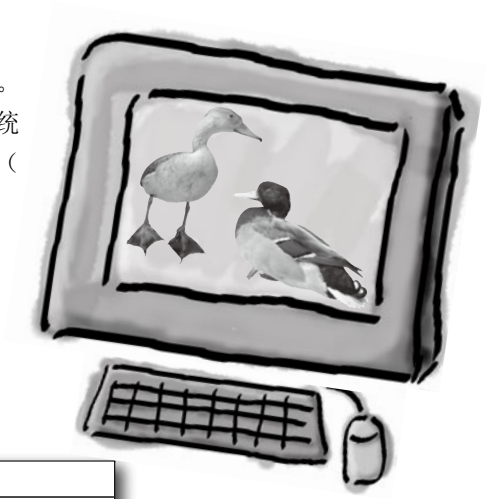
1 介绍设计模式

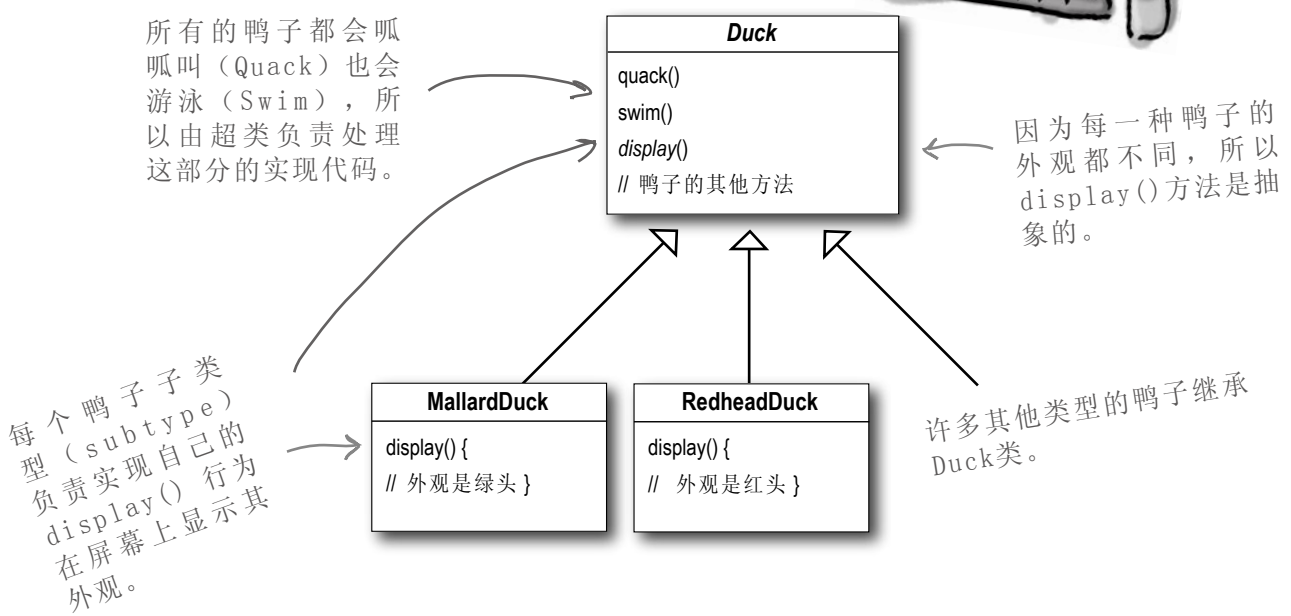
欢迎来到设计模式

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OO设计原则，并透过一个例子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。

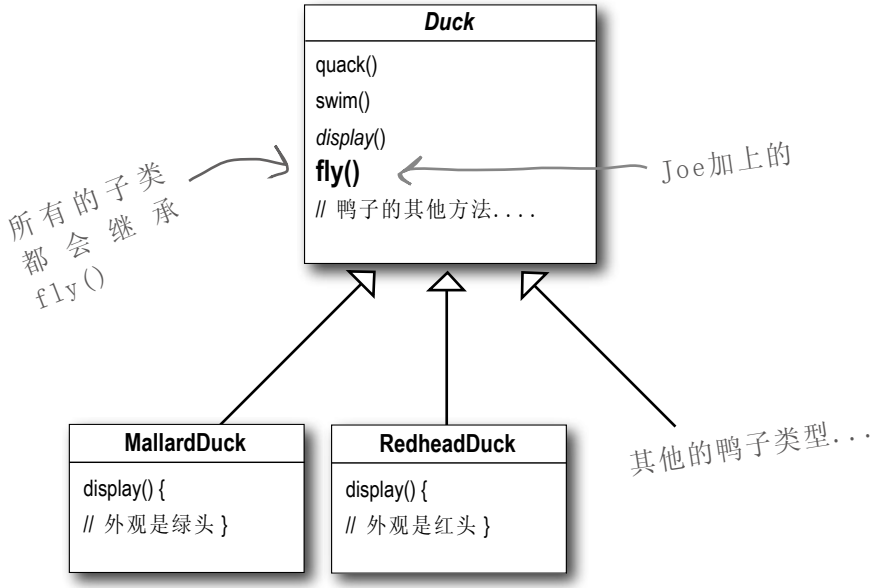
现在我们得让鸭子能飞

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？」

我们想要的

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。

Joe



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？

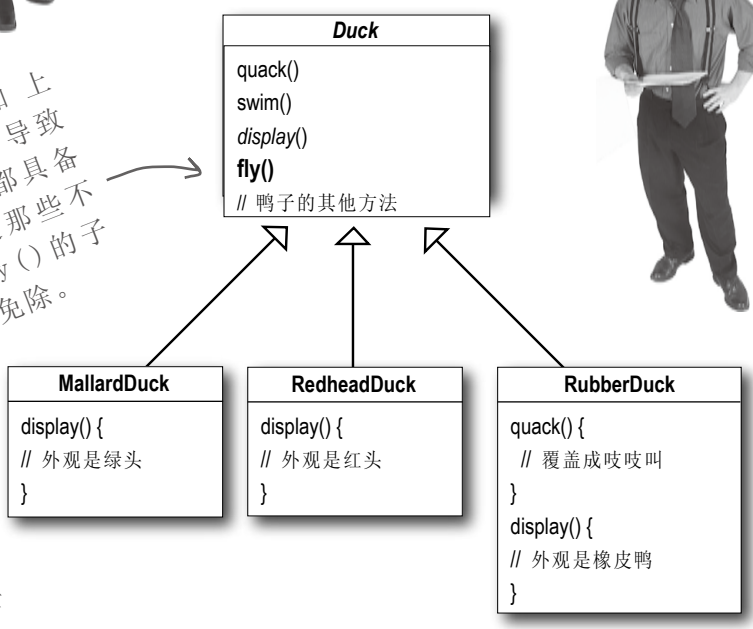
Joe忽略了一件事：并非Duck 所有的子类都会飞。当Joe在Duck超类中加上新的行为，这会使得某些子类也具有这个不恰当的行为。现在可好了！SimUDuck程序中有一个会飞的非动物。

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..

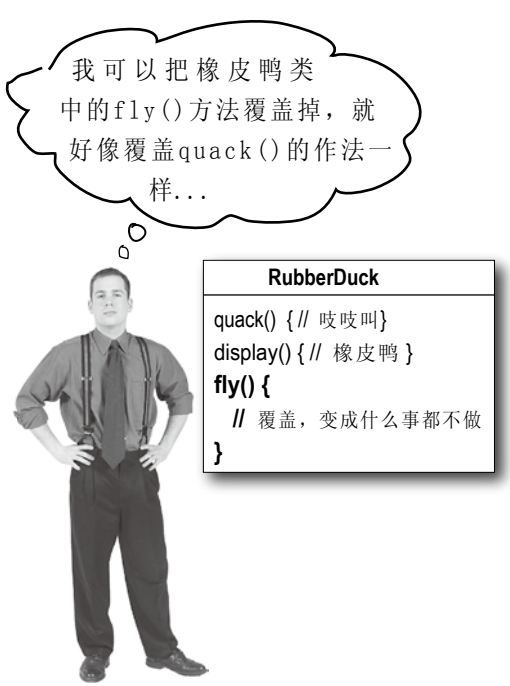
他体会到了一件事：当涉及「维护」时，为了「复用」（reuse）目的而使用继承，结局并不完美。

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）。

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

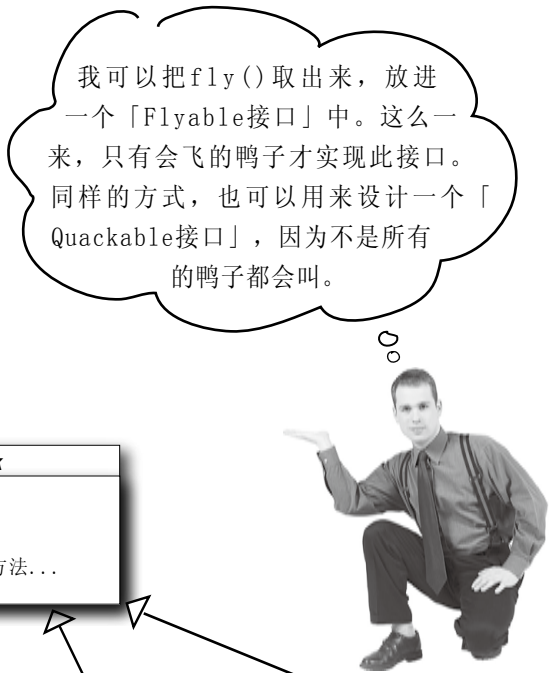
利用继承来提供Duck的行为，这会导致下列哪些缺点？（多选）

- ❑ 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. 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。

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？

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」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出苦难的一天。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，好让我们需要改变软件时，可以在对既有的代码影响最小的情况下，轻易达到花较少时间重做代码，而多让程序去做更酷的事。该有多好...

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，用何种编程语言，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」。

驱动改变的因素很多。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，你必须得往下追踪并修改每一个定义此行为的类，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。

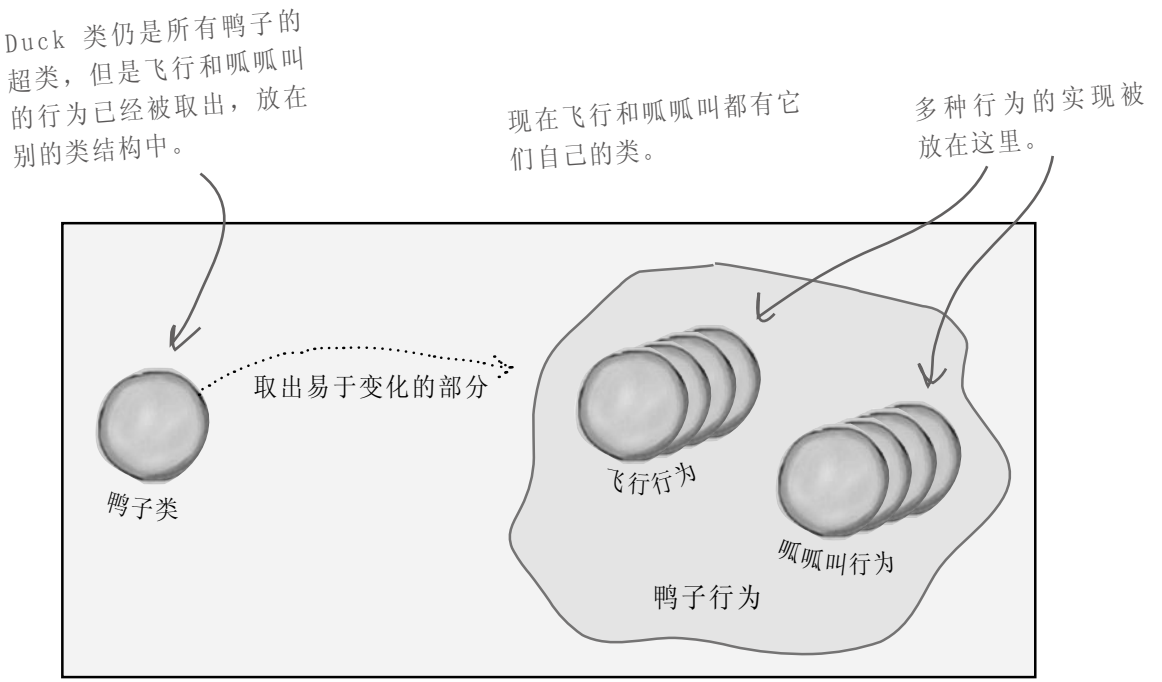
分开变化和不会变化的部分

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。

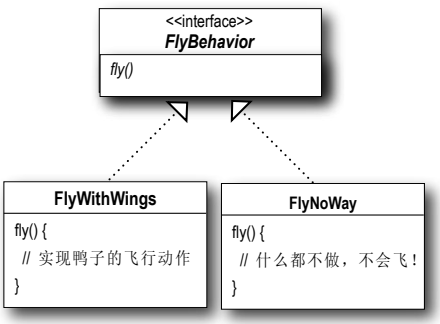
所以这次鸭子类不会负责实现Flying与Quacking接口，反而是由其他类专门实现FlyBehavior与QuackBehavior，这就称为「行为」类。由行为类实现行为接口，而不是由Duck类实现行为接口。

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）。

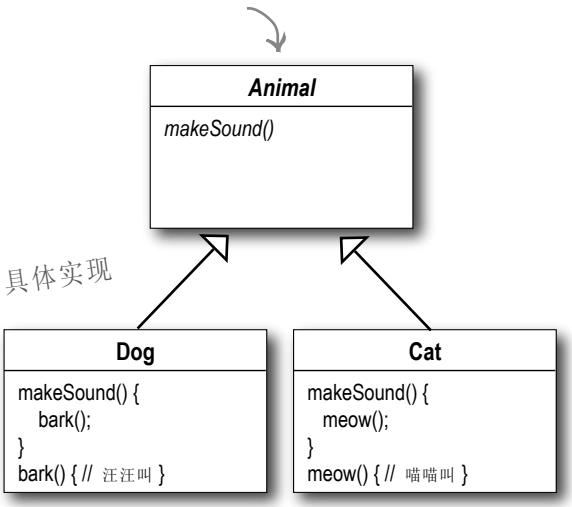
在我们的新设计中，鸭子的子类将使用接口（FlyBehavior与QuackBehavior）所表示的行为，所以实际的「实现」不会被绑死在鸭子的子类中。（换句话说，特定的实现代码位于实现FlyBehavior与QuakcBehavior的特定类中）。

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。

这样，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。「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。

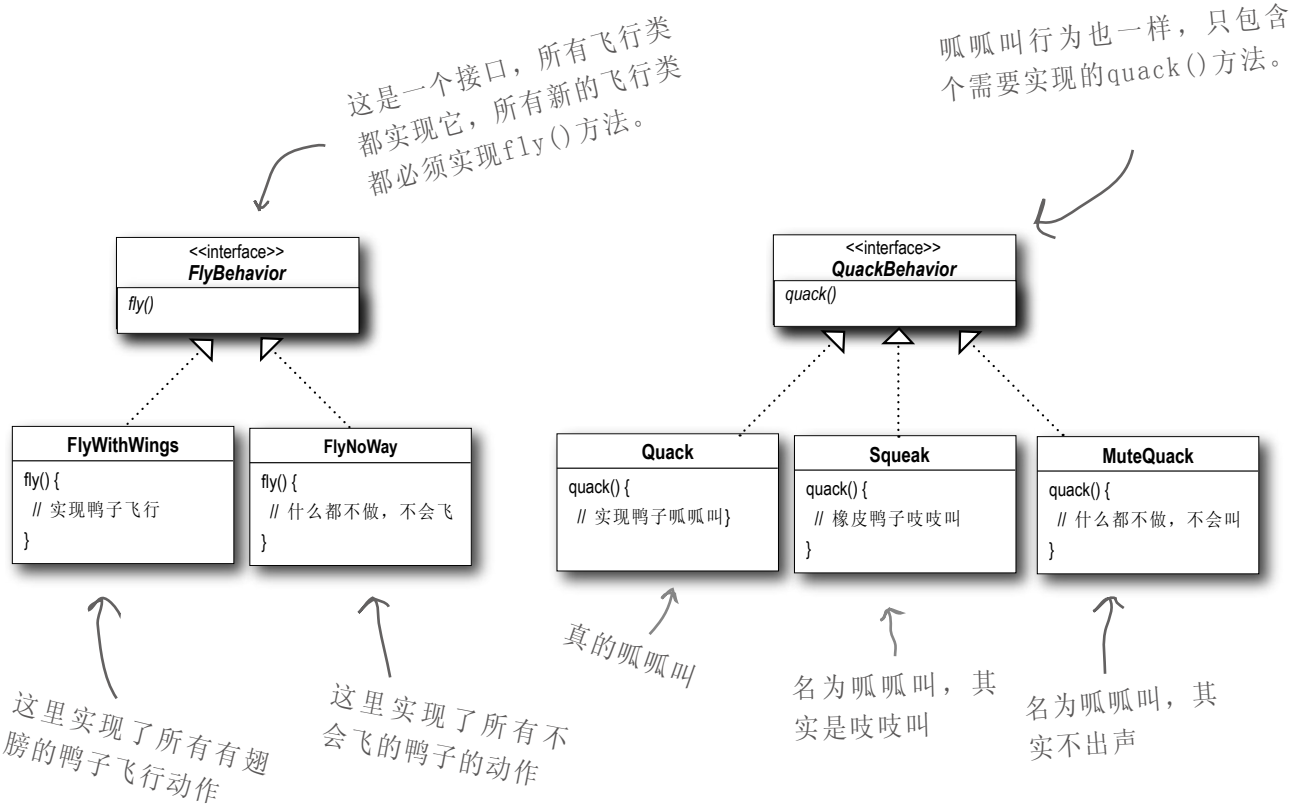
更棒的是，子类型实例化的动作不再需要在代码中硬编码，例如new Dog()，而是「在运行时才指定具体实现的对象」。

```
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。

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。

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）

整合鸭子的行为

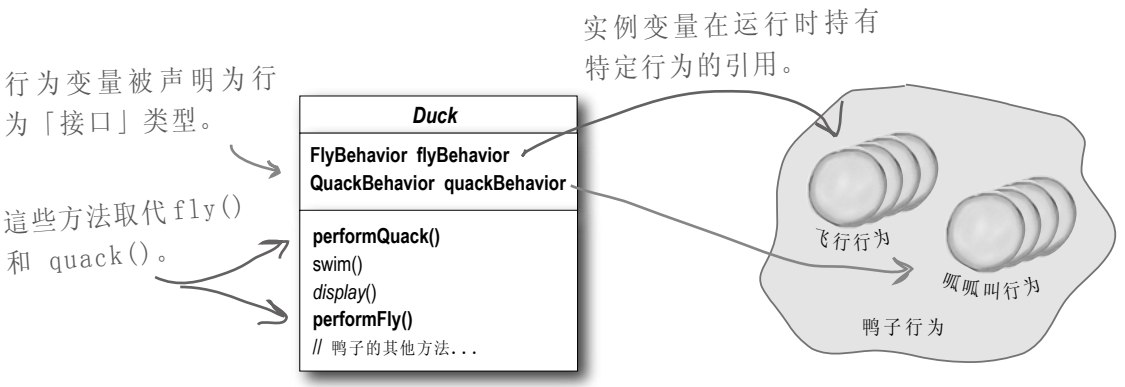
关键在于，鸭子现在会将飞行和呱呱叫的动作，「委托」（delegate）别人处理，而不是使用定义在自己类（或子类）内的方法。

作法是这样的：

❶ 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。



❷ 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，而是委托给quackBehavior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();
   }

   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，就把责任委托给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）。

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，你就会看到做这件事的代码。）

测试Duck的代码

❶ 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

❷ 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）

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！

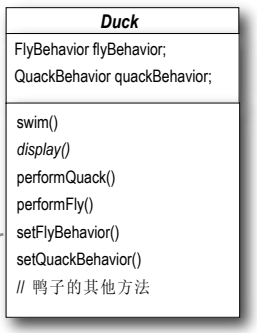
```
File Edit Window Help Yadayadayada
%java MiniDuckSimulator
Quack
I'm flying!!
```

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。

❶ 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。

❷ 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。

❸ 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!");
    }
}
```

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

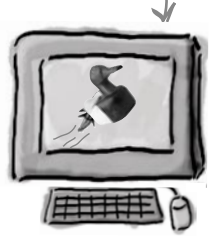
```
public class MiniDuckSimulator {
    public static void main(String[] args) {
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();

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

改变前

第一次调用performFly()会被委托给flyBehavior对象（也就是FlyNoWay对象），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。哇咧！模型鸭突然具有火箭动力飞行能力。

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。



改变后

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。

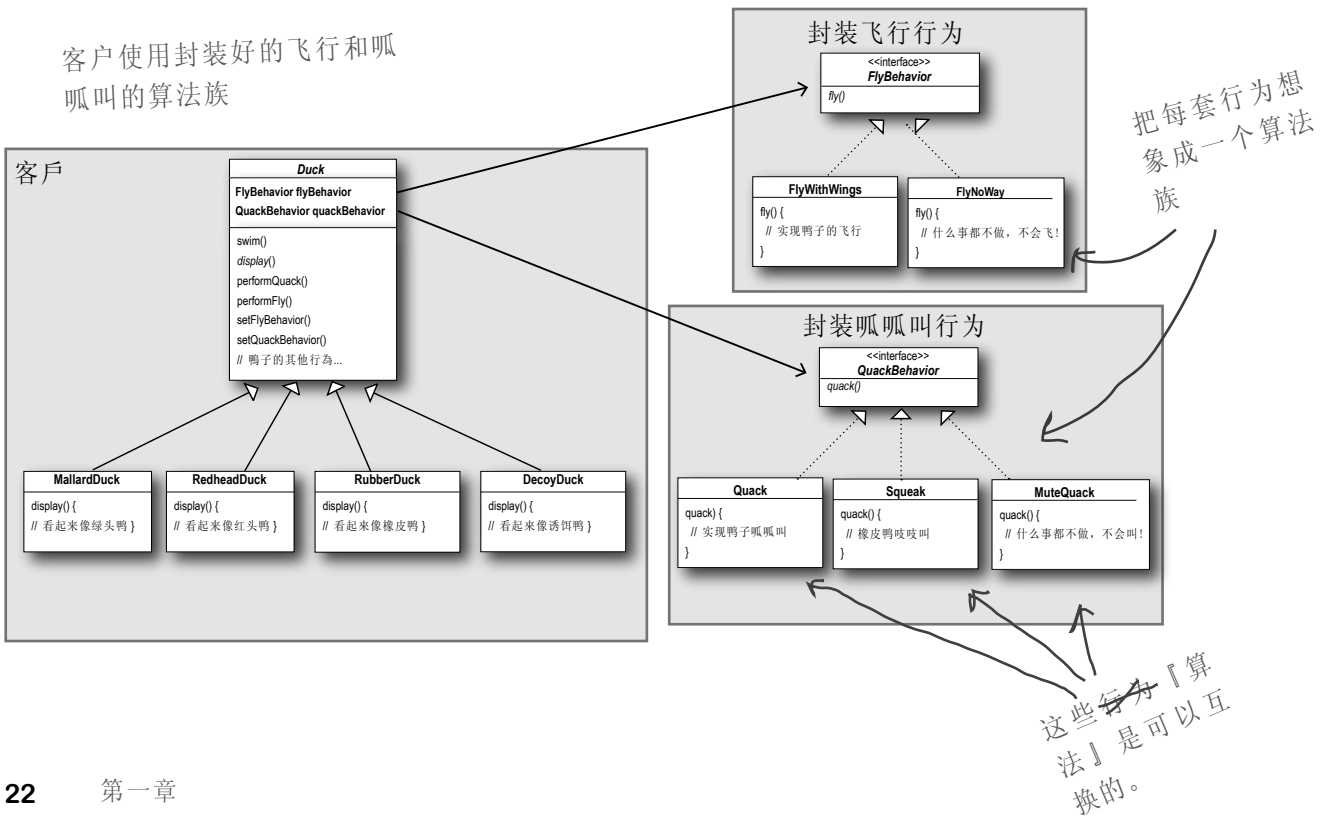
封装行为的大局观

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结构，你所期望的一切都有：鸭子继承Duck，飞行行为实现FlyBehavior接口，呱呱叫行为实现QuackBehavior接口。

也请注意，我们描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「一族算法」。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），这样的作法也能用于用一群类计算不同州的销售税金。

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）。



『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，面向对象之路承诺了『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砍竹子一样。

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展性上的复用程度呀？

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。

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独立于使用算法的客户。

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。

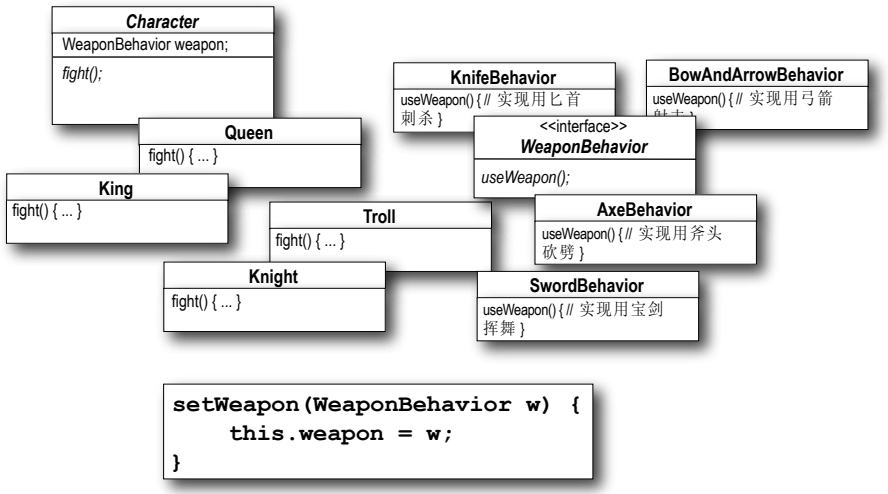
设计谜题

在下面，你将看到一堆杂乱的类与接口，这是取自一个动作冒险游戏。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。



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，一旦懂得这些词汇，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。

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移除。这样的设计方式相当动态和松耦合。

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需要共享词汇？（暗示：想一想汽修、木工、料理、航管）利用这些行话进行沟通的质量如何？

你能否想到OO设计的什么方面，能够和模式名称匹配的？「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...

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『行话』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社区。

共享词汇能帮助初级开发人员迅速成长。

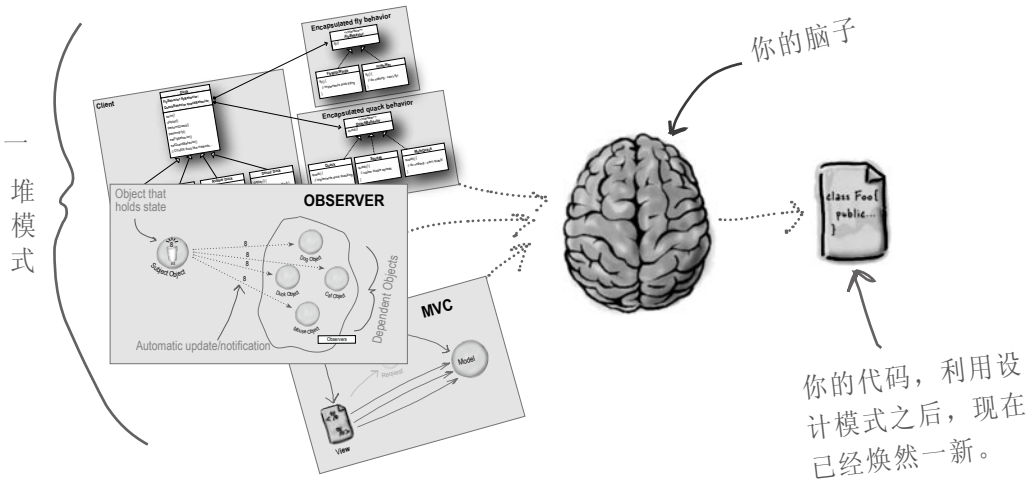
初级开发人员向有经验的开发人员看齐。当高级开发人员使用设计模式，初级开发人员也会有样学样。把你的组织建立成一个模式用户的社区。

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...

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有弹性时，可用它们重做旧代码。



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？

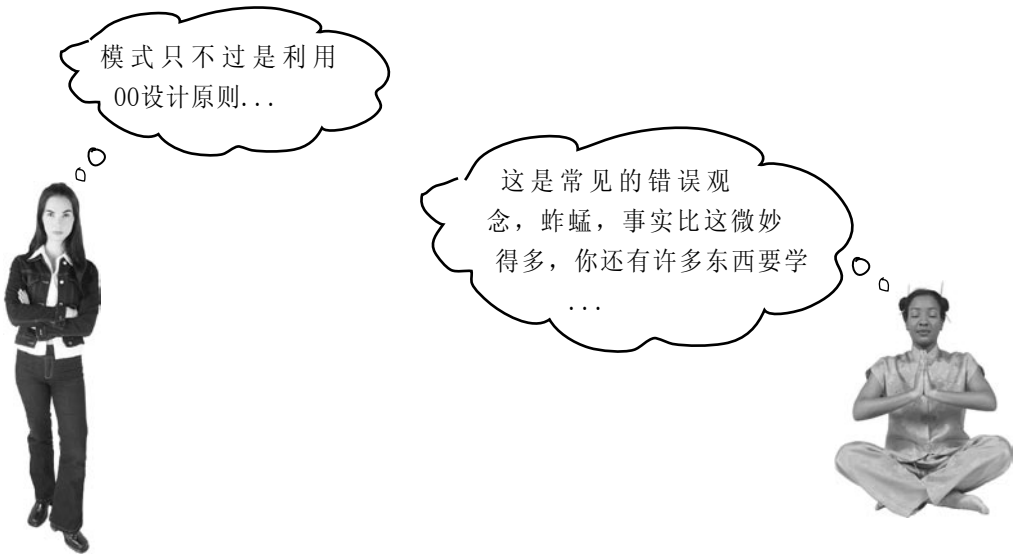
答：设计模式比库的等级更高。设计模式告诉我们如何组织类和对象，以解决某种问题。采纳这些设计并使它们适合我们特定的应用，是我们责无旁贷的事。

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类目。你可以在应用中利用这些设计模式。



怀疑的开发人员　　友善的模式大师

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，我真的还有必要考虑设计模式吗？运用00，一切不是都很直接吗？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？

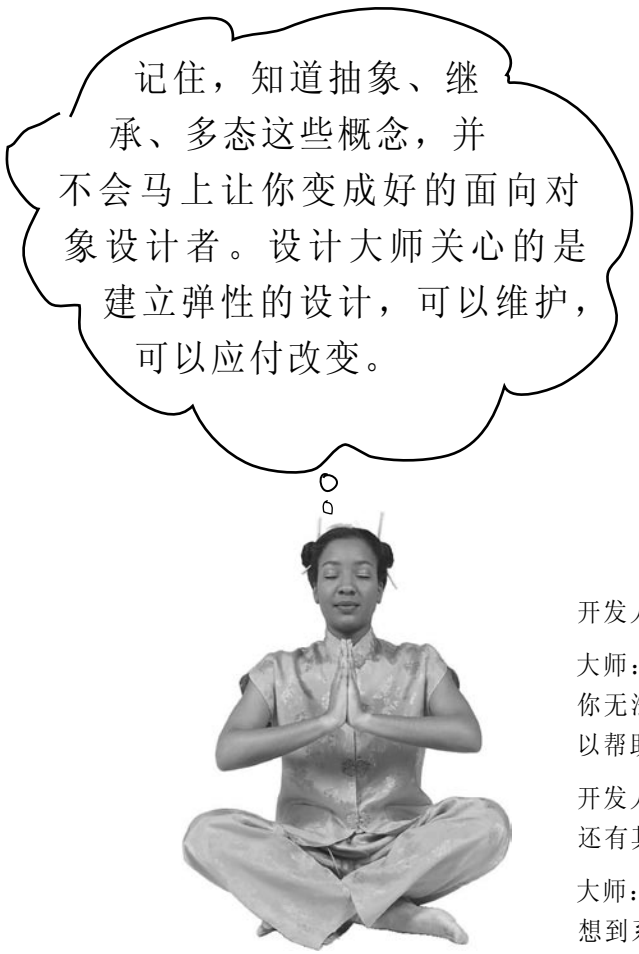
大师：不是！要构造有这些特征的00系统，事实证明只有通过不断地艰苦实践，才能成功。

开发人员：我想我开始了解了，这些构造00系统的隐含经验于是被收集整理出来...

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。

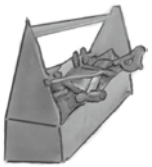


开发人员：如果我找不到模式，怎么办？

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以及应付变化的原则。



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

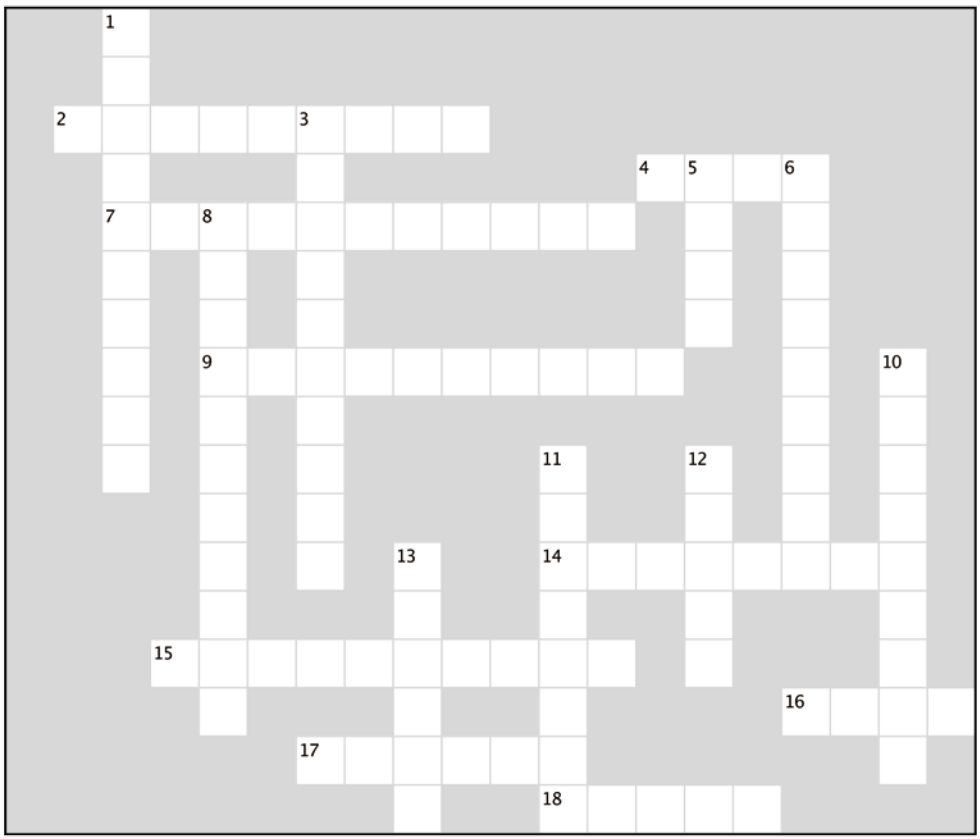
『策略』——定义算法族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化独立于使用算法的客户。

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。
- 模式可以让我们建造出具有良好00设计质量的系统。
- 模式被认为是历经验证的00设计经验。
- 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，都着眼于软件变化的主题。
- 大多数的模式都允许系统局部改变独立于其他部分。
- 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。

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

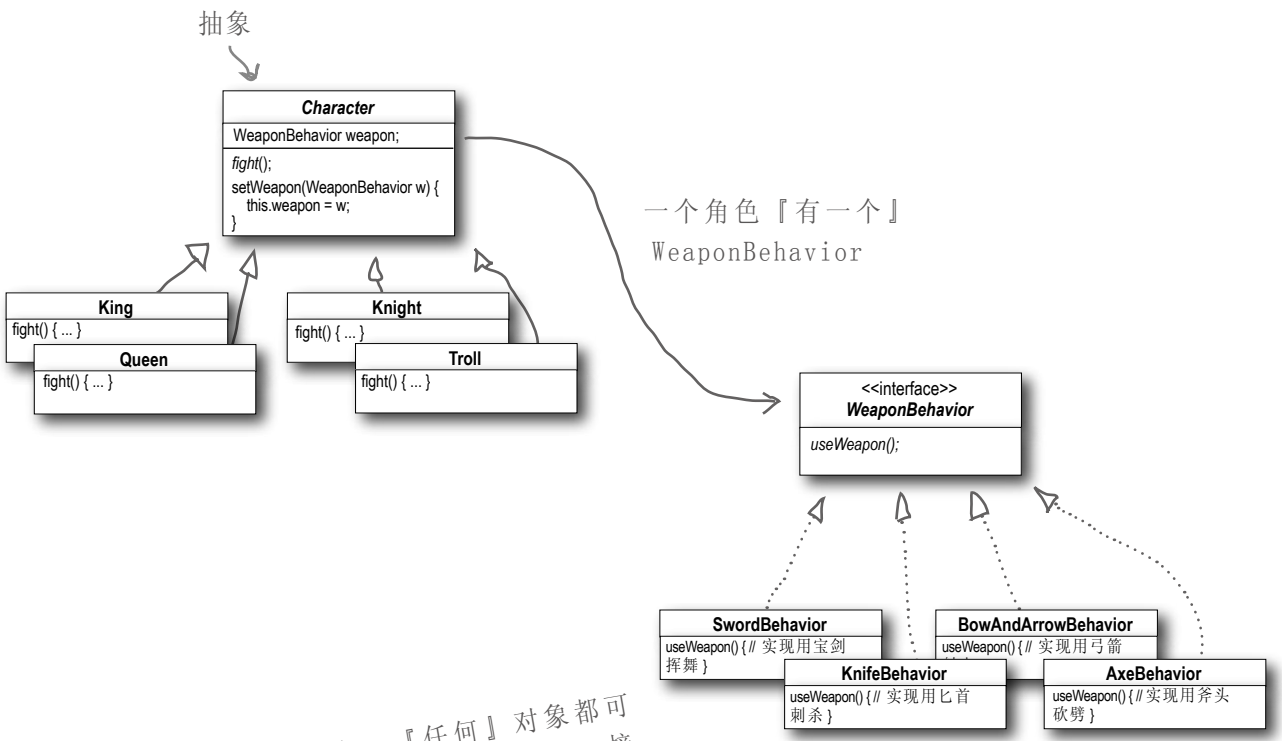
直排提示：

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。

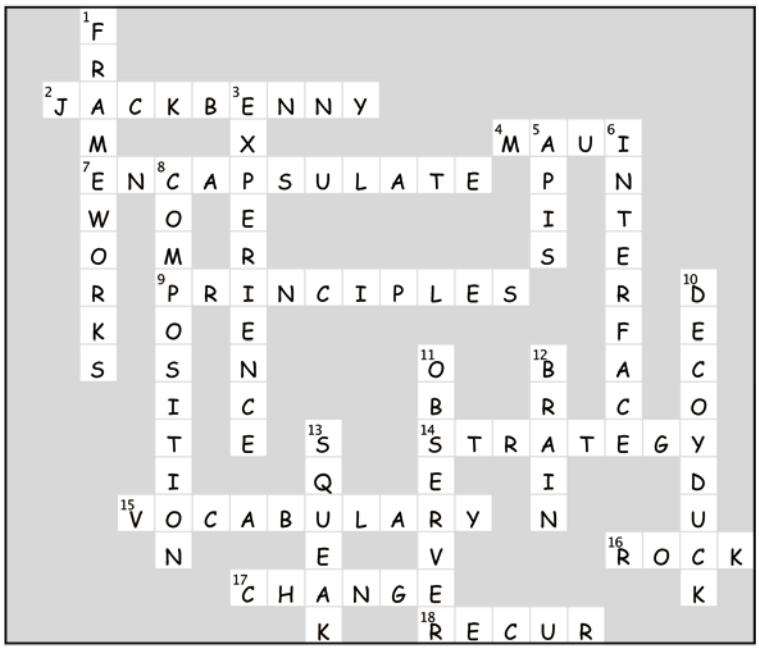


答案

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。

我们学到了足够的建造系统的知识，希望回头去把事情做得更好。

真實世界中的模式

現在你已經準備好迎接一個充滿設計模式的嶄新世界。但是，在你打開所有的機會大門之前，我們需要告訴你一些即將在真實世界中遇到的細節－沒錯，外面的世界比物件村來得複雜。來吧！從下頁開始，我們會指引你方向 ...

物件村指南 與設計模式相處

請接受我們的隨身指南，這裏提供了一些技巧，讓你能夠和真實世界的模式相處愉快。在這份指南中，你將會：

- 知悉所有對於「設計模式」定義的謬誤。
- 發覺有哪些琳瑯滿目的設計模式目錄，並認清為何你只需要買其中一本。
- 避免在錯誤的時間點使用設計模式的尷尬。
- 學習如何將模式維持在它所應屬的分類中。
- 體認到發覺新的模式並非只有專家做得到。閱讀過我們的祕笈之後，你也有機會寫下自己的模式。
- 親眼目睹神祕的四人幫身份被揭露。
- 能夠和鄰居有共同的話題－喝咖啡，聊模式。
- 學習著像東方禪師一樣訓練你的心智。
- 藉由改進你的模式語彙，結交一些朋友，並影響周遭的編程員。

定義設計模式

我敢說，在閱讀完這本書之後，你已經相當瞭解什麼是設計模式，但我們至今還從未給一個正式的定義。你可能會對於這個常用的定義感到驚訝：

> **模式**是在某情境（context）下，針對某問題的某種解決方案。

這個定義並不會讓人有恍然大悟的感覺，但是別擔心，我們現在就逐步瞭解定義中所提到的情境、問題、解決方案：

例如：你擁有一個物件的收集。

情境就是，採取某個模式的狀況。這應該是一個不斷出現的狀況。

你需要注意走訪每個物件，而且不需理會該收集的實踐方式。

問題就是，你想在某情境下達到的目標，但也可以是某情境下的限制。

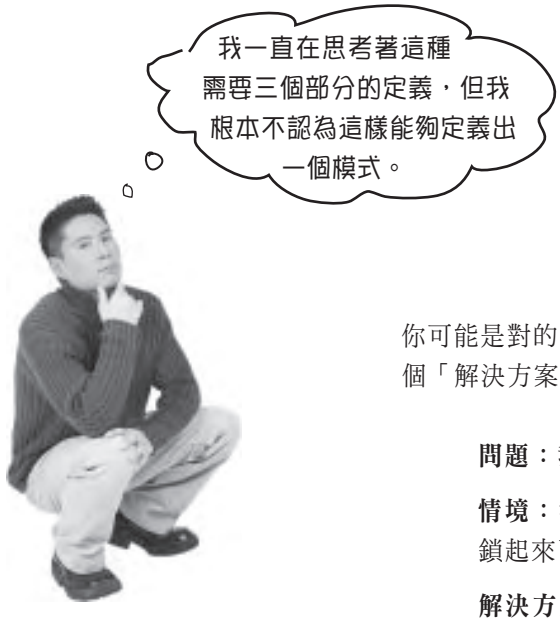
解決方案就是，你所追求的：一個一般性的設計，用來解決限制、達到目標。

將反覆的動作封裝進個別的類別中。

這是一個定義，需要花些時間逐步理解。下面有個幫你記憶的方法：

「如果你發現處於某個情境下，面對著一群限制的問題，正影響著所欲達成的目標，然而，你能夠採用某個設計，克服這些限制並達到該目標，將你領向某個解決方案。」

現在，想搞清楚什麼是設計模式，似乎就需要費點功夫。畢竟，你已經知道一個設計模式是解決一個常見的設計問題。將這一切搞得如此地拘謹，究竟所為何來？這個嘛，等一下你就知道，我們採用一種正式的方式描述模式，就能為模式建立出「型錄」，這個型錄能帶來各種層面的好處。



你可能是對的；讓我們再多想一想 … 我們需要一個「問題」、一個「解決方案」、一個「情境」：

問題：我要如何準時上班？

情境：我將鑰匙留在車子內，並將車子鎖起來了。

解決方案：打破窗戶，進入車內，啟動引擎，然後開車上路。

在定義中所需要的三個部分我們全都有了：我們有一個問題，這個問題牽涉到所欲達成目標，以及時間距離的限制，可能還有其他的影響因子；我們也具有一個情境，也就是車鑰匙被反鎖在車內；我們也有一個解決方案，讓我們能夠取得鑰匙，解決時間和空間的限制。既然這三個部分都有了，我們也就等於有了一個模式，對吧？

動動腦

我們依據設計模式的定義，定義了一個問題、一個情境、以及一個解決方案（而這個方案是行得通的！）這是一個模式嗎？如果這還不算是一個模式，那麼到底是為什麼？當我們試圖定義一個 OO 設計模式的時候，也有可能定義失敗嗎？

更仔細地觀察設計模式的定義

我們的這個例子似乎符合設計模式的定義，但這其實不算是一個真正模式。怎麼說呢？其一，我們知道模式必須要能處理一個再三出現的問題。雖然一個心不在焉的人，可能老是把車鑰匙反鎖在車內，但是一再地打破車窗，這實在稱不上是一個解決方案。

除了上述之外，它在某些方面也不符合規定：首先，別人想要在自己的特殊問題上，採用這個解決方案並不容易。其次，它也違反了模式所應該具備的一個重要而簡單的原則：沒有一個名字！如果沒有名字，就無法變成程式員之間共通的語彙。

幸運地，模式並非只是被描述成簡單的問題、情境、以及解決方案；我們有更好的方式能描述模式，並將它們收錄進「模式型錄」。

問：模式的描述是否由一個問題、一個情境、以及一個解決方案所構成？

答：通常模式描述會被放在模式型錄中，模式描述通常不只是問題、情境、以及解決方案。我們很快就會看到模式型錄的細節；模式型錄是標示某個模式的意圖、動機、可能採用該模式的地方、還有解決方案的設計以及後果（好的或壞的）。

問：稍微改變某個模式的結構以符合我設計上的需求，這樣可以嗎？或者是一定要遵照嚴格的定義？

答：當然你可以改變。模式不是法律或準則，模式只是指導方針，你可以改變模式來符合你的需求。我們也說過，真實世界中的許多實例，都沒有符合經典的設計模式。

然而，當你在採用模式的時候，最好能夠在文件上面註明與經典的設計模式有何差異，如此一來，其他的開發者就能夠很快地認出這個模式，並了解兩者的差異。

問：我要從那裏取得模式型錄？

答：第一個，也是最重要的一個設計型錄是由 Gamma、Helm、Johnson、Vlissides 所著作的《Design Patterns: Elements of Reusable Object-Oriented Software》（Addison-Wesley 出版）。這個型錄列出 23 個基本的模式，再過幾頁我們就會談到這本書。

還有許多其他模式型錄，將焦點放在不同的領域（例如：企業軟體、即時系統、使用者介面）。

技客的玩意兒

願力量與你同在

設計模式的定義告訴我們，問題包含了一個目標和一組限制。模式大師們對此有個稱呼，將其稱為「力量」。為什麼？這個嘛，我們確信他們有自己的理由，但是如果你是星際大戰迷，應當不會忘記：力量「型塑並控制了宇宙」。類似地，模式定義中的力量也型塑並控制了解決方案。只有當解決方案在力量的兩邊中取得平衡時（光明面是你的目標，黑暗面是這些限制），這才算是有用的模式。

當你第一次在模式的討論中看到這個叫作「力量」的術語，可能感到很困惑，但只要記得，力量有兩個方向（目標和限制），而且需要力量平衡才能夠建立一個模式的解決方案。別讓這個行話阻撓了你，願力量與你同在！

法蘭克：吉姆，也算我們一份吧，我們東拼西湊的看了些文章，才學了一招半式的模式。

吉姆：沒問題，每個模式的型錄都包含了一組模式，也描述了模式之間關係上的細節。

喬伊：你是說模式的型錄不只一份？

吉姆：當然；有些型錄是基礎的設計模式，有些則是特定領域的模式，例如 EJB 模式。

法蘭克：你在看的是哪一份型錄？

吉姆：這是經典的四人幫型錄；包含了 23 個基礎的設計模式。

法蘭克：四人幫？

吉姆：沒錯，四人幫是四個作者的簡稱，他們合作寫了第一本設計模式的型錄。

喬伊：這個型錄有些什麼？

吉姆：有一組關聯模式。每個模式的描述方式都遵照一個樣板，並闡述該模式的許多細節。比方說，每個模式都有一個「名稱」。

法蘭克：哇塞！模式還有名稱，真不得了！

吉姆：別小看名稱，法蘭克；事實上，名稱可重要的呢！當每個模式都有一個名稱的時候，我們談論模式就相當容易；也就是說，大家有一個共同的語言。

法蘭克：好啦！好啦！我只是在開玩笑。繼續說吧，還有些什麼？

吉姆：就像我所說的，每個模式都要遵照一個樣板。每一個模式都有名稱，和幾節完整的敘述。例如，有一節叫做 Intent（意圖），描述該模式是什麼，有點像是定義。然後還有一節叫做 Motivation（動機）以及 Applicability（適用性），描述何時何地該使用此模式。

喬伊：那麼關於設計呢？

吉姆：有幾節是描述類別圖內的所有的類別，以及每個類別扮演的角色。有一節描述如何實踐此模式，而且通常有範例程式碼，讓你知道為什麼。

法蘭克：聽起來好像面面俱到。

吉姆：還不只這樣。還有一些範例告訴我們在真實的系統中，此模式會使用在何處。除此之外，我認為最有用的小節之一在於：此模式和其他的模式之間有何關聯。

法蘭克：噢！你的意思是說他們有提到狀態和策略之間的差異？

吉姆：沒錯！

喬伊：那麼吉姆，你到底要如何使用這個型錄？當遇到問題時，你會去翻閱這份型錄，找到適當的解決方案嗎？

吉姆：首先，我試著讓自己熟悉所有的模式以及它們之間的關係。然後，當我需要一個模式的時候，大概就知道是什麼模式。我會參考描述動機和實用性的小節，確認我的想法沒錯。還有一個很重要的小節：後果。我瀏覽此模式的「後果」，確保此模式不會對我的設計帶來某些不好的影響。

法蘭克：聽起來很有道理。一旦你知道這個模式是正確的，究竟要如何應用到你的設計中，並實踐之？

吉姆：這就是為什麼需要類別圖。我先是閱讀「結構」這一節，以瞭解類別圖，然後看「參與者」這一節，確定我瞭解每一個類別的角色。接下來，就可以開始進行自己的設計，做出任何更改符合我自己的需求。並繼續閱讀「實踐/範例程式」小節，以確認我知道所有實踐上的技巧。

喬伊：現在我終於瞭解型錄如何真正的幫我加快使用模式的腳步。

法蘭克：吉姆，你能帶我們瀏覽一遍模式的敘述嗎？

型錄中所有的模式都是以一個「名稱」開始的，名稱是模式中很重要的一部份—如果沒有好名稱，那麼該模式就無法成為開發者之間共同的語彙。

這是模式的分類或類目，再過幾頁就會詳細描述。

「動機」具體地描述問題以及如何解決這個問題。

「意圖」簡短地描述該模式的作用。你也可以把它想成是模式的定義（就如同本書中的模式定義一般）。

「適用性」負責敘述此模式適合在哪些狀況下被採用。

「結構」提供了圖示，顯示出參與此模式的所有類別之間的關係。

「參與者」描述在此設計中所牽涉到的類別和物件，其責任和角色為何。

「合作」告訴我們參與者如何在此模式中互相支援。

「後果」描述採用此模式之後可能有的效果；好的與不好的。

「實踐」提供了你所需要使用的技巧，以及你應該小心面對的問題。

「範例程式」提供程式碼的片段，可能對你的實踐有所幫助。

「已知用途」用來描述，在真實的系統中，有哪些運用此模式的實際例子。

「相關的模式」描述了此模式和其他模式之間的關係。

SINGLETON — Object Creational

Intent

Et aliquat, velesto ent lore feuis acillao rperci tat, quat nonsequam il ea at nim nos do enim qui eratio ex ea faci tet, sequis dion utat, volore magnisi.

Motivation

Et aliquat, velesto ent lore feuis acillao rperci tat, quat nonsequam il ea at nim nos do enim qui eratio ex ea faci tet, sequis dion utat, volore magnisi.Rud modolore dit laoreet augiam iril el dipis dionsequis dignibh euммy nibh esequat. Duis nulputem ipisim esecte conullut wissi.

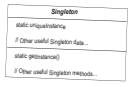
Os nisissenim et lumsandre do con el utpatuero corercipis augue doloreet luptat amet vel iuscidunt digna feugue dunt num etummy nim dui blaor sequat num vel etue magna augiat.

Aliquis nonse vel exer se minissequis do dolortis ad magnit, sim zzrillut ipsummo dolorem

Applicability

Duis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er

Structure



Participants

Duis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er

- A dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er
  - A feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissec
  - Ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit

Collaborations

- Feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore.

Consequences

Duis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre:

1. Dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.
2. Modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem.
3. Dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.
4. Modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem.

Implementation/Sample Code

DuDuis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

Nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Known Uses

DuDuis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

DuDuis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er. alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Related Patterns

Elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er. alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

問：有可能創建自己的設計模式嗎？還是只有「模式大師」才辦得到？

答：首先，請務必牢記在心，模式是被「發現的」，而不是被創建的。所以，任何人都可能發現某個設計模式，然後將它寫成一個敘述；然而，這並非唾手可得的事情，也不常發生。想成為一個「模式作家」，是需要全力以赴的。

你應該先想想看為何你想發現自己的模式－大多數的人都不是「撰寫」模式，只是單純的使用模式。然而，你可能是在某一個特定的領域中工作，而你認為新的模式將大有幫助，或者是你找到一個解決方案，能夠解決一個再三出現的問題，或者，你只想要加入設計模式的社群貢獻己力。

問：我有意願；我要如何開始？

答：就像任何守則一樣，你知道得越多越好。先研究已經被發現的這些模式，了解他們做了些什麼，並弄清楚他們和其他模式之間的關係。這些準備工作非常重要，不但可以讓你熟悉模式是如何打造出來，也可以避免做多餘的工作。完成這些準備工作之後，你可以開始將你的模式寫在紙張上，以便與其他開發者溝通之用；我們稍後將針對「如何溝通你的模式」多談一些。如果你真的非常感興趣，可以閱讀下一頁以後的內容。

問：我怎麼知道我是否真的有一個模式？

答：這個問題很好：除非其他人使用它並且發現它很有用，否則你並不算擁有一個模式。一般來說，必須要通過「三次規則」，才算是一個合格的模式。也就是說，只有在真實的世界中被採用三次以上，才能算是一個規則。

您想要當一個設計模式的巨星嗎？

那麼，聽清楚了。

先取得一份模式的型錄，

然後花些時間好好地學習它。

當你寫下一個正確的敘述，

而且有三個開發者都同意你的看法時，

那麼你就成功了。

這一段文字也可以改成「你想要當一個搖滾巨星嗎？」

想當一個設計模式的作家嗎？

先做好家庭作業。
在發掘新的模式之前，你必須先熟稔既有的模式。許多模式看起來像是全新的，但是事實上只是既有模式的變種。藉由研究既有的模式，你可以比較容易辨識出模式，並且知道某個模式和其他模式之間的關聯。

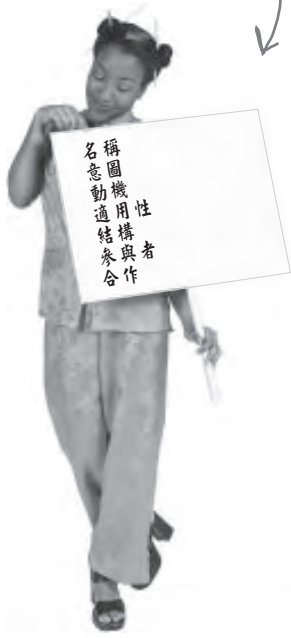
花時間自我檢視與評估。
你的經驗（你所遭遇過的問題、以及採取的解決方案）正是模式想法的來源。所以，花時間自我檢視過去的經驗，並將它用在以後發生的新設計上面。請牢記，大多數的模式都是既有模式的變種，而非嶄新的模式。而且當你真的找到了好像是新模式的東西時，常常都侷限在很窄的適用性中，而不太稱得上是一個真正的模式。

將你的想法寫在紙上，好讓其他人能夠理解。
如果其他人不能夠使用你所找到的模式，那麼這個新模式作用也就不大；你需要將你的「準模式」寫成一份文件，好讓其他人能夠閱讀、理解、並採用它來解決問題，然後將使用的心得回饋給你。很幸運地，你不需要為你的模式新創文件的撰寫方式，你可以直接採用四人幫的樣板。

讓其他人使用你的模式；然後再持續改進。
不要認為你可以一次就把模式搞定，應該要把模式當成是隨著時間不斷進步的一項工程。讓其他人檢視你的模式，並嘗試著使用它，然後將意見回饋給你。將這些意見彙整到你的敘述中，再持續上述的步驟。你的描述永遠不會是完美的，但是到了某個時間點之後，就會相當的穩固，足以讓其他的程式員能夠閱讀並理解它。

不要忘了三次規則。
請記得，除非你的模式已經成功地被採用在真實世界的三個例子中，否則就不夠資格被當成模式。當你的模式能夠在別人的手中使用，並將意見回饋給你時，就有機會能夠將它變成一個實用的模式。

使用已經存在的模式樣板定義你的模式，因為這些樣板已經包含了許多前輩的智慧，而且其他的模式使用者也比較習慣這樣的格式。



連連看

每個模式的敘述為何者

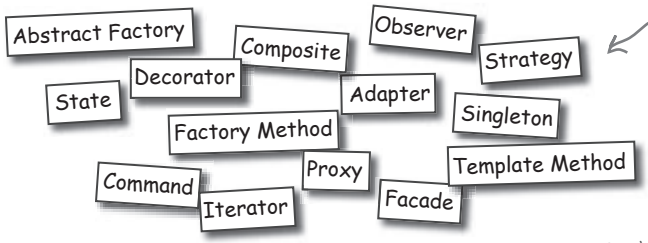
模式	敘述
裝飾者	將一個物件封裝起來，並提供一個不同的介面給它。
狀態	由次類別決定如何實踐一個演算法中的步驟。
反覆器	由次類別決定要建立的具象類別為何者。
表象	確保只有一個物件能夠被建立。
策略	將可以互換的行為封裝起來，並使用轉接方式決定要使用何者。
代理人	客戶面對物件的集合以及個別物件時，用一致的方式處理。
工廠方法	封裝了以狀態為基礎的行為，並使用轉介的方式在行為之間作切換。
轉接器	能夠在物件的集合之中游走，而且不需要暴露集合的實踐方式。
觀察者	將一群類別的介面予以簡化。
樣板方法	將一個物件包裝起來，以提供新的行為。
合成	允許客戶建立物件的家族，而無需指定他們的具象類別。
獨體	讓物件能夠在狀態改變時接獲通知。
抽象方法	將物件包裝起來，以控制對此物件的存取。
命令	將一個請求封裝起來，成為一個物件。

組織設計模式

隨著發掘的設計模式數目逐漸增加，有必要將它們分級分類，好將它們組織起來，以簡化我們尋找模式的過程，並讓同一群組內的模式互相比較。

在大多數的型錄中，模式通常根據某種做法被歸為幾類。最常用的分類方式，就是第一本模式型錄中所採用的方式，根據模式的目標分成三類：生成、行為、以及結構。

削尖你的鉛筆

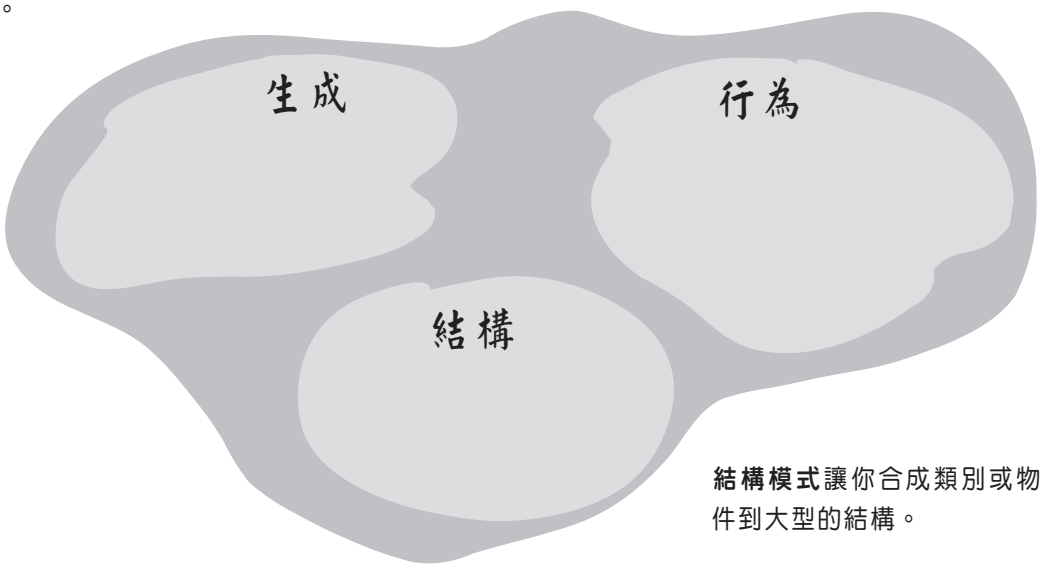


閱讀每類的敘述，試著將這些模式正確地歸類。這並不容易！但是請盡力而為。正確答案在下一頁。

這裏的每個模式都屬於底下類目之一。

生成模式牽涉到將物件實體化，這類模式都提供一個方法，將客戶從所需要實體化的物件中鬆綁出來。

只要是**行為模式**，它的重點都在於類別和物件如何互動，以及各自的責任。



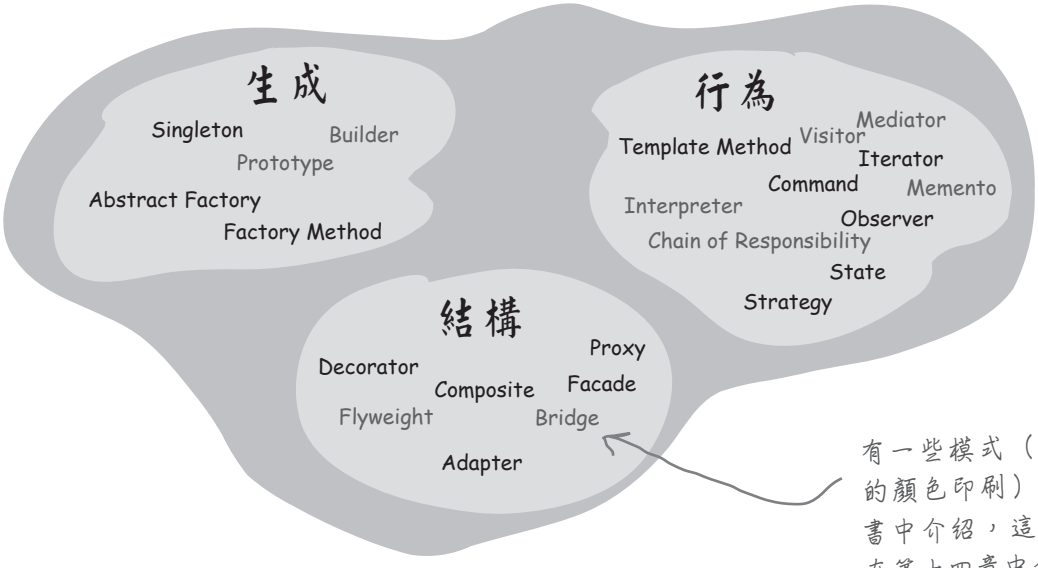
結構模式讓你合成類別或物件到大型的結構。

解答：模式分類

這是模式的類目，你可能覺得這個習題很困難，因為許多模式似乎不只符合一個類目，別擔心，其實每個人都有這樣的困擾。

生成模式牽涉到將物件實體化，這類模式都提供一個方法，將客戶從所需要實體化的物件中鬆綁出來。

只要是**行為模式**，它的重點都在於類別和物件如何互動，以及各自的責任。

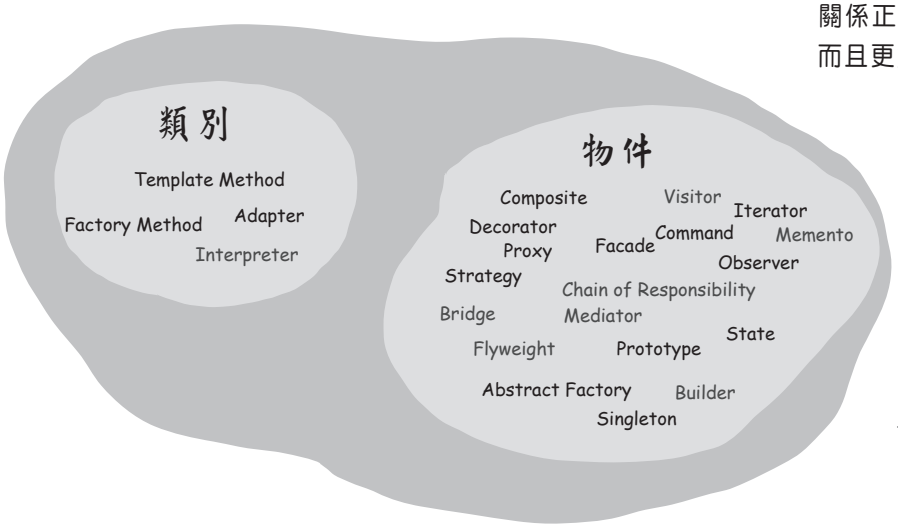


結構模式讓你合成類別或物件到大型的結構。

除了剛才的分類方式之外，模式還有另一種分類方式：模式所處理的是類別或物件：

類別模式會描述類別之間的關係如何透過繼承定義。類別模式的關係是在編譯時期就建立好的。

物件模式定義了物件之間的關係，而且物件模式主要是利用合成定義。物件模式的關係正常是在執行期建立的，而且更動態、更有彈性。



請注意，物件模式比類別模式的數目多出許多！

問：只有這些分類方式嗎？

答：不，還有人提出其他的分類方式。有些分類方式先分成三大類，然後再分成幾個小類（例如「鬆綁模式」）。你想熟悉的是最常用的分類方式。如果建立自己的分類方式，可以幫你了解這些模式的話，那麼你也可以這麼做。

問：將模式分成不同的類目，這麼做真的能夠幫助我們記憶這些模式嗎？

答：這麼做可以讓你對模式之間的比較有清晰的概念，這是無庸置疑的。但是許多人被生成、結構、以及行為模式搞得一頭霧水；常常發現某個模式似乎不只適合一個類目。請記住，怎麼分類並不重要，重要的是了解這些模式和它們之間的關係。只要類目有助於我們就用它，反之就不用。

問：為何裝飾者模式被歸類到結構類中？我認為它應該是行為類，畢竟這個模式用來新增新的行為！

答：是的，有許多開發者都這麼說！四人幫之所以這麼分類，他們的想法是這樣子的：結構模式用來描述類別或物件，如何被合成以建立新的結構或新的功能。裝飾者模式允許你透過「將某物件包裝進另一個物件的方式」，將物件合成以提供新的功能。所以焦點是在如何動態地合成物件，以獲取某功能，而不屬於行為模式的目的－在物件之間的溝通與聯繫。請牢記，這幾個模式的意圖並不相同，而這通常是了解某個模式屬於哪個類目時，重要的關鍵。

大師與門徒 ...

大師：蚱蜢啊！你看起來很苦惱。

門徒：是的，這是因為我正在學習模式的分類，而令我感到困惑。

大師：繼續說 ...

門徒：在學習這麼多模式之後，我被告知每個模式都屬於結構、行為，生成三種類目之一。為什麼我們需要為模式分類呢？

大師：我說蚱蜢，不管是在什麼時候，只要我們有一大堆東西，很自然地就會想要為它們分類，這可以幫助我們用更抽象的方式，思考這些東西。

門徒：大師，你能舉一個例子嗎？

大師：當然可以。就拿汽車來說；有許多種不同的汽車款式，我們很自然地把汽車分成幾類，例如：經濟車、跑車、休旅車、卡車、以及豪華轎車。

大師：蚱蜢，你看起來好像大吃一驚，難道你無法體會我說的話？

門徒：大師，我很能體會你說的話，我只是對於你如此地瞭解車子感到震驚！

大師：蚱蜢，畢竟不是所有的例子都適合使用蓮花或飯缽來舉例。現在，我能繼續說嗎？

門徒：是的，是的，很抱歉打斷你，請繼續。

大師：一旦你有了分類或類目，你就可以很方便地這麼說：「如果你想從矽谷開車到聖庫茲，那麼跑車將會是最好的選擇。」或者「因為石油儲存量在惡化中，所以應該購買經濟車，比較省油。」

門徒：所以透過分類，我們可以將一組模式視為一個群體。當我們知道需要一個生成模式，但又不知道確切是哪一個的時候，就可以用生成模式這個詞來統稱它。

大師：是的，而且分類也有助於我們比較相同類目內的其他項目，比方說，「迷你車是最有風格的小型車。」或者幫助我們縮小搜尋範圍，「我需要一部省油的車子。」

門徒：我瞭解了，所以我就可以說「轉接器模式在架構模式中，是最適合改變物件介面的模式」。

大師：是的，類目還可以開發新領域；比方說，「我們真的想要開發一部跑車，具有法拉利的性能以及國民車的售價」。

門徒：這種車聽起來就像是死亡陷阱。

大師：對不起，我沒聽清楚你說什麼。

門徒：唔！我是說「我懂了」。

門徒：所以類目可以讓我們思考模式群組之間的關係，以及同一組模式內模式之間的關係，還可以讓我們找出新的模式。但是，為什麼使用三個類目，而不是四個或五個？

大師：就像是夜晚天空中的星星一樣，你可以看見許多類目。三是一個恰好的數目，由許多人所決定出來的數目。但是有些人的確有建議用四個、五個、或更多個。

用模式思考

情境、限制、力量、類目、分類 ... 我的天，聽起來非常的學術呢！好吧，這一切都很重要，而知識就是力量。但是，當我們面對它，如果你了解學校學的東西，而沒有使用模式的經驗和實務，那麼這不會在你的生活中造成多大的差別。

下面是一份簡單的指南，可以幫助你開始「用模式思考」。所謂的「用模式思考」意思是說，能夠看到一個設計，並體會到哪些地方可以採用模式，而那些地方不應該採用模式。

你思考模式的腦袋

保持簡單（Keep It Simple / KISS）

首先，當你在設計時，儘可能地用簡單的方式解決問題。你的目標應該是要達到簡單性，而不是「如何在這個問題中採用模式」。千萬不要認為：如果沒用模式解決某個問題，就不是專家。如果你能夠保持簡單的設計，其他的開發者將會相當尊敬。正確的說法是，為了要讓你的設計簡單且有彈性，有時候需要使用模式。

設計模式非萬靈丹；事實上，連什麼丹都算不上！

如你所知道的，模式是用來解決一再發生的問題。模式已經被許多開發者實際測試過。所以，當你需要某個模式的時候，可以高枕無憂地使用它，畢竟你知道這個模式已經身經百戰。

然而，模式並非萬靈丹，你不能在採用一個模式之後，編譯這個程式，然後就早早地去吃午餐。要使用模式，你需要考慮到模式對你的設計中，其他部分所造成的影響。

你知道何時需要模式...

啊...這是最重要的問題：何時使用模式？當你在設計的時候，如果確定在你的設計中可以利用某個模式解決某個問題，那麼就使用這個模式！如果有更簡單的解決方案，那麼在使用模式之前應該先考慮這個方案。

如何知道何時該採用模式，這就需要經驗和知識。一旦你確定某一個簡單的解決方案，無法達到需求，應該考慮這個問題以及相關的限制－這可幫你將問題對應到一個模式中。如果你對於模式有很深的認知，就可能知道有甚麼模式適合這樣的狀況。否則，就花些時間調查一下。你可能需要瀏覽模式類目中，對於意圖和應用的描述，這或許有幫助。一旦找到了一個適合的模式，要先確定這個模式所帶來的後果，以及對系統的影響是你所能夠忍受的。如果一切看起來都很好，就用它吧！

有一種狀況，即使有更簡單的解決方案，你仍然想要使用模式，這種狀況就是：當你預期系統在未來會改變時。我們也看過了，找出你的設計中會改變的地方，通常正是需要採用模式的地方。但是務必要確定一件事：加入模式是要因應實際的改變，而不是假定（可能）的改變。

並非只有在設計時，你才考慮到模式，事實上，當你在程式重構（refactoring）的時候，也要考慮到模式。

重構的時間，就是模式的時間！

所謂的重構，就是改變你的程式碼，來改進它的組織方式。目標是要改善其結構，而不是其行為。這是一個很好的時機，可以重新檢視你的設計，是否能夠利用模式讓它擁有更好的結構。比方說，程式碼內如果充滿了條件敘述，這可能意味著需要使用狀態模式，或者意味著，應該利用工廠模式，將這些具體的關係消除掉。許多書都著墨在如何利用模式進行重構，而隨著你的能力增強，依需要多多地涉獵這個領域。

拿掉你所不需要的，不要害怕從你的設計中，將某些設計模式移除。

還沒有人談到何時應該將某個模式移除，你可能認為這很難啟齒！不，我們都是大人了；應該要面對這個問題。

所以何時你應該移除某個模式呢？當你的系統變得非常複雜，而且並不需要預留任何彈性的時候，就不要使用模式。換句話說，也就是當一個較簡單的解決方案，比使用模式更恰當的時候。

如果你現在不需要，就別做。

設計模式威力很強大，你很容易就可以在目前的設計中，看到許多模式被採用。開發者本來就傾向於建立漂亮的架構，能夠因應未來任何方向的改變。

要抗拒這樣的誘惑呀！如果你今天有實際的需要去支援改變，就放手採用模式處理這個改變吧！然而，如果說理由只是假想的，就不要加進這個模式，因為採用越多模式，便有機會將你的系統越搞越複雜，而其中許多模式可能永遠無法為你帶來任何好處。

大師與門徒 ...

大師：蚱蜢，你的基礎訓練幾乎完成了，接下來的計畫是什麼？

門徒：我要去迪士尼樂園大玩特玩！然後開始利用模式建立許多程式碼！

大師：等等！你可別忘了「殺雞焉用牛刀」的道理呀！

門徒：這是什麼意思呢，大師？我已經學了這麼多的設計模式，難道不應該將它們全部用在我的設計中，達到最強的威力、彈性、以及掌控性嗎？

大師：不，模式只是一種工具，只有在真正需要時才使用這種工具。你也花了很多時間學習設計守則。一開始總是先依循這些守則，建立最簡單的程式碼以完成工作，在這個過程中，你看到有需要模式的地方，就採用模式。

門徒：也就是說，我的設計並不是從模式開始？

大師：「大量運用模式」絕對不是你開始設計時所該有的目標，應該要讓模式在你的設計過程中，自然而然地出現。

門徒：既然模式這麼好，為什麼在使用它們的時候還得如此的小心？

大師：模式可能帶來複雜性，而我們不希望在程式中出現沒有必要的複雜性。相信你已經知道，模式是一種被証實過的設計經驗，可以避免某些常見的錯誤。模式也是一種共通的詞彙，能夠讓我們和其他的開發者之間，針對我們的設計溝通。

門徒：那麼，我們又如何知道何時應該採用設計模式呢？

大師：當你確信你的設計中有一個問題需要解決的時候，或者當你確信未來的需求可能會改變時，都可以採用模式。

門徒：雖然我已經了解了許多的模式，但我覺得我的學習應該要繼續下去。

大師：是的，蚱蜢。學習管理複雜度以及軟體的改變，這是一生的課題。但是現在既然你已經知道許多模式，就可以開始在需要的地方採用它們，並持續地學習更多的模式。

門徒：等一下，你是說我還沒有學完「全部」？

大師：蚱蜢，你已經學會一些基礎模式，你會發現還有更多的模式在等著你學習，包括了一些只有使用在特定領域的模式，例如同時系統（Concurrent System）和企業系統都有各自的特殊模式。現在你已經有了良好的基礎，學習這些模式就不會太難！

在模式的心智

初學者的心智

「我需要找個模式給 HelloWorld 程式使用。」

初學者到處都在使用模式。
這很好：可以藉此培養許多使用模式的實際經驗。初學者也認為「我使用越多模式，我的設計就愈好」。初學者將慢慢體認到事實並非如此，所有的設計都應該儘量地保持簡單。只有在需要擴充的地方，才值得使用模式。

中階者的心智

「或許在這裏我需要一個獨體模式。」

隨著學習的過程，中階者的心智開始能夠辨識何時需要模式，而何時不需要。
中階者的心智真的會企圖把過多模式套用在不適當的地方，但他們也開始察覺到有些模式並不適合目前的狀況。

悟道者的心智

「在這裏採用裝飾著模式相當自然。」

悟道者的心智能夠看到模式在何處能夠自然融入。
悟道者的心智並不急切于使用模式，而是致力於簡單的解決方案。悟道者的心智會思索著物件的守則，以及它們之間的折衷取捨。當自然地出現模式的需要時，悟道者的心智就拿捏得宜地採用模式。悟道者的心智也能看到相似的模式之間的關係，以及它們之間在意圖上微妙的差異。悟道者的心智不會讓這些模式的知識，過度影響了設計的決策。

衛生署警告：過度使用設計模式可能導致程式碼被過度工程化。經常用最簡單的解決方案完成工作，並在真正需要模式的地方才使用。

當然我們希望你使用設計模式！

但是我們更希望你能夠成為一個良好的物件導向設計者。

當一個設計解決方案決定要使用某個模式的時候，將為你帶來好處，因為任何模式都是身經百戰，驗証能夠解決此問題的。模式具有良好的文件，而且容易被其他開發者所了解（你知道的，模式是開發者共同的語彙）。

然而，當您使用設計模式的時候，仍然有缺點。設計模式常常產生一些額外的類別和物件，所以會增加設計的複雜度。設計模式也能夠在你的設計中，加入更多層級（layer），這非但複雜，且效率差。

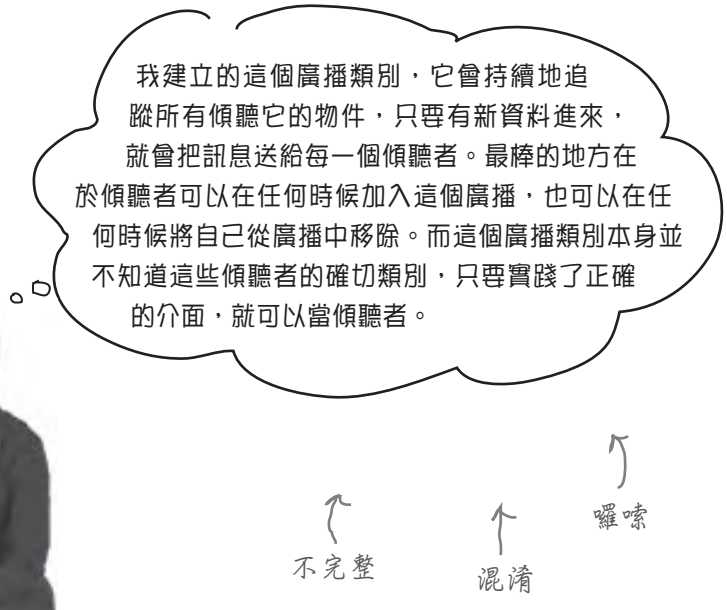
還有，使用設計模式，有時候大材小用。有時候你可以回頭看看設計守則，然後將發現，有更簡單的解決方案能解決相同的問題。果真如此，可別抗拒。就用較簡單的解決方案吧！
話雖如此，我們並非鼓勵你不要用模式。當設計模式運用得宜，好處其實非常多。

別忘了共通語彙的威力

在這本書中，我們花了相當多的時間在討論物件導向的基礎知識，但可別忘了設計模式對於人類層面的影響－設計模式不只可以幫助你在腦袋中裝進這些解決方案，也可以讓你和其他開發者之間有共同的語彙，而這正是設計模式最大的優點之一。

想想看，從上次我們談到共同語彙至今，有些事情已經不一樣了；你現在已經開始建立了自己的某些語彙！更別說學會了一整組的物件導向設計守則，而透過這些設計守則，你能輕易地了解所遇到的任何新模式之動機和運作方式。

現在你已經有了設計模式的基礎，應該「把模式傳出去」，讓大家都知道。為什麼呢？因為當你的同伴開發者，也知道這些模式並和你使用相同的詞彙，將使得你們的設計更好，更容易溝通。



共通語彙的五種方式

1. **在設計會議中**：當你和你的團隊在會議中討論軟體設計時，使用設計模式可以幫你們待在「設計中」久一點。從設計模式以及物件導向守則的觀點，討論設計，可以避免你的團隊很快地陷入實踐的細節，也可以避免發生許多誤解。
2. **和其他開發者**：當你和其他開發者討論的時候，可以使用模式。這可以幫助其他開發者學習新模式，並建立一個社群。和別人分享你所學會的東西是很有成就感的一件事情。
3. **在架構文件的時候**：當你在寫架構文件的時候，使用模式將會縮減文件的篇幅，一定讓讀者更清楚地了解你的設計。
4. **在程式碼註解以及命名慣例上**：當你在寫程式碼的時候，在註解中清楚地註明你所使用的模式，還有在選擇類別和方法的名稱，盡可能地揭露出對應的模式。其他的開發者在閱讀你的程式碼時，將很感激你，他們也能夠很快地了解你的程式碼內容。
5. **將志同道合的開發者聚合在一起**：分享你的知識。許多開發者都聽過模式，但並不真正瞭解什麼是模式。你可以自願為他們講一堂模式介紹，或者成立一個讀書會。

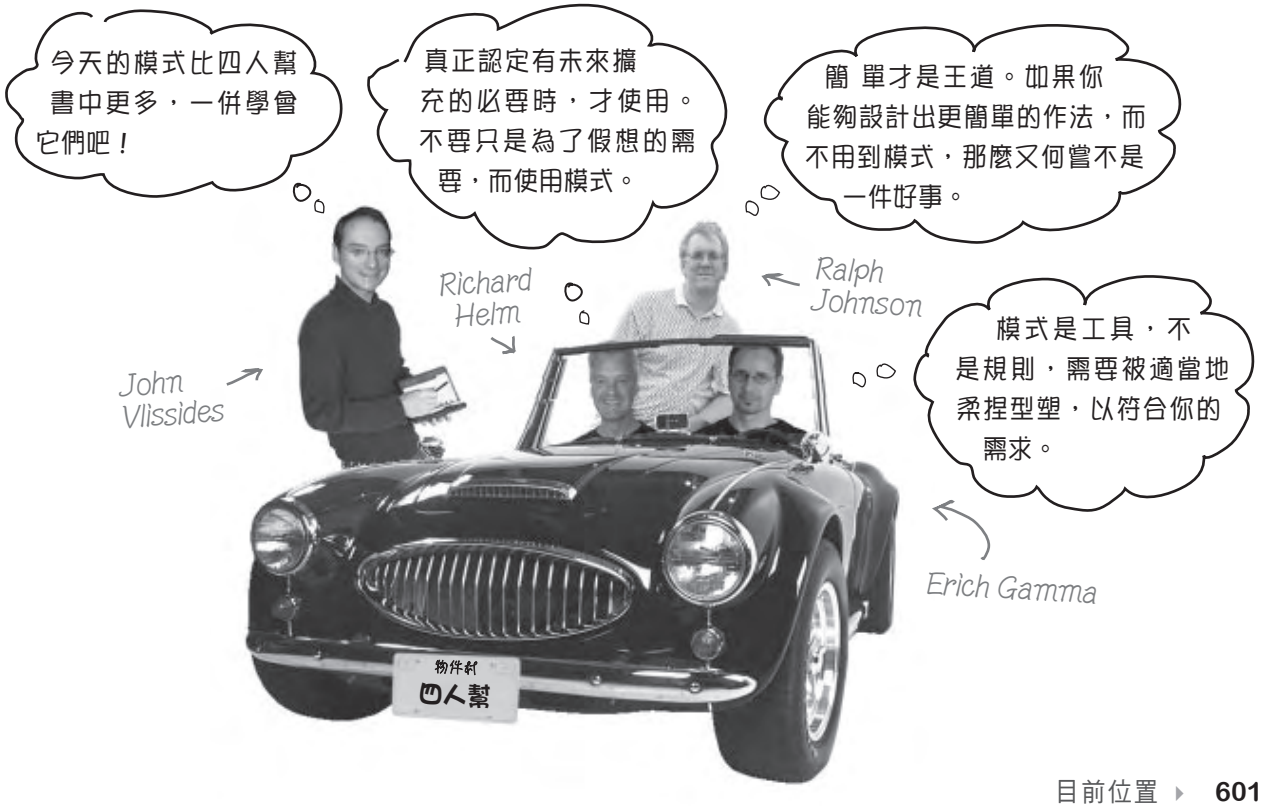
和四人幫一同巡迴物件村

在物件村內，你不會遇到閃亮三姐妹，但是你有機會遇到四人幫。你大概也注意到了，想要在模式的世界中走得夠遠，你就一定會遇到他們。那麼，到底這個神祕的幫派是怎麼一回事？

簡單地說，四人幫包括了 Erich Gamma、Richard Helm、Ralph Johnson、以及 John Vlissides。他們是第一群人將這些模式歸納出來的功臣，正因為他們此舉，造成了軟體領域的一大躍進。

這個稱號又是怎麼來的？沒有人知道，反正大家都是這麼稱呼。但是想想看：如果你想成為「幫派成員」，好好地認識物件村，那麼該怎麼辦呢？幸好，他們同意帶我們一同去巡迴物件村 ...

四人幫發起了軟體模式，隨後有許多人也做出很大的貢獻，包括了 Ward Cunningham、Kent Beck、Jim Coplien、Grady Booch、Bruce Anderson、Richard Gabriel、Doug Lea、Peter Coad、以及 Doug Schmidt。



你的旅途剛開始...

現在你已經站在設計模式的頂端，準備挖得更深了；我們為你準備了三本書，好添到你的書架上 ...

設計模式的經典書籍

這本書在 1995 年出版，揭開了設計模式的序幕。你可以在這本書中找到所有基礎的模式。事實上，這本書中所介紹模式，也正是本書的基礎。

這本書並非涵蓋了所有的模式－從這本書出版之後，這個領域就不斷地擴大－但儘管如此，這本書還是第一本也是最重要的一本書。

在你讀完 Head First 設計模式之後，這本書是個不錯的選擇。

這本書的作者後來被簡稱為「四人幫」，或簡稱 GoF。

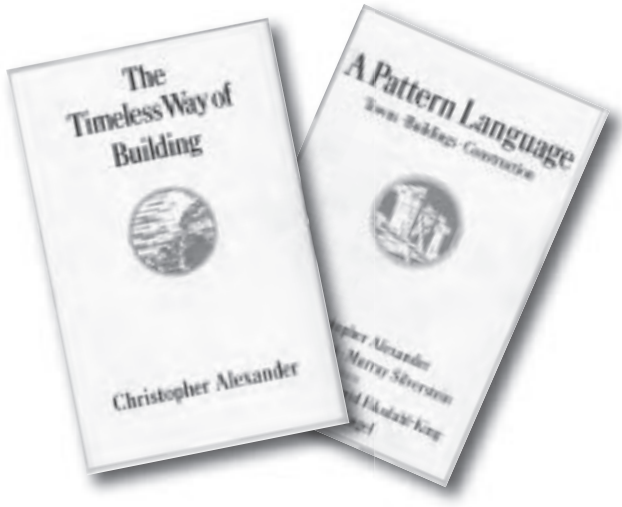
克里斯多夫亞歷山大發明了建築用的模式，引發軟體也跟進。

模式的經典書籍

模式並不是從四人幫開始的，而是由克里斯多夫、亞歷山大（Christopher Alexander）開始的。他是柏克萊的建築教授－沒錯，亞歷山大是個建築師，不是電腦科學家。亞歷山大發明了建築用的模式。

下次當你有心情挖掘得更深入時，可以閱讀這兩本書《The Timeless Way of Building》以及《A Pattern Language》，你就了解到設計模式的真正意義，並體認到建築和軟體之間的對比。

所以，拿起你的星巴茲咖啡，坐下靠在椅背上，開始享受這一切 ...



其他設計模式的資源

你會發現外面有許多的模式使用者族群，他們正敞開雙臂等候您的加入。這裏列出一些你一開始可以取得的資源 ...

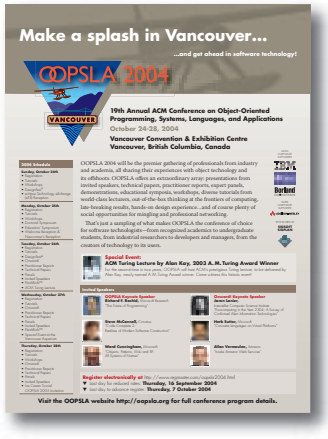
網站

The Portland Patterns Repository
(波特蘭模式儲存地)
由 Ward Cunningham 所經營，這是一個致力於模式相關資訊的 WIKI，任何人都可以參加。你可以在這裏看到許多人，對於模式和 OO 系統的相關問題，有豐富的討論。

http://c2.com/cgi/wiki?WelcomeVisitors

The Hillside Group
培養編程以及設計的事務經驗，並提供許多模式。這個網站包含了許多模式相關資源的資訊，例如文章、書籍、郵件論壇、以及工具。

http://hillside.net/



會議以及研討會

如果你想和模式的社群面對面的接觸，可以選擇參加會議和研討會。Hillside 網站對於這些活動有完整的清單。另外你至少也應該去看看 OOPSLA 的活動資訊。OOPSLA 是 ACM 的研討會，主題是針對 OO 系統、語言、應用。

模式動物園

就如我們剛剛所說的，模式並非從軟體開始；而是從建築的架構開始。事實上，模式的概念可以被採用在許多不同的領域。讓我們現在就來逛逛模式動物園，瞧瞧有那些模式 ...

架構模式

是建立生氣勃勃的建築結構。這也正是模式開始的地方。

產地：從你所居住、觀看、參觀的建築物中，可以發現它的蹤跡。

產地：出現在三層架構、兩層架構、以及 Web 系統中。

應用模式

是建立系統架構的模式。許多多層的架構都屬於這一類。

領域註記：MVC 可算是其中的一種。

特定領域的模式

解決特定領域的問題，例如 concurrent 系統、即時系統。

幫忙找到它的產地

J2EE

業務程序模式

描述業務、顧客、以及資料之間的互動，此種模式能夠處理某些問題，例如如何有效地做決策並溝通決策。

出沒在公司的會議室以及專案管理會議中。

幫助找到一個棲息地

開發團隊

顧客支援團隊

組織模式

描述了公司組織的結構以及業務項目。今日大多數的努力方向都是在於製造或支援軟體的組織。

使用者介面設計模式

致力於解決設計互動式軟體時的問題。

棲息地：被發現在遊戲設計程式、GUI 建構程式、製作程式中。

領域註記：請將你對於模式領域的觀察發現寫在這裏：

以反模式殲滅惡勢力

如果我們只有模式，而沒有反模式，那麼這個宇宙就不完整了。

如果設計模式能夠讓你在某個特定的情境之下，對某個一再出現的問題，提供一個有效的解決方案，那麼反模式能夠為你帶來什麼？

> **反模式**告訴你如何採用一個不好的解決方案解決一個問題。

你可能會這麼問「怎麼有人願意浪費時間將不好的解決方案寫成文件？」

這麼說好了：如果老是有人用某個不好的解決方案處理某個問題，而藉由將它寫成文件，可以幫助別人避免犯同樣的錯誤。畢竟，避免不好的解決方案，就和採用好的解決方案一樣有價值！

讓我們來看看一個反模式的組成單元：

反模式告訴我們為什麼不好的解決方案會有吸引力。
不可諱言的，如果差勁的解決方案沒有任何吸引力，那麼根本不會有人想要使用它。反模式最重要的工作之一，在於警告你不要陷入某種致命的吸引力。

反模式告訴你為何這個解決方案長遠以後會造成不好的影響。
為了瞭解為什麼這是一個反模式，你必須了解它在未來如何造成不好的影響。反模式會告訴你使用這個解決方案，在未來會為你帶來怎樣的麻煩。

反模式建議你改用其他的模式，提供更好的解決方案。
反模式除了告訴你什麼解決方案不好之外，也會告訴你該採用什麼其他的解決方案，這樣子反模式才真正有幫助。

讓我們現在就來看看一個反模式。

反模式看起來總像是一個好的解決方案，但是當它真正被採用後，就會帶來麻煩。

藉由將反模式寫成文件，我們能夠幫助其他人辨識出不好的解決方案，以免誤用。

就如同模式一般，有許多不同種類的反模式，包括了開發反模式、物件導向反模式、組織反模式、以及各種領域特定的反模式。

下面是一個軟體開發反模式的例子。

反模式

名稱：黃金榔頭

問題：你需要選擇技術作為軟體的開發之用，而且你相信正好有一個技術能夠主宰這個架構。

情境：你需要開發某個新的系統或者是一套軟體，然而此系統或軟體卻無法和現今的開發團隊所熟悉的技術相吻合。

力量：

- 開發團隊致力於採用他們所熟悉的技術。
- 開發團隊並不熟悉其他的開發技術。
- 採用不熟悉的技術被認為風險比較高。
- 使用熟悉的技術做開發，比較容易規劃和預估。

原本的解決方案：反正就使用熟悉的技術好了。將熟悉的技術採用在許多問題上，甚至連不適當的地方也照用。

重構的解決方案：開發者藉由教育、訓練、以及讀書會，可以學會新的解決方案。

範例：

當採用開放源碼的其他產品時，Web 公司依然持續使用，並維護他們內部自行開發的快取系統。

如同設計模式一般，每個反模式都有名字，讓我們方便稱呼。

問題與情境，如同設計模式所使用的描述。

告訴你為什麼這個解決方案是有吸引力的。

不好的、但是卻有吸引力的解決方案。

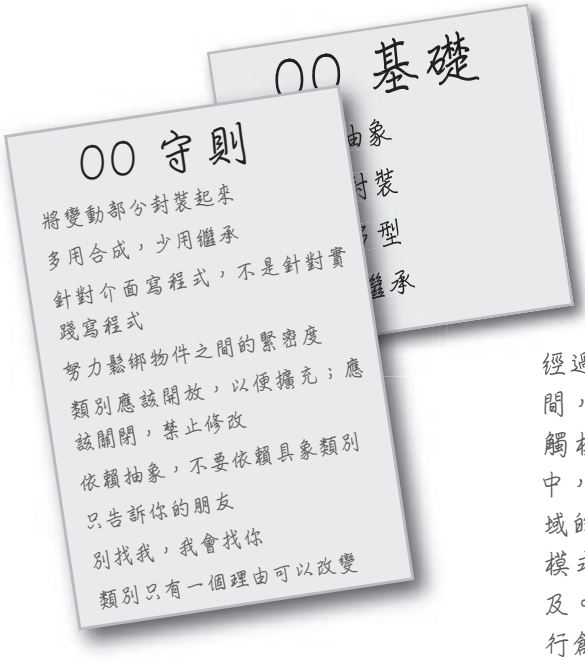
如何使用一個好的解決方案。

這個反模式會出現在什麼地方。

此例取自 Portland Pattern Repository的Wiki，從此處可以看到許多反模式，以及相關的討論。Wiki 的網址在 http://c2.com/

設計箱內的工具

你已經到了可以脫離我們的階段，現在該是你走向外面的世界，憑著自己的能力摸索模式。



經過了這麼長的一段時間，你現在可以獨立接觸模式。外面的世界中，有許多屬於特定領域的模式和一些基礎的模式，並未在本書中提及。另外，也也可以自行創建新的模式。

去看看第十四章，有介紹一些基礎的模式，你可能會感興趣。

要點

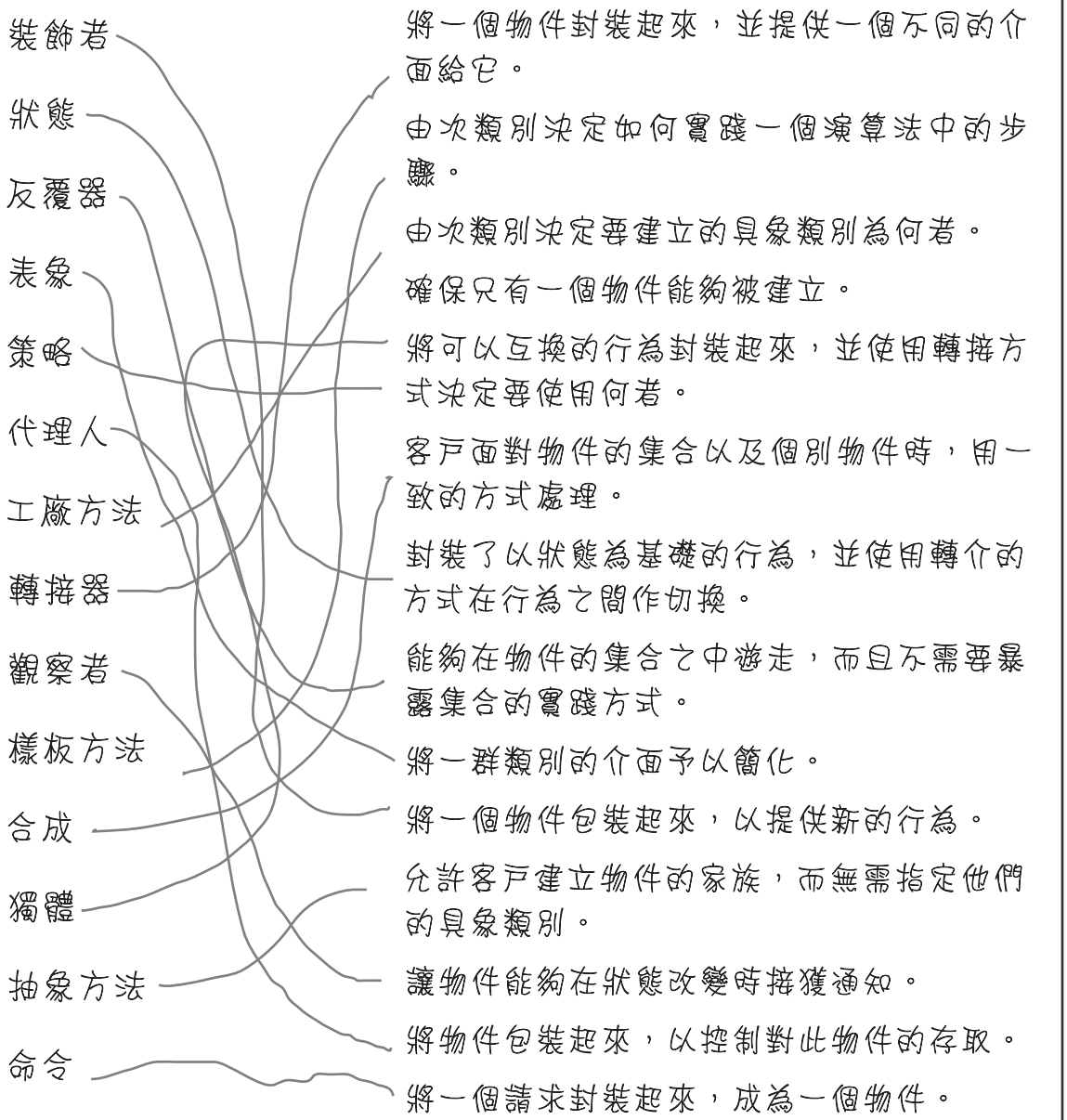
- 讓設計模式自然而然地出現在你的設計中，而不是為了使用而使用。
- 設計模式並非僵化的教條；你可以依據自己的需求採用或調整。
- 有沒有使用模式，並不是重點；使用能夠符合需求的最簡單解決方案，才是上策。
- 學習設計模式的類目，可以幫你自己熟悉這些模式，以及它們之間的關係。
- 模式的分類是將模式分成不同的族群，如果這麼做對你有幫助，就採用吧！
- 你必須相當專注才能夠成為一個模式的作家：這需要時間也需要經驗，同時還必須有很高的意願，能夠進行相當多的修改過程。
- 請牢記：你所遇到大多數的模式都是既有的模式，而非新的模式。
- 模式能夠為你帶來的好處之一是：讓你的團隊具有共同語彙。
- 任何社群都有自己的術語，模式社群也是如此。別讓這些術語絆著，在讀完這本書之後，你已經能夠運用大部份的術語了。

離開物件村...

有你們的日子真好。

我們一定會想念你們的。但是，別擔心，下一本 Head First 書籍很快就會出版，到時候歡迎你們再度來訪。你問我下一本書是什麼主題，這...真是好問題！你要不要給點意見？把你的想法透過 email 告訴我們吧！booksuggestions@wickedlysmart.com

習題解答

連連看

每個模式的敘述為何者