



软件建模与分析

Software Modeling and Analyzing

主讲人：徐峰磊



设计模式

- ❑ 设计模式：Design Pattern
- ❑ 简单地讲，所谓模式，就是得到很好研究范例。设计模式就是设计范例。
- ❑ 在孙子兵法中充斥着各种模式，孙子说，“置于死地而后生”就是战争模式。36计条条都是模式，比如“走为上”，“空城计”也都是战争模式。
- ❑ 这些模式中每一个都具有典型意义，具有学习价值。通过研究这些模式，学习者可以相互交流，可以在自己实战中举一反三，推陈出新，加以应用。

面向对象软件设计过程

- ❑ 找出与当前问题相关的对象;
- ❑ 以适当的颗粒度将它们归类;
- ❑ 定义类的接口和继承层次;
- ❑ 建立对象间的基本关系。



设计目标

- 设计应该对手头的问题有针对性;
- 同时对将来的问题和需求也要有足够的通用性;
- 避免重复设计或尽可能少做重复设计。



设计的困难

□ 有经验的面向对象设计者会告诉你：

- 要一下子就得到复用性和灵活性好的设计，即使不是不可能的，至少也是非常困难的。
- 一个设计在最终完成之前常要被复用好几次，而且每一次都有所修改。



新手的弱势

- 有经验的面向对象设计者的确能做出良好的设计，而新手则面对众多选择无从下手，总是求助于以前使用过的非面向对象技术。新手需要花费较长时间领会良好的面向对象设计是怎么回事。



内行设计者

- ❑ 不是解决任何问题都要从头做起，他们更愿意复用以前使用过的解决方案。当找到一个好的解决方案，他们会一遍又一遍地使用。这些经验成为内行的部分原因。
- ❑ 因此，你会在许多面向对象系统中看到类和相互通信的对象的重复模式。这些模式解决特定的设计问题，使面向对象设计更灵活、优雅，最终复用性更好。它们帮助设计者将新的设计建立在以往工作的基础上，复用以往成功的设计方案。
- ❑ 一个熟悉这些模式的设计者不需要再去发现它们，而能够立即将它们应用于设计问题中。



类比

- ❑ 小说家和剧本作家很少从头开始设计剧情，他们总是沿用一些业已存在的模式，像“悲剧性英雄”模式（《麦克白》、《哈姆雷特》等）或“浪漫小说”模式（存在着无数浪漫小说）。
- ❑ 同样地，面向对象设计员也沿袭一些模式，像“用对象表示状态”和“修饰对象以便于你能容易地添加/删除属性”等。
- ❑ 一旦懂得了模式，许多设计决策自然而然就产生了。



设计模式

- ❑ 将面向对象的设计经验作为设计模式记录下来。
- ❑ 每一个设计模式系统地命名、解释和评价了面向对象系统中一个重要的和重复出现的设计。
- ❑ 目标是将设计经验以人们能够有效利用的形式记录下来。
- ❑ 鉴于此目的，编写了一些最重要的设计模式，并以编目分类的形式将它们展现出来。



设计模式的作用

- ❑ 设计模式使人们可以更加简单方便地复用成功的设计和体系结构。将以证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。设计模式帮助你做出有利于系统复用的选择，避免设计损害了系统复用性。
- ❑ 通过提供一个显式类和对象作用关系以及它们之间潜在联系的说明规范，设计模式甚至能够提高已有系统的文档管理和系统维护的有效性。简而言之，设计模式可以帮助设计者更快更好地完成系统设计。



什么是设计模式？

- ❑ Christopher Alexander说过：“每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复劳动”。
- ❑ 尽管Alexander所指的是城市和建筑模式，但他的思想也同样适用于面向对象设计模式，只是在面向对象的解决方案里，我们用对象和接口代替了墙壁和门窗。两类模式的核心都在于提供了相关问题的解决方案。



模式的四个基本要素：

□ 模式名称(pattern name)

- 一个助记名，它用一两个词来描述模式的问题、解决方案和效果。

□ 问题(problem)

- 描述了应该在何时使用模式。它解释设计问题和问题存在的前因后果，它可能描述了特定的设计问题，如怎样用对象表示算法等。也可能描述了导致不灵活设计的类或对象结构。有时候，问题部分会包括使用模式必须满足的一系列先决条件。



模式的四个基本要素（续）：

□ 解决方案(solution)

- 描述了设计的组成成分，它们之间的相互关系及各自的职责和协作方式。因为模式就像一个模板，可应用于多种不同场合，所以解决方案并不描述一个特定而具体的设计或实现，而是提供设计问题的抽象描述和怎样用一个具有一般意义的元素组合(类或对象组合)来解决这个问题。



模式的四个基本要素（续）：

□ 效果(consequences)

- 描述了模式应用的效果及使用模式应权衡的问题。尽管我们描述设计决策时，并不总提到模式效果，但它们对于评价设计选择和理解使用模式的代价及好处具有重要意义。
- 软件效果大多关注对时间和空间的衡量，它们也表述了语言和实现问题。因为复用是面向对象设计的要素之一，所以模式效果包括它对系统的灵活性、扩充性或可移植性的影响，显式地列出这些效果对理解和评价这些模式很有帮助。

模式的四个基本要素（续）：

- 一个设计模式命名、抽象和确定了一个通用设计结构的主要方面，这些设计结构能被用来构造可复用的面向对象设计。设计模式确定了所包含的类和实例，它们的角色、协作方式以及职责分配。
- 每一个设计模式都集中于一个特定的面向对象设计问题或设计要点，描述了什么时候使用它，在另一些设计约束条件下是否还能使用，以及使用的效果和如何取舍。
- 既然我们最终要实现设计，设计模式还提供了**C++**和**Smalltalk**实例代码来阐明其实现。



Smalltalk MVC中的设计模式

- ❑ **MVC**是三个单词的缩写，这三个单词分别为：模型（**Model**）、视图(**View**)和控制器(**Controller**)。
- ❑ 把一个应用的输入、处理、输出流程按照**Model、View、Controller**的方式进行分离，这样一个应用被分成三个类对象——模型、视图和控制器。

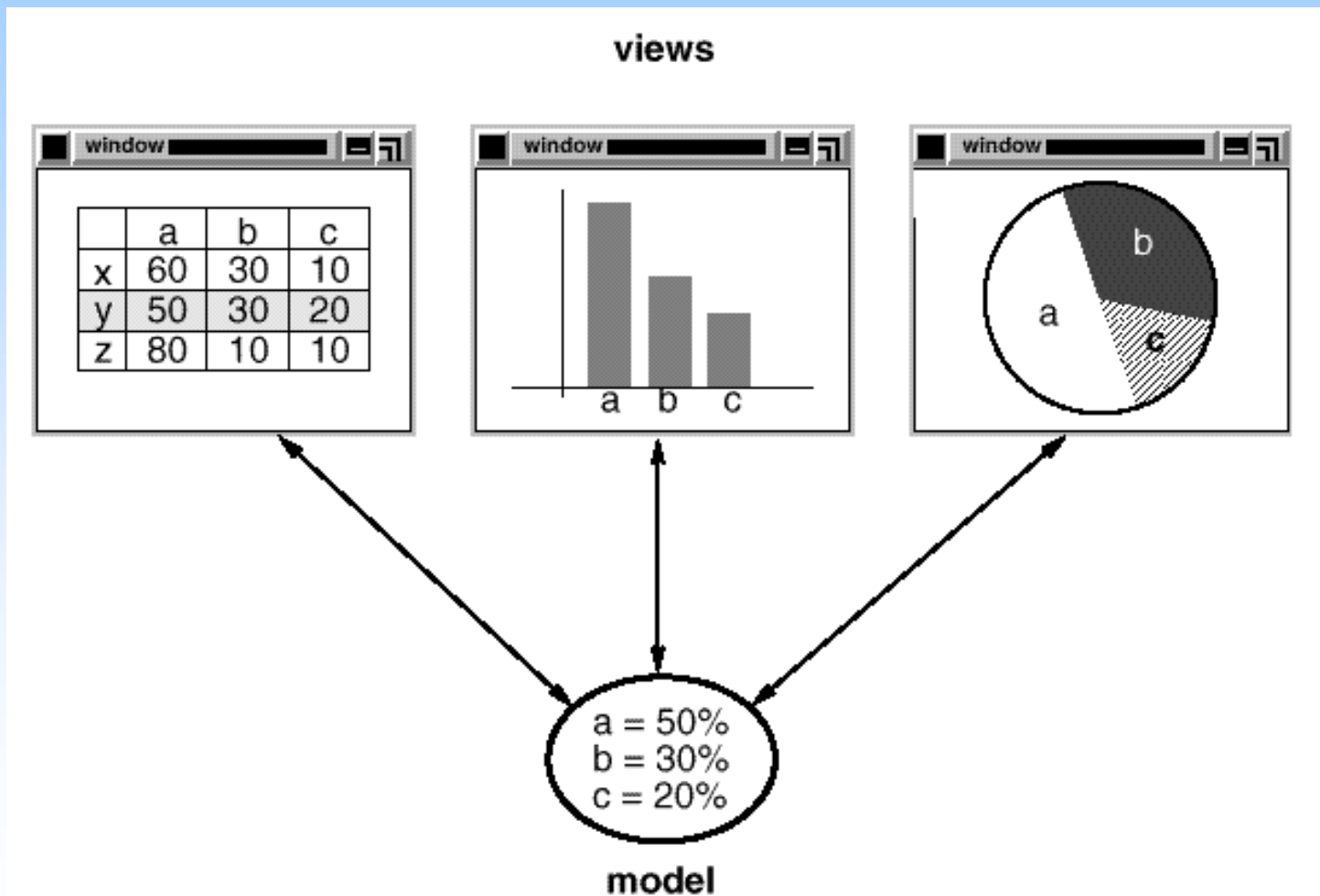


MVC中的设计模式

- ❑ MVC通过建立一个“订阅/通知”协议来分离视图和模型。
- ❑ 视图必须保证它的显示正确地反映了模型的状态。一旦模型的数据发生变化，模型将通知有关的视图，每个视图相应地得到刷新自己的机会。
- ❑ 为一个模型提供不同的多个视图表现形式，也能够为一个模型创建新的视图而无须重写模型。



□ 一个模型和三个视图



Observer模式

□表面上看，这个例子反映了将视图和模型分离的设计，然而这个设计还可用于解决更一般的问题：将对象分离，使得一个对象的改变能够影响另一些对象，而这个对象并不知道那些被影响的对象的细节。这个更一般的设计被描述成**Observer**模式（观察者模式）。



Composite模式（组合模式）

- ❑ **MVC**的另一个特征是视图可以嵌套。采用**View**类的子类**Composite View**类来支持嵌套视图。
- ❑ 该设计也适用于更一般的问题：将一些对象划为一组，并将该组对象当作一个对象来使用。这个设计被描述为**Composite**模式，该模式允许你创建一个类层次结构，一些子类定义了原子对象，而其他类定义了组合对象，这些组合对象是有原子对象组合而成的更复杂的对象。



Strategy模式（策略模式）

- ❑ **MVC**允许在不改变视图外观的情况下改变视图对用户输入的响应方式。**MVC**将响应机制封装在**Controller**对象中，存在一个**Controller**的类层次结构，使得可以方便地对原有**Controller**做适当改变而创建新的**Controller**。
- ❑ **View**使用**Controller**子类的实例来实现一个特定的响应策略。要实现不同的响应策略只要用不同种类的**Controller**实例替换即可。
- ❑ **View-Controller**关系是**Strategy**模式的一个例子。



描述设计模式

□ 模式名和分类

- 模式名简洁地描述了模式的本质。

□ 意图

- 设计模式是做什么的？
- 它的基本原理和意图是什么？
- 它解决的是什么样的特定设计问题？

□ 别名

- 模式的其他名称



□ 动机

- 用于说明一个设计问题以及如何用模式中的类、对象来解决该问题的特定情景。该情景会帮助你理解随后对模式更抽象的描述。

□ 适用性

- 什么情况下可以使用该设计模式？该模式可用来改进哪些不良设计？你怎样识别这些情况？

□ 结构

- 采用基于对象建模技术(**OMT**)的表示法对模式中的类进行图形描述。
- 使用交互图来说明对象之间的请求序列和协作关系。



□ 参与者

- 指设计模式中的类和/或对象以及它们各自的职责。

□ 协作

- 模式的参与者怎样协作以实现它们的职责。

□ 效果

- 模式怎么支持它的目标？使用模式的效果和所需做的权衡取舍？系统结构的哪些方面可以独立改变？

□ 实现

- 实现模式是需要直达的一些提示、技术要点及应避免的缺陷，以及是否存在某些特定于实现语言的问题。

□ 代码示例

- 用来说明怎样用C++实现该模式的代码片段。

□ 已知应用

□ 相关模式

- 与这个模式紧密相关的模式有哪些？其间重要的不同之处是什么？这个模式应与其他模式一起使用？



设计模式的编目

- ❑ **Abstract Factory (3.1)** : 提供一个创建一系列相关或相依赖对象的接口，而无需指定它们具体的类。
- ❑ **Adapter (4.1)** : 将一个类的接口转换成客户希望的另外一个接口。**Adapter**模式使得原本由于接口不兼容而不能工作的那些类可以在一起工作。
- ❑ **Bridge (4.2)** : 将抽象部分与它的实现部分分离，使它们都可以独立地变化。



- ❑ **Builder (3.2)**：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。
- ❑ **Chain of Responsibility (5.1)**：为解除请求的发送者和接受者之间的耦合，而使多个对象都有机会处理这个请求。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它。
- ❑ **Command (5.2)**：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持取消的操作。



- ❑ **Composite (4.3)**：将对象组合成树形结构以表示“部分-整体”的层次结构。**Composite**使得客户对单一对象和复合对象的使用具有一致性。
- ❑ **Decorator (4.4)**：动态地给一个对象添加一些额外的职责。就扩展功能而言，**Decorator**模式比生成子类方式更为灵活。
- ❑ **Façade (4.5)**：为子系统中的一组接口提供一个一致的界面，**Façade**模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。
- ❑ **Factory Method (3.3)**：定义一个用于创建对象的接口，让子类决定哪一个类实例化。



- ❑ **Flyweight (4.6)** : 运用共享技术有效地支持大量细粒度的对象。
- ❑ **Interprete (5.3)** : 给定一种语言, 定义它的文法的一种表示, 并定义一个解释器, 该解释器使用该表示来解释语言中的句子。
- ❑ **Iterator (5.4)** : 提供一种方法顺序访问一个聚合对象中各个元素, 而又不需要暴露该对象内部表示。
- ❑ **Mediator (5.5)** : 用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用, 从而使其耦合松散, 而且可以独立地改变它们之间的交互。
- ❑ **Memento (5.6)** : 在不破坏封装性的前提下, 捕获一个对象的内部状态, 并在该对象之外保存这个状态。这样以后就可将该对象恢复到保存的状态。



- ❑ **Observer (5.7)** : 定义对象间的一种一对多的依赖关系, 以便当一个对象的状态发生改变时, 所有依赖它的对象都得到通知, 并自动刷新。
- ❑ **Prototype (3.4)** : 用原型实例指定创建对象的种类, 并且通过拷贝这个原型来创建新的对象。
- ❑ **Proxy (4.7)** : 为其他对象提供一个代理以控制对这个对象的访问。
- ❑ **Singleton (3.5)** : 保证一个对象仅有一个实例, 并提供一个访问它的全局访问点。



- ❑ **State (5.8)**：允许一个对象在内部状态改变时改变它的行为。对象看起来似乎修改了它所属的类。
- ❑ **Strategy (5.9)**：定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法的变化可独立于使用它的客户。
- ❑ **Template Method (5.10)**：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。**Template Method**使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤，
- ❑ **Visitor (5.11)**：表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。



参考书



作者: 弗里曼

出版社: 中国电力出版社

译者: O'Reilly Taiwan公司

出版年: 2007-9

页数: 636

定价: 98.00元

ISBN: 9787508353937



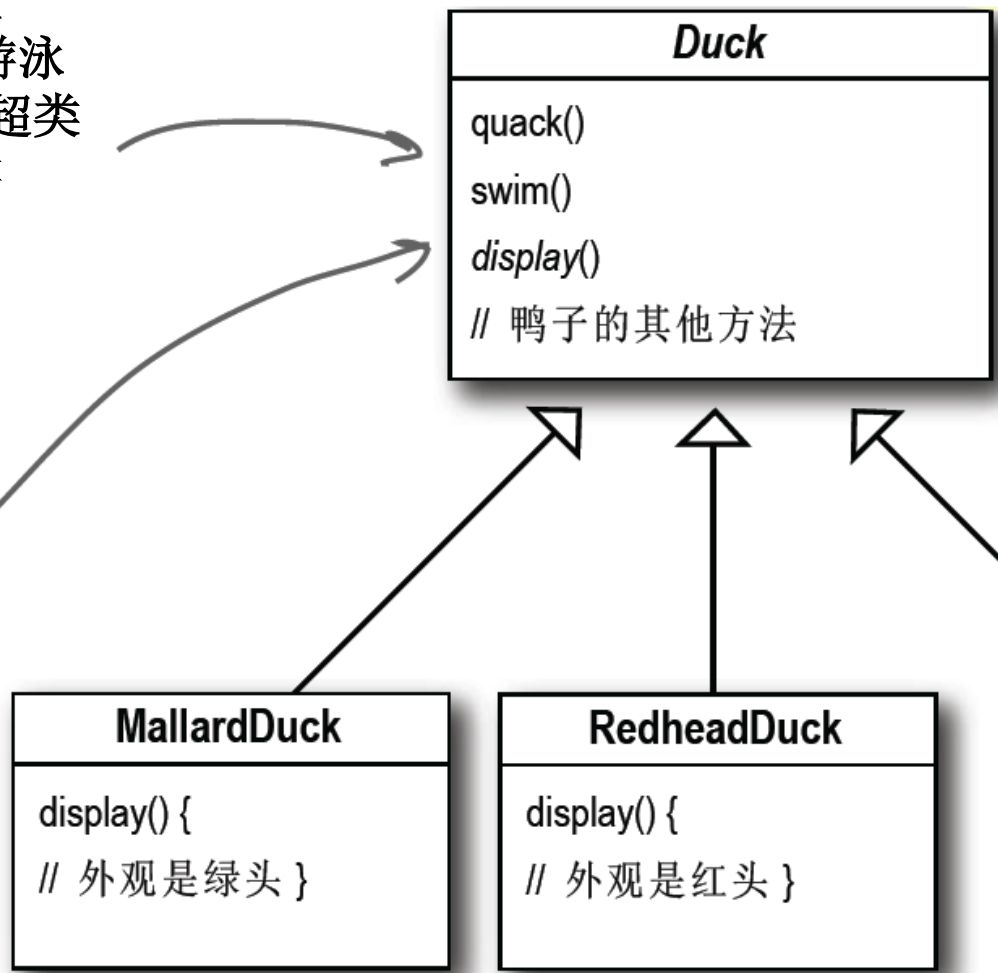
先从简单的模拟鸭子应用做起

□ **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。



所有的鸭子都会呱呱叫（**Quack**）也会游泳（**Swim**），所以由超类负责处理这部分的实现代码。

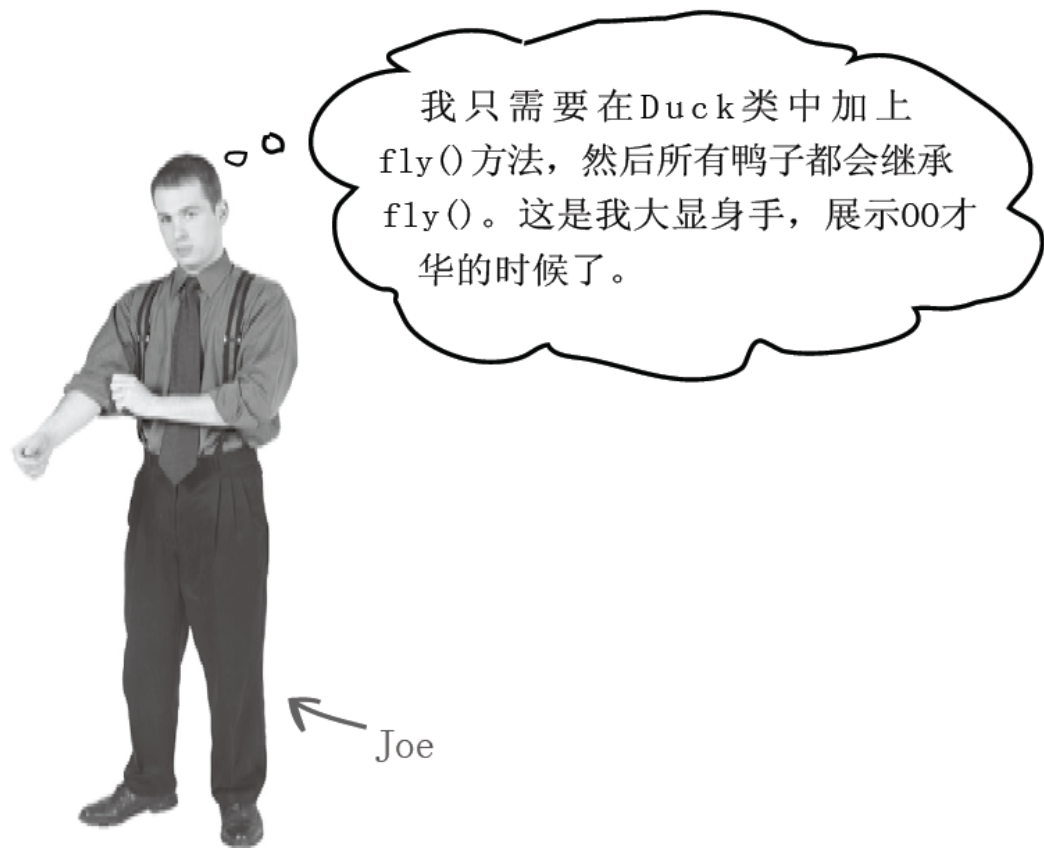


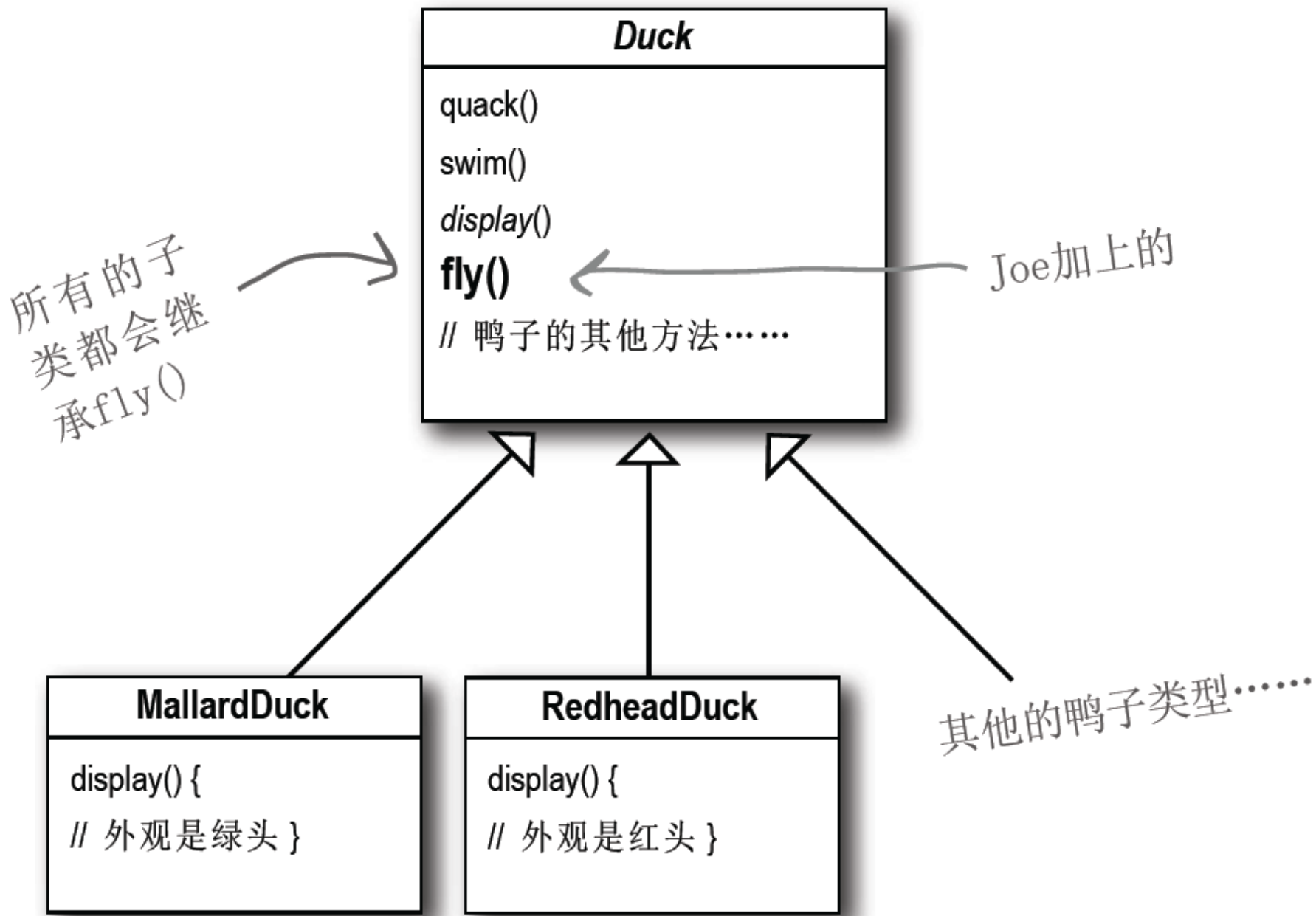
因为每一种鸭子的外观都不同，所以**display()**方法是抽象的。

每个鸭子子类型（**subtype**）负责实现自己的**display()**行为在屏幕上显示其外观。

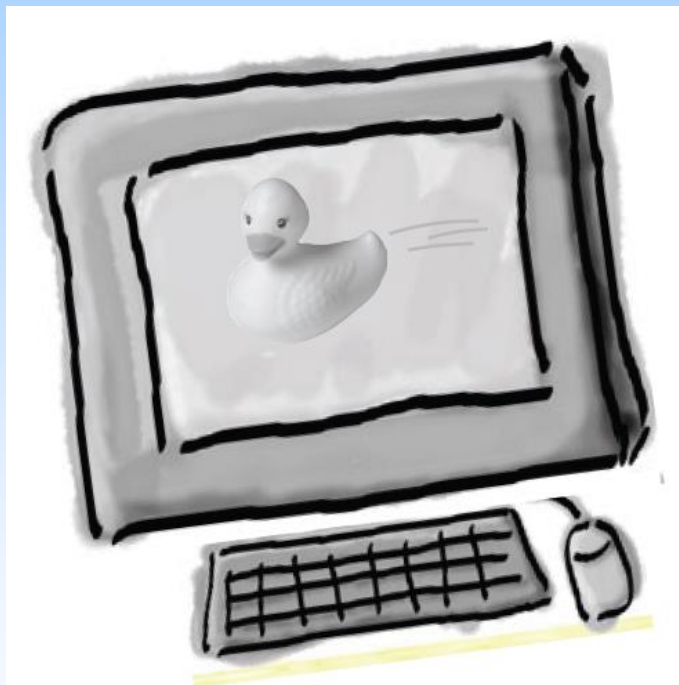
许多其他类型的鸭子继承**Duck**类。

现在我们得让鸭子能飞





但是，可怕的问题发生了.....

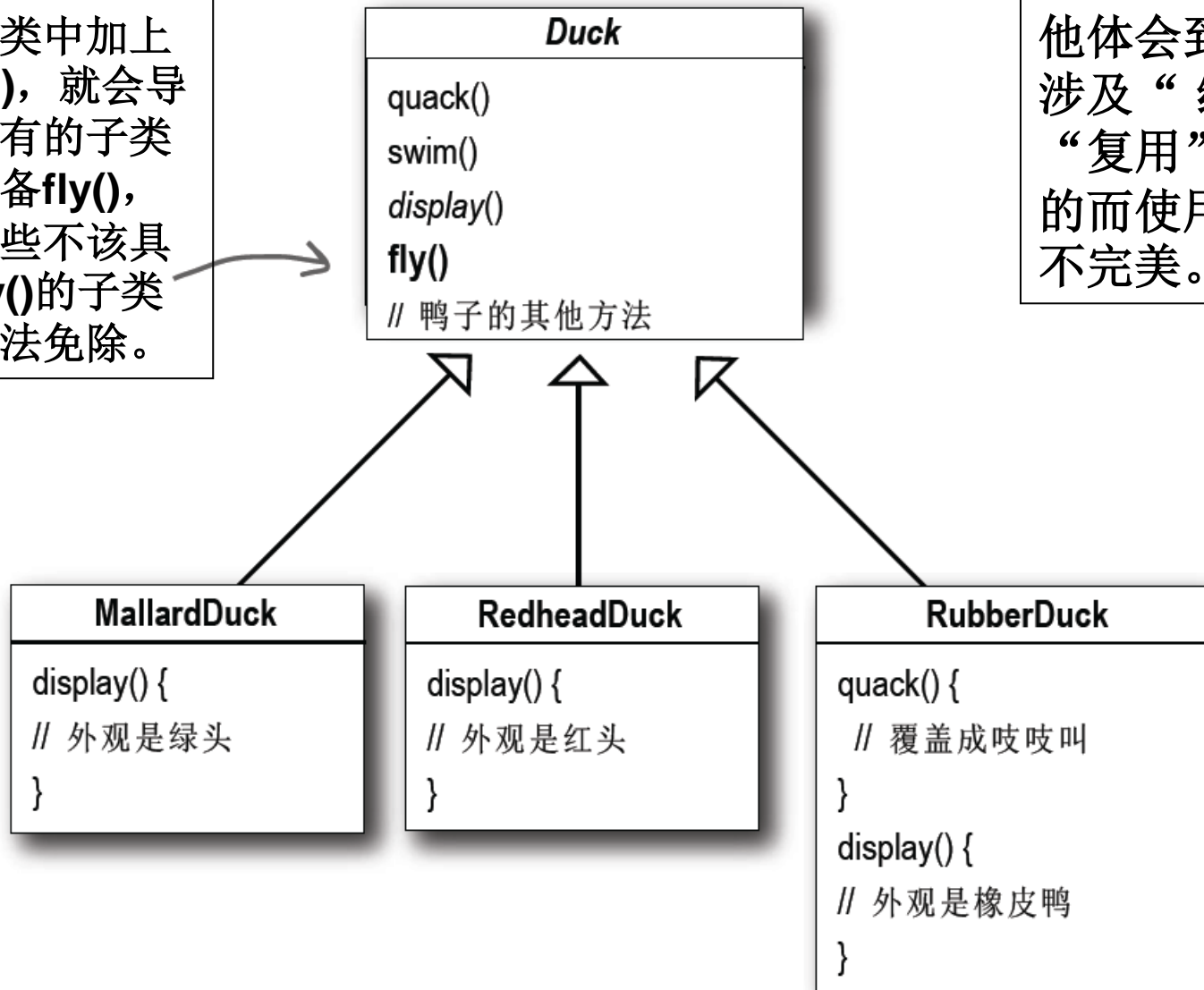


Joe，我正在股东会议上，刚刚看了一下展示，有很多“橡皮鸭子”在屏幕上飞来飞去，这是你在开玩笑吗？

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

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。

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。



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

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



RubberDuck

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

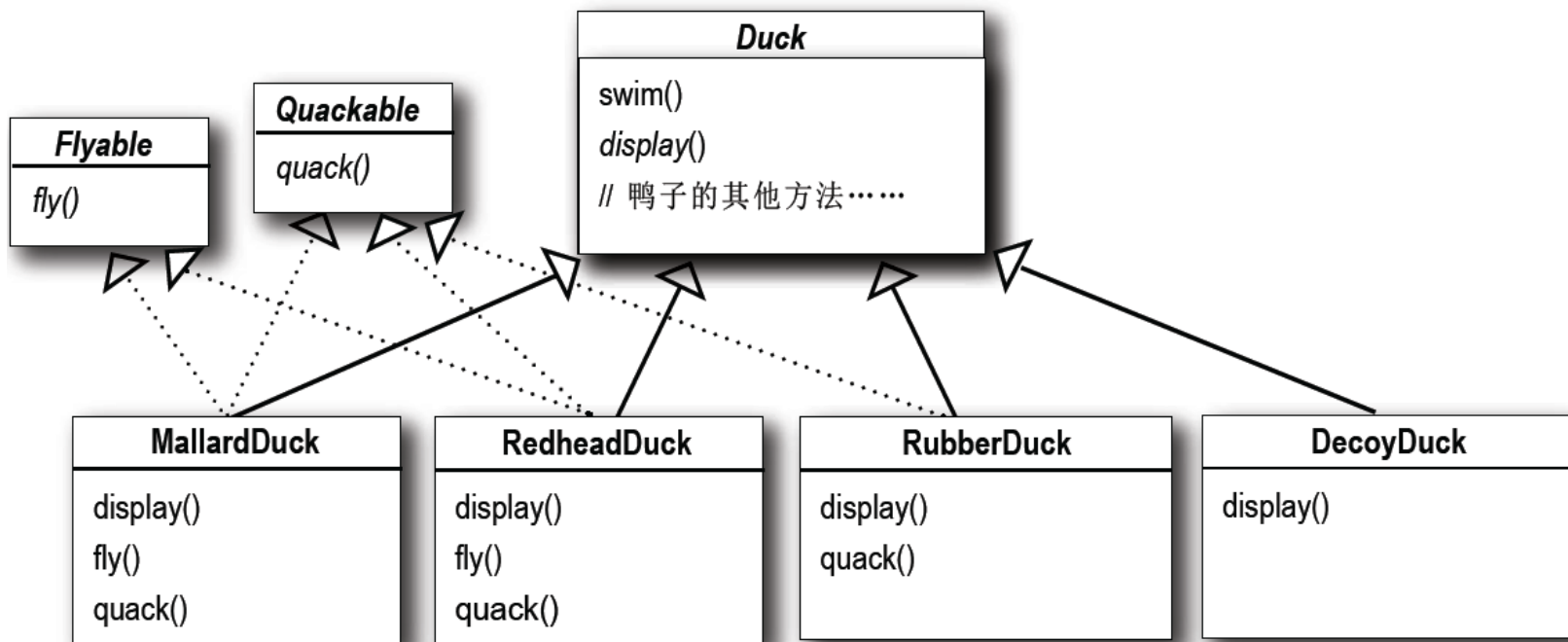
可是，如果以后我加入诱饵鸭（DecoyDuck），又会如何？诱饵鸭是木头假鸭，不会飞也不会叫……



DecoyDuck

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

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



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

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？！

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



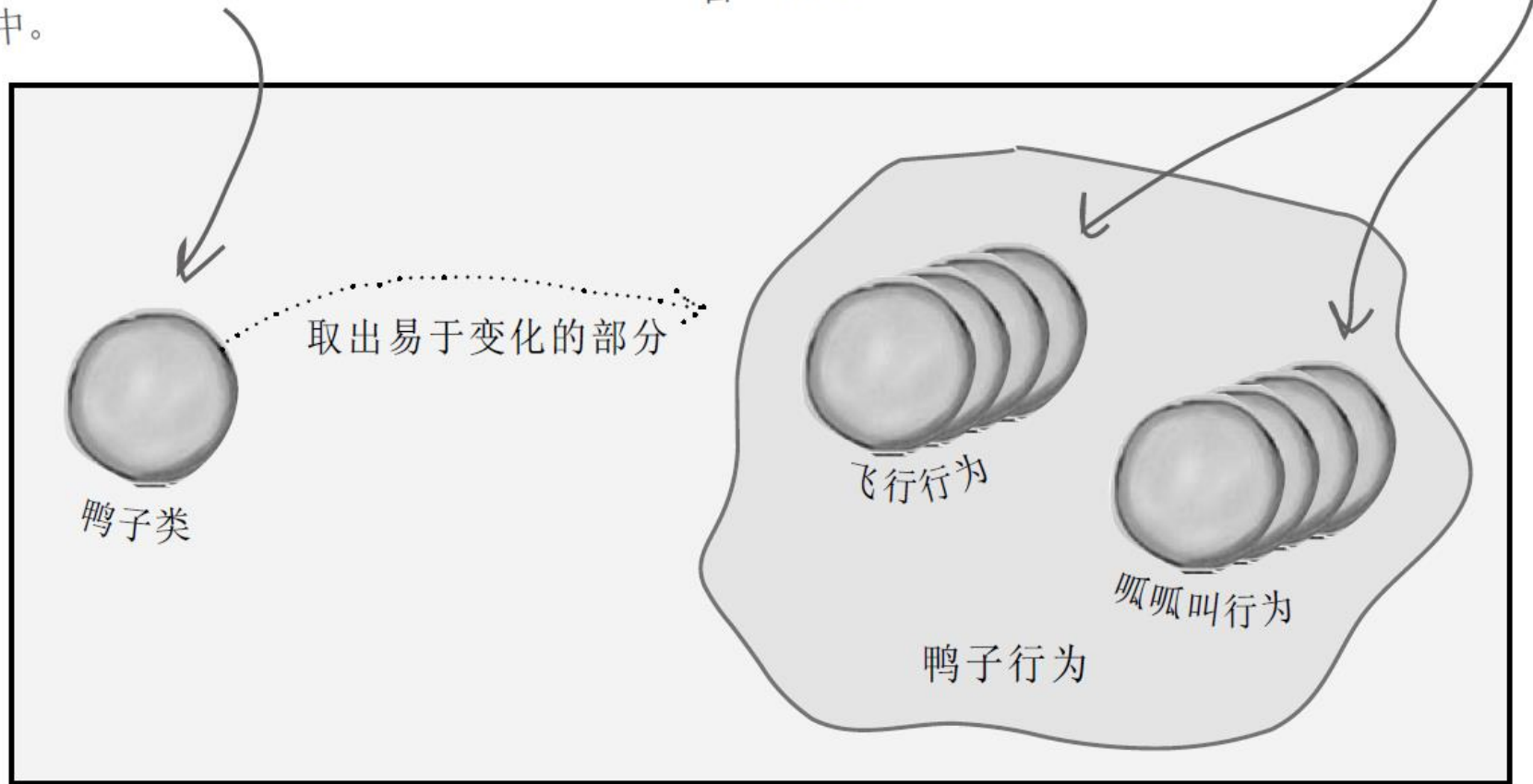
找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代码混在一起。

- ❑ 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。

Duck类仍是所有鸭子的超类，
但是飞行和呱呱叫的行为已经
被取出，放在别的类结构中。

现在飞行和呱呱叫都有它们
自己的类了。

多种行为的实现被放
在这里。

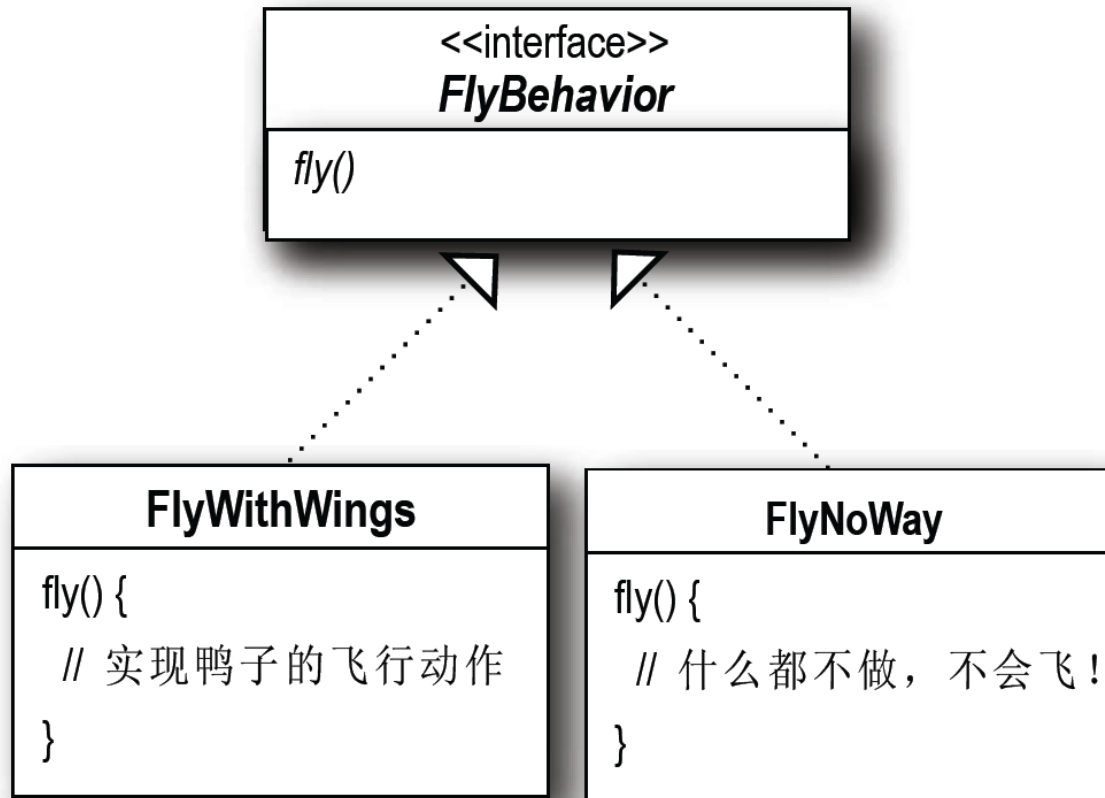




设计原则

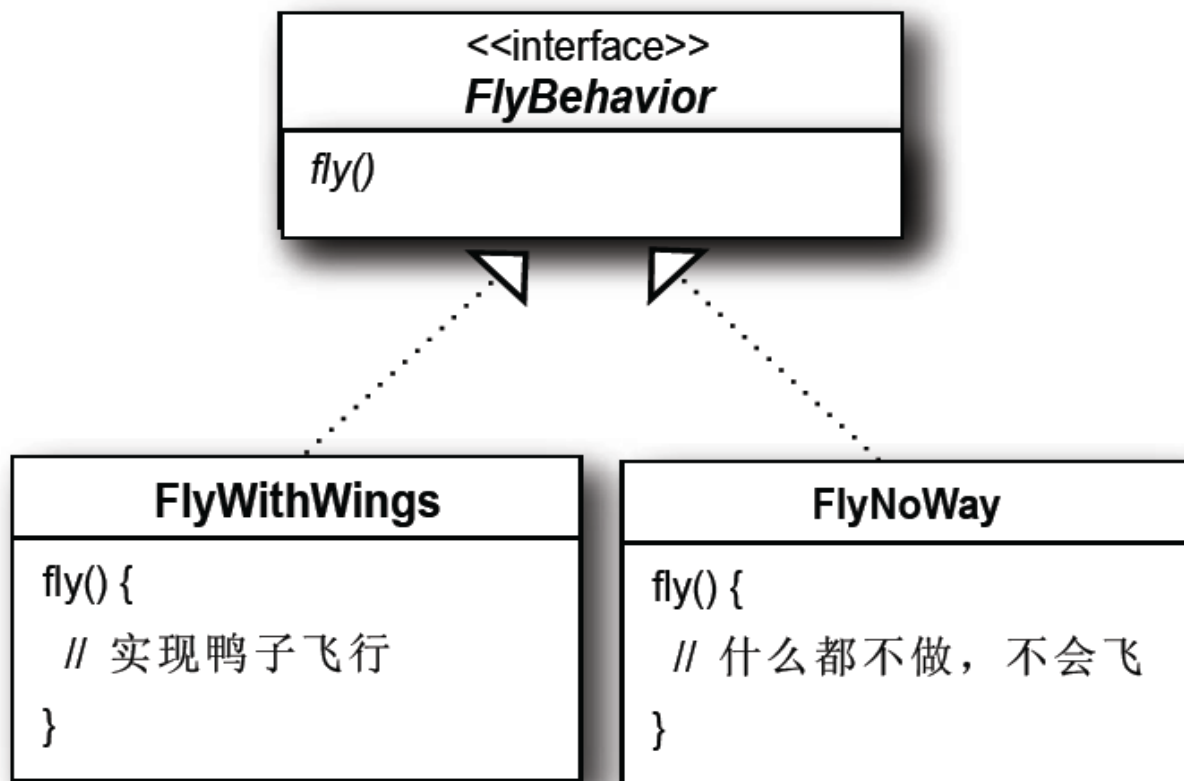
针对接口编程，而不是针对实现编程。

- ❑ 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。

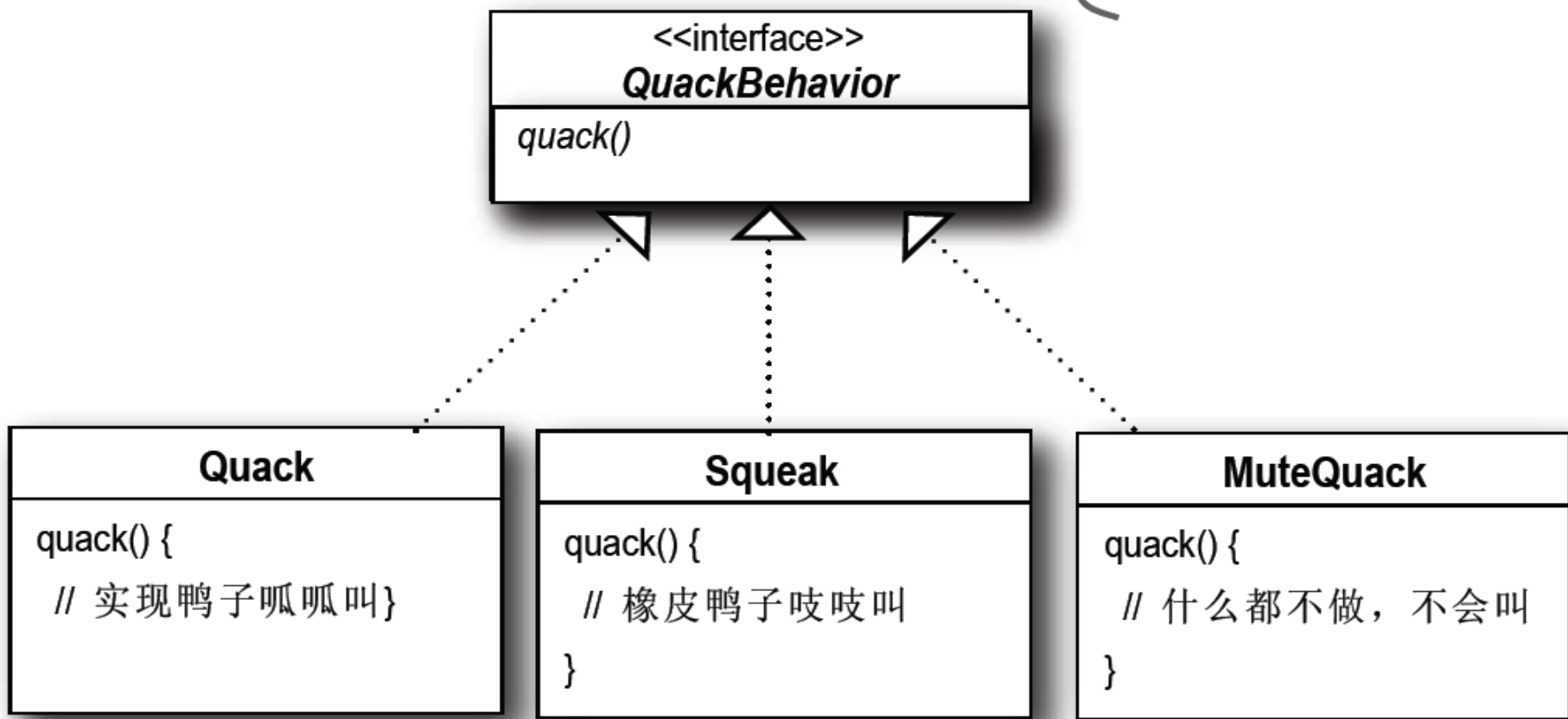


在新设计中，鸭子的子类将使用接口（**FlyBehavior**与**QuackBehavior**）所表示的行为，所以实际的“实现”不会被绑死在鸭子的子类中。（换句话说，特定的具体行为编写在实现了**FlyBehavior**与**QuackBehavior**的类中）。

这是一个接口，所有飞行类都实现它，所有新的飞行类都必须实现fly()方法。



呱呱叫行为也一样，一个接口只包含一个需要实现的quack()方法。

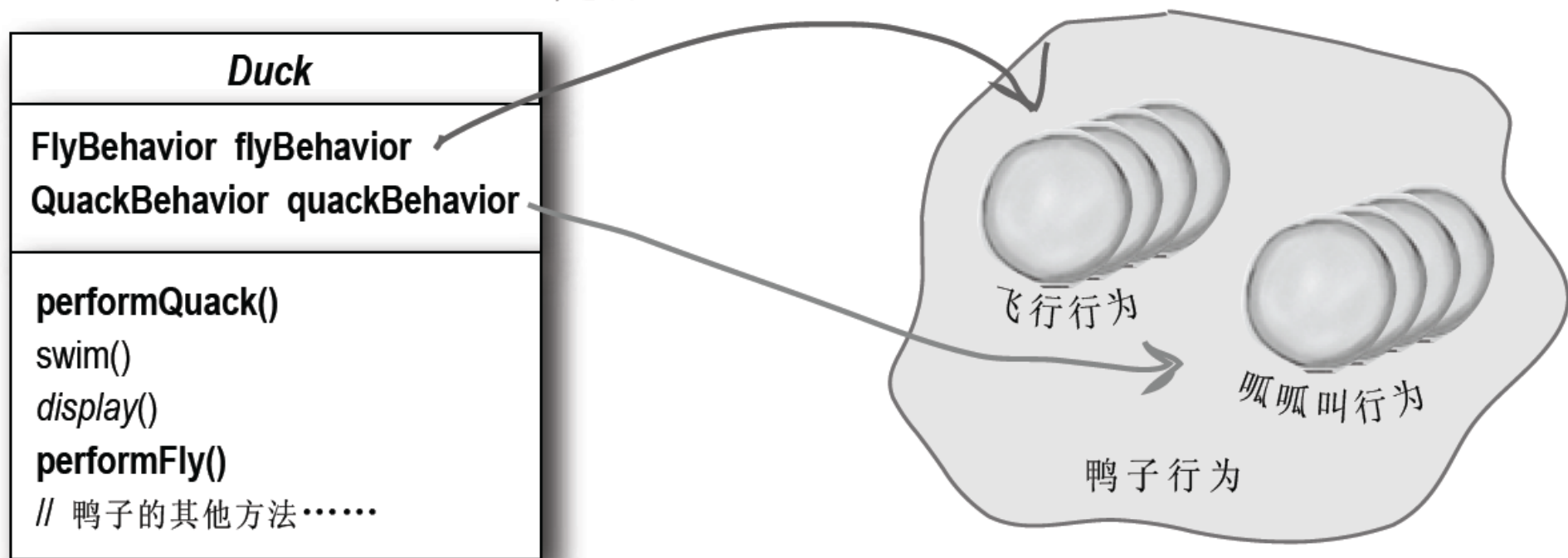


整合鸭子的行为

- ❑ 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。
- ❑ 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。



实例变量在运行时持有特定行为的引用。



performQuack()替代了原来的**quack()**
performFly()替代了原来的**fly()**

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。



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");  
    }  
}
```

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

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

使用FlyWithWings作为其FlyBehavior类型。



1

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

```
public abstract class Duck {  
  
    FlyBehavior flyBehavior;  
    QuackBehavior quackBehavior;  
    public Duck() {  
    }  
  
    public abstract void display();  
  
    public void performFly() {  
        flyBehavior.fly();  
    }  
  
    public void performQuack() {  
        quackBehavior.quack();  
    }  
  
    public void swim() {  
        System.out.println("All ducks float, even decoys!");  
    }  
}
```

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

委托给行为类



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

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

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）。



3 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");  
    }  
}
```

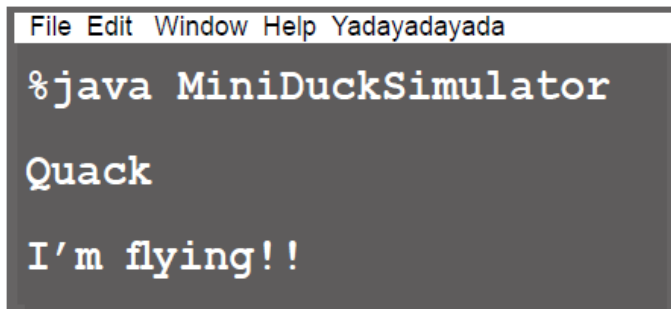


4 输入并编译测试类 (MiniDuckSimulator.java)

```
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()）。

5 运行代码！



A terminal window with a menu bar (File, Edit, Window, Help, Yadayadayada) and a dark background. The command `%java MiniDuckSimulator` has been entered. The output shows `Quack` on the first line and `I'm flying!!` on the second line.

至于performFly(), 也是一样的道理。



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;  
}
```

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

从此以后，我们可以“随时”调用这两个方法改变鸭子的行为。



2 制造一个新的鸭子类型：模型鸭 (ModelDuck.java)

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

一开始，我们的模型鸭是不会飞的。

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

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

没关系，我们建立一个利用火箭动力的飞行行为。



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

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

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

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

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！

```
mallard.performFly();
```

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

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！

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。

5 运行！

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

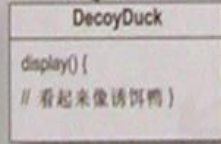
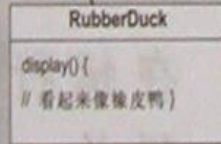
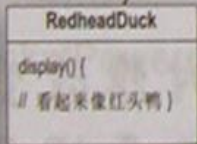
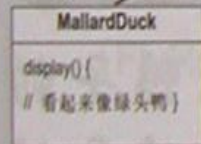
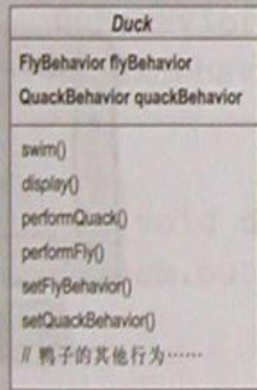


改变后

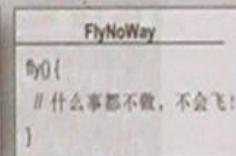
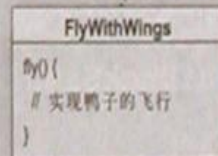
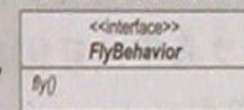


客户使用封装好的飞行和呱呱叫
算法族。

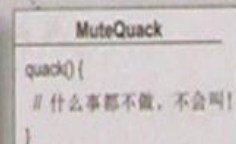
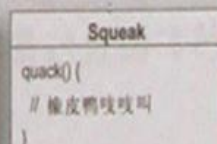
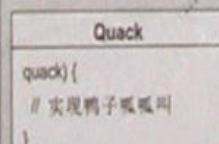
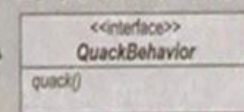
客户



封装飞行行为



封装呱呱叫行为



把每组行为想象
成一个算法族。

这些“~~行为~~”
“算法”是可以互换





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。



第一个设计模式

□ 策略模式(Strategy Pattern)

定义了算法族，分别封装起来，让它们之间可以相互替换，此模式让算法的变化独立于使用算法的客户。



在下面，你将看到一堆杂乱的类与接口，这取自一个动作冒险游戏。你将看到代表游戏角色的类和角色可以使用的武器行为的类。每个角色一次只能使用一种武器，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……

(答案在本章结尾处)

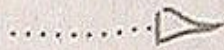
你的任务：

- ❶ 安排类。
- ❷ 找出一个抽象类、一个接口，以及八个类。
- ❸ 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”）。

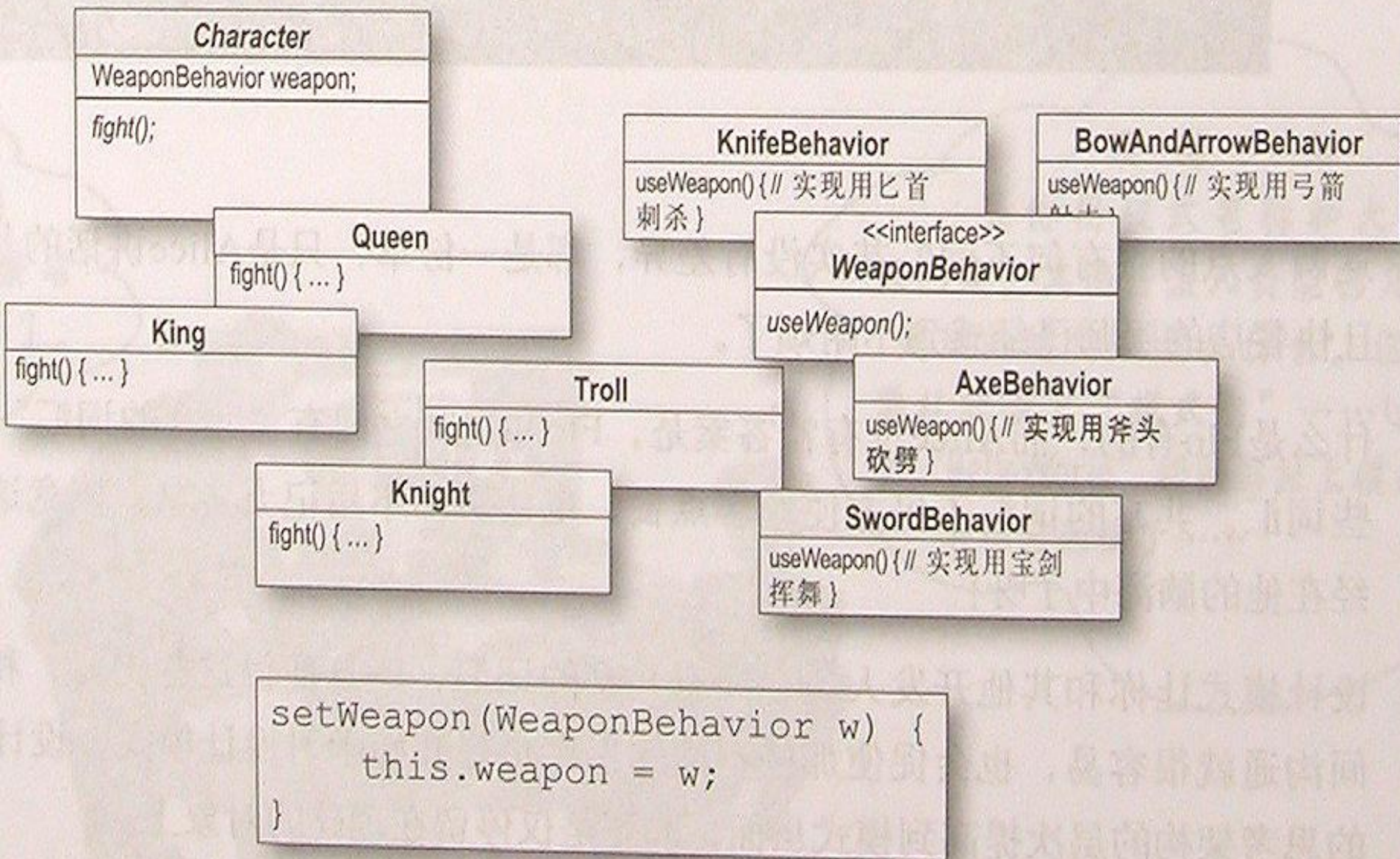


c. “有一个”关系就画成这样。



- ❹ 把setWeapon()方法放到正确的类中。



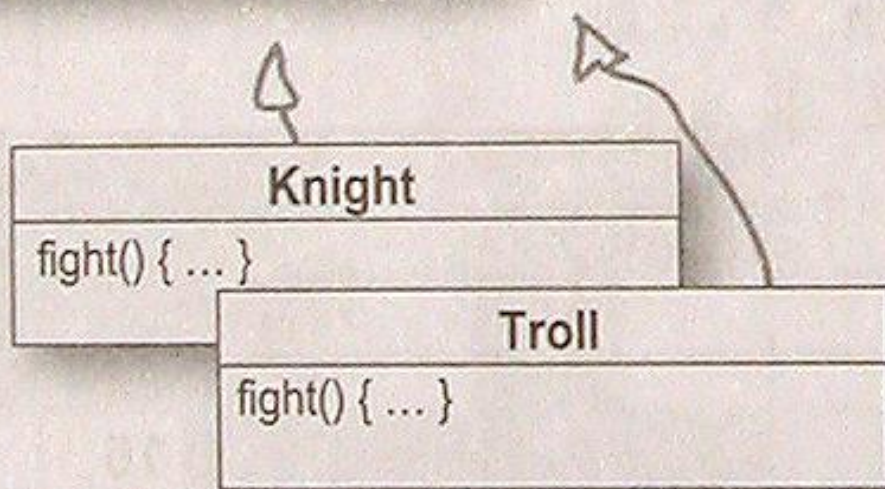
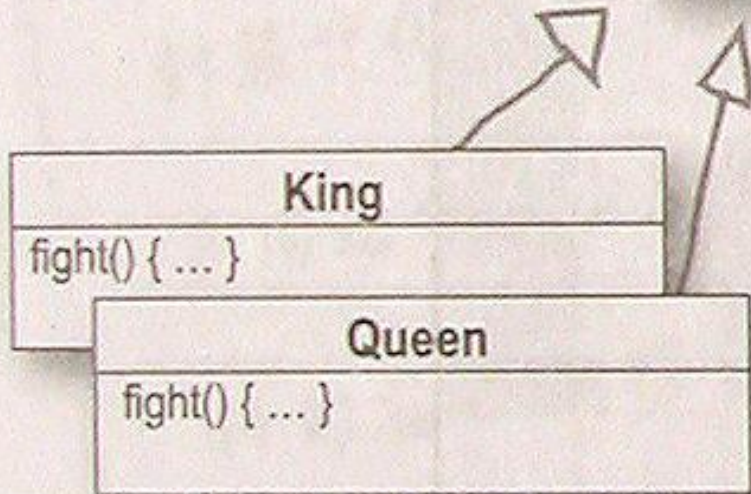
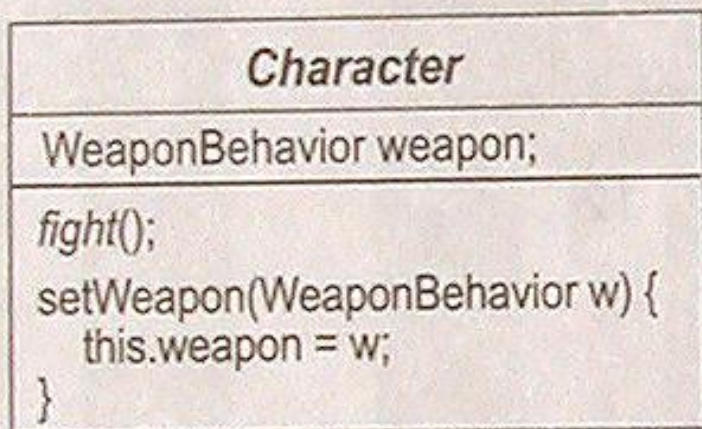


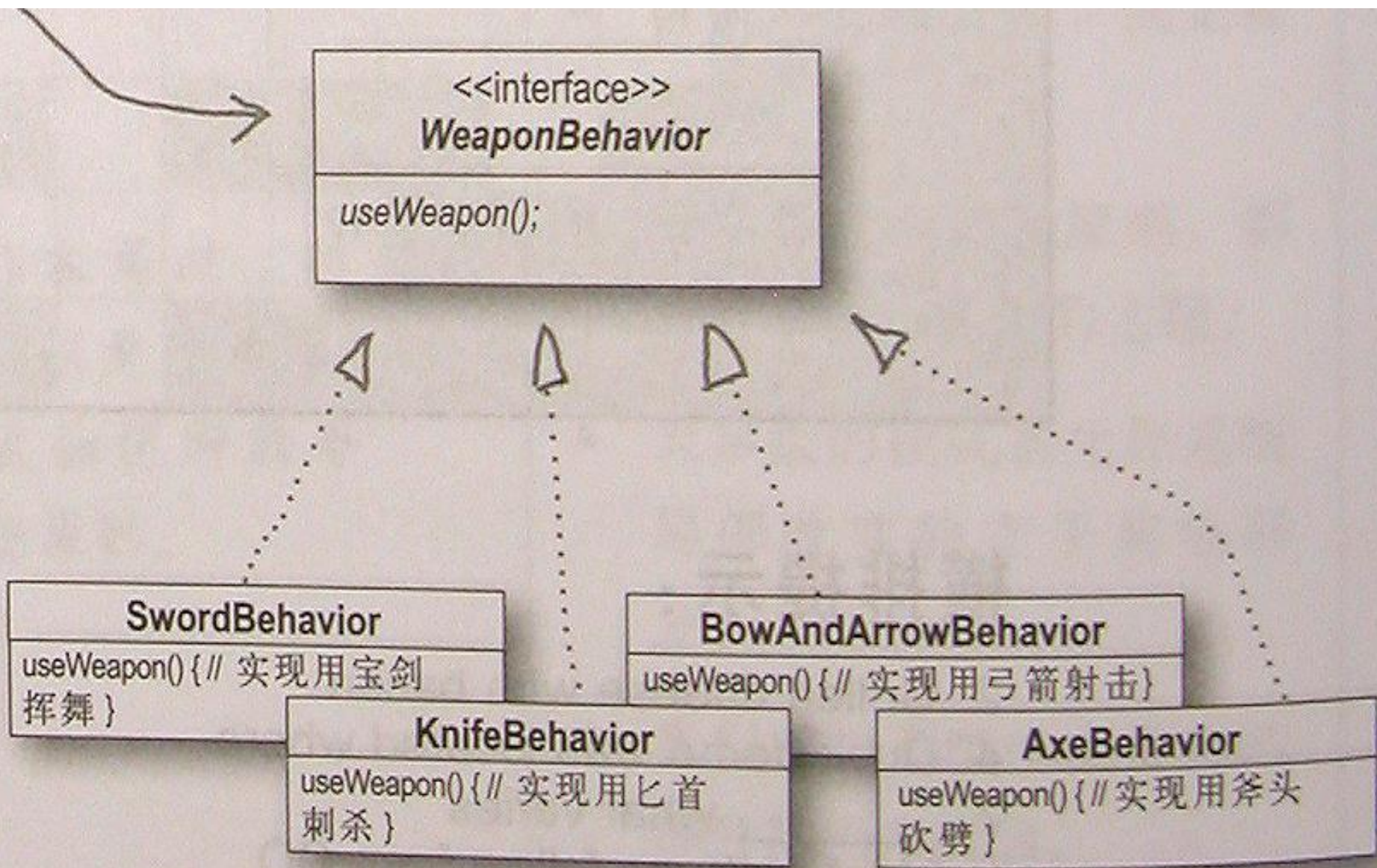
Character（角色）是抽象类，由具体的角色来继承。具体的角色包括：国王（King）、皇后（Queen）、骑士（Knight）、妖怪（Troll）。而Weapon（武器）是接口，由具体的武器来继承。所有实际的角色和武器都是具体类。

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（fight）过程中，会调用到目前武器的useWeapon()方法，攻击其他角色。



抽象





OO基础

OO基础

抽象

封装

多态

继承

← 我们假设你知道OO基础包括了多态的用法、继承就像按契约进行设计、封装是如何运作的。如果你觉得脑袋有一点生锈了，快快拿出你的《Head First Java》复习，然后再把这一章读一遍。



OO原则

OO原则

封装变化

多用组合，少用继承

针对接口编程，不针对实现
编程



我们会在后续的内容中更详细地看看这些原则，还会再多加一些原则到清单上。



OO模式

OO模式

策略模式——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。

阅读本书时，时时刻刻要思考着：模式如何仰赖基础与原则。

学了一个，还有更多！

