第3章 面向对象的设计原则

在第1章中我们了解了设计模式的基本概念，第2章我们掌握了设计模式的UML类图表示方法。本章将学习如何正确地进行面向对象的设计。理解了面向对象的主要机制以及面向对象的设计原则，才能更好地领会设计模式的本质并灵活运用设计模式。

3.1 软件设计的“七宗罪”

在软件开发过程中，常常有这样一种现象：起初我们对开发的系统架构非常清晰，但是随着开发的深入，或者因为功能的增加，或者因为需求的变更，我们可能逐渐偏离原来的设计并且发现开发工作很难进行下去。最后软件即使发生最细微的变化也会带来灾难性的后果，有人把这时的软件比作“坏面包”或者“坏鸡蛋”，这时软件设计的“臭味”就表现出来了。

导致软件设计“臭味”的主要原因是设计人员没有养成良好的分析、设计习惯、不遵循适当的设计原则，以至于出现很多常犯的错误，这些错误使得系统的可扩展性、可重用性、可维护性等变差、阻碍项目进程甚至让项目偏离轨道。我们把软件设计人员常犯的错误归纳为如下七种，称为软件设计“七宗罪”。

1. 僵化性

僵化性(Rigidity)是指难以对软件进行改动，即使是简单的改动。如果单一的改动会导致有依赖关系的模块中的连锁改动，那么设计就是僵化的。必须要改动的模块越多，设计就越僵化。

2. 脆弱性

脆弱性(Fragility)是指在进行一个改动时，程序的许多地方就可能出现问题，即设计易于遭受破坏。并且，往往是出现新问题的地方与改动的地方并没有概念上的关联。

3. 牢固性

牢固性(Immobility)是指设计中包含了对其他系统有用的部分，但要想把这些部分分离出来所需要的努力和风险是巨大的，即设计难以复用。例如，一段代码、函数、模块的功能可以在新模块或者新系统中使用，但是发现这些现存的代码、函数、模块依赖于一大堆其他的东西，以至于很难将它们抽取出来移动到其他地方使用。

4. 粘滞性

有的时候，一个改动可以以保持原有的设计意图和原有的设计框架的方式进行，也可以以破坏原始意图和框架的方式进行。第一种办法无疑会对系统的未来有利，第二种办法是权宜之计，可以解决短期问题，但是会牺牲中长期利益。如果第二种办法比第一种办法容易得多的话，程序员就有可能牺牲长期利益，采取权宜之计，在一个通用的逻辑中建立一种特例，以便解决眼前的问题。一个系统设计，如果总是使得第二种办法比第一种办法来得容易，说明粘滞性(Viscosity)过高。一个粘滞性过高的系统会诱使维护它的程序员采取错误的维护方案。

5. 不必要的重复

不必要的重复(Needless Repetition)是指滥用“剪切”和“粘贴”等操作。“剪切”和“粘贴”等操作也许是有用的文本编辑操作，但却是灾难性的代码编辑操作。大量的重复代码往往是由于开发人员忽略了抽象，从而使系统不易理解。而且，软件中的重复代码，也会使系统的改动变得困难，不易于系统的维护。

6. 不必要的复杂性

不必要的复杂性(Needless Complexity)是指设计中包含了当前没有用的成分，即过分设计。例如，对于逻辑复杂、技术先进的过度追求，导致了技术框架虽看似华丽却复杂难用。再例如，在设计产品功能或界面交互时，过度追求体验完美、需求满足却导致实际体验下降、功能没人用。所以，软件设计应“有所为有所不为”。

7. 晦涩性

晦涩性(Opacity)是指模块难于理解。代码随时间而不断演化，往往会变得越来越晦涩、可读性差。代码晦涩难懂常体现在如下几点：

(1) 代码不良。以单个字母表示的奇怪变量和1000行代码的冗长函数。

(2) 代码的格式不正确或不一致。

(3) 代码中包含冗余代码。

(4) 代码中包含未备注的低层次优化。

(5) 代码逻辑过于复杂。

简言之，上述软件“七宗罪”是因为没有遵循合适的设计原则。那么什么才是一个好的设计呢？一个好的软件设计应该具有如下性质：

(1) 可扩展性(Extensibility)——新的功能可以很容易地加入到系统之中。

(2) 灵活性(Flexibility)——可以容许代码修改平稳，而不会波及其他的模块。

(3) 可插入性(Pluggability)——可以很容易地将一个类抽出去，同时将另一个有同样接口的类加入进来。

目前，面向对象早已成为主流的软件开发方法，而面向对象正是为了解决系统的可维护性、可扩展性、可重用性等才产生的。一个好的软件设计应以面向对象的三大机制(封装、继承、多态)为基础，并遵循坚实(SOLID)的面向对象设计原则。

3.2 面向对象的三大机制

谈到“面向对象”，必须正确理解“对象”这个概念。通俗地说，对象是一件事、一个实体、一个名词，可以说万物皆是对象。一个对象保存了某些信息，并知道如何执行某些操作。对象具有状态，状态是对对象的一个或多个属性的描述（如：一盏灯，“灯是亮的”，这是它的状态。）

在软件开发中，可以从如下层面理解对象的含义：

（1）从概念层面上讲，对象是某种拥有责任的抽象；

（2）从规格层面上讲，对象是一系列可以被其他对象使用的公共接口；

（3）从语言实现层面上讲，对象作为类的实例封装了代码和数据。

面向对象方法论以封装、继承和多态为三大基石。

3.2.1 封装

封装是面向对象的基本特征之一。封装就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或对象操作，对不可信的类或对象进行信息隐藏。封装性是保证软件部件具有优良的模块性的基础。

面向对象的类作为封装良好的模块，将其说明（用户可见的外部接口）与实现（用户不可见的内部实现）显式地分开，其内部实现按其具体定义的作用域提供保护。

封装的好处在于：

（1）良好的封装能够减少耦合（例如实现界面和逻辑分离）；

（2）可以让类对外接口不变，内部可以实现自由的修改；

（3）类具有清晰的对外接口，使用者只需调用，无须关心内部实现；

（4）因为封装的类功能相对独立，因此能更好地实现代码复用；

（5）可保护代码不被无意中破坏，通过私有字段等实现内部。注意：这里的代码保护不是指代码本身的加密，而是对不想外部更改的代码通过私有实现。

例如，一个男人具有姓名、年龄、妻子等私有属性，但是可以向外界提供一些方法设定妻子的信息，获取他的姓名、年龄等。同样，一个女人也具有姓名、年龄、丈夫等私有属性，也可以向外界提供一些设置或获取相关信息的方法。下面用 Java 代码来描述对男人和女人的封装。

```
public class Man {
    //对属性的封装,一个人的姓名、年龄、妻子都是这个对象(人)的私有属性
    private String name;
    private int age;
    private Woman wife;
    //对改人对外界提供方法的封装,可以设定妻子,姓名,年龄也可以获得男人的姓名和年龄
    public void setWife(Woman wife) {
        this.wife = wife;
    }
    public String getName() {
```

```
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
public class Woman {
    //属性封装
    private String name;
    private int age;
    private Man husband;
    //方法封装
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public Man getHusband() {
        return husband;
    }
   public void setHusband(Man husband) {
          this.husband = husband;
     }
}
```

仔细看就会发现，Man 类没有提供 getWife 的方法，这可以理解为男人不想让自己的妻子被外界访问，也就是说，封装可以把一个对象的属性私有，而提供一些可以被外界访问的属性的方法，例如，对于 name 属性，Man 和 Woman 类都有相应的 get 和 set 方法，外界都可以通过这些方法访问和修改；同时对一些该对象不想让外界访问的属性，就不提供其方法，例如 Man 的 wife 属性，就没有 get 方法。

因此，封装的原则就是隐藏对象的属性和实现细节，仅对外提供公共访问方式。需要注意的是，面向对象方法论中提到封装的时候，并没有带宾语，那么这个封装的意义就是不完整的，从另外一个角度来说它是抽象的，封装只在特定的场景中才是有具体意义的。

那么，封装什么呢？对于设计模式来说，大多数模式陈述的一个主题就是“封装变化”。这些变化可以是：

(1) 对象的行为,如工厂方法模式、抽象工厂模式、建造者模式、原型模式、观察者模式、策略模式等。

(2) 类/对象的内部结构,如迭代器模式、桥接模式等。

(3) 对象之间的交互,如享元模式等。

我们将在后续章节通过对各种设计模式的学习加深对“封装变化”的理解。

3.2.2　继承

继承是面向对象方法的另一块基石,它允许创建分等级层次的类,可以理解为一个对象从另一个对象获取属性的过程。通过继承创建的新类称为“子类”或“派生类”。被继承的类称为“基类”、“父类”或“超类”。继承的过程,就是从一般到特殊的过程。

在 Java 中,类的继承是单一继承,也就是说,一个子类只能拥有一个父类。事实上,所有 Java 的类均是由 java.lang.Object 类继承而来的,所以 Object 是所有类的祖先类,而除了 Object 外,所有类必须有一个父类。

继承中最常使用的两个关键字是 extends(即实现继承)和 implements(即接口继承)。这两个关键字的使用决定了一个对象和另一个对象是否是 Is-A(是一个)关系。通过使用这两个关键字,我们能实现一个对象获取另一个对象的属性。例如,使用关键字 extends 实现继承的代码形式如下:

A.java

```
public class A {
    private int i;
    protected int j;

    public void func() {

    }
}
```

B.java

```
public class B extends A {
}
```

以上代码片段说明,B 由 A 继承而来的,B 是 A 的子类。而 A 是 Object 的子类,这里可以不显式地声明。通过使用关键字 extends,子类可以继承父类所有的方法和属性,但是无法使用 private(私有)的方法和属性。作为子类,B 的实例拥有 A 所有的成员变量,但对于 private 的成员变量 B 却没有访问权限,这保障了 A 的封装性。

在类继承接口的情况下,需要使用 implements 关键字,例如:

```
public interface Animal {}

public class Mammal implements Animal{
}

public class Dog extends Mammal{
}
```

可以使用 instanceof 运算符来检验 Mammal 和 dog 对象是否是 Animal 类的一个实例。例如：

```
interface Animal{}

class Mammal implements Animal{}

public class Dog extends Mammal{
   public static void main(String args[]){

      Mammal m = new Mammal();
      Dog d = new Dog();

      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

以上实例编译运行结果如下：

```
true
true
true
```

一般继承基本类和抽象类用 extends 关键字，实现接口类的继承用 implements 关键字。尽管 Java 只支持单继承，但是可以通过接口来体现多继承，如：

```
public class Apple extends Fruit implements Fruit1,Fruit2{}
```

上面的例子一个是实现继承，另一个是接口继承。实现继承尽管实现了代码复用，但这种复用将超类的实现细节暴露给了子类，在某种程度上破坏了封装性。由于超类的内部细节对于子类常常是透明的，所以这种复用是透明的复用，又称“白箱”复用，如果超类发生改变，那么子类的实现也不得不发生改变；而且，从超类继承而来的实现是静态的，不可能在运行时间内发生改变，没有足够的灵活性。因此，在很多情况下实现继承会造成程序的僵化、粘滞、脆弱等，所以，我们需要接口继承。接口代表着抽象，而往往抽象的东西都相对稳定。使用接口继承不但可以解耦两类或多类对象之间的耦合关系，还可以让系统更稳定。因此，在后续章节学习各种设计模式时，我们会发现：很多设计模式都阐述了一个主题——面向接口编程而不是实现编程，例如桥接模式、组合模式、策略模式等。

3.2.3 多态

多态按字面的意思就是“多种状态”。在面向对象语言中，接口的多种不同的实现方式即为多态。方法的重写、重载与动态连接构成多态性。

Java 之所以引入多态的概念，原因之一是它在类的继承问题上和 C++不同，后者允许多继承，这确实给其带来了非常强大的功能，但是复杂的继承关系也给 C++开发者带来了更大的麻烦，为了规避风险，Java 只允许单继承，派生类与基类间有 Is-A 的关系。这样做虽然

保证了继承关系的简单明了，但是势必在功能上有很大的限制，所以，Java 引入了多态性的概念以弥补这点的不足，此外，抽象类和接口也是解决单继承规定限制的重要手段。同时，多态也是面向对象编程的精髓所在。

多态分为设计时多态和运行时多态，例如重载又被称为设计时多态。而对于覆盖或继承的方法，Java 运行时系统根据调用该方法的实例的类型来决定选择调用哪个方法则被称为运行时多态。

关于多态，需要注意如下几个要点：

(1) 使用父类类型的引用指向子类的对象；该引用只能调用父类中定义的方法和变量；

(2) 如果子类中重写了父类中的一个方法，那么在调用这个方法的时候，将会调用子类中的这个方法；

(3) 变量不能被重写(覆盖)，“重写”的概念只针对方法。

多态最主要的作用在于改写对象的行为。利用设计模式在封装对象的行为时，最具有实现价值的就是多态。在以封装对象行为为基本主题的模式中，多态的作用在于它让你更直接地表达行为变化的概念。因此多态的意义更多地在于实现。

3.3　面向对象基本原则

面向对象的三大机制“封装、继承、多态”可以表达面向对象的所有概念，但这三大机制本身并没有刻画出面向对象的核心精神。换言之，既可以用这三大机制做出“好的面向对象设计”，也可以用这三大机制做出“差的面向对象设计”。不是使用了面向对象的语言，就实现了面向对象的设计与开发。根本性的问题在于要明确：为什么要使用面向对象？应该怎样使用三大机制来实现“好的面向对象”？应该遵循什么样的面向对象原则？

S. O. L. I. D 是面向对象设计的五大基本原则的首字母缩写，由罗伯特·C. 马丁等在 20 世纪早期提出，包括表 3.1 所列出的几种原则。当这些原则被同时应用时，程序员开发一个易于维护和扩展的系统就变得更加可能。

表 3.1　面向对象设计的原则

五大基本原则	英 文 全 拼	中 文 名 称
SRP	The Single Responsibility Principle	单一职责原则
OCP	The Open Closed Principle	开放封闭原则
LSP	The Liskov Substitution Principle	里氏替换原则
ISP	The Interface Segregation Principle	接口分离原则
DIP	The Dependency Inversion Principle	依赖倒置原则

3.3.1　单一职责原则

单一职责原则指的是一个类应该仅有一个引起它变化的原因。这是最简单、最容易理解却最不容易做到的一个设计原则。

所谓职责，是指类变化的原因。如果一个类因多于一个的动机而被改变，那么这个类就

具有多于一个的职责。而单一职责原则就是指一个类或者模块应该有且只有一个改变的原因。

之所以需要单一职责原则就是因为在软件设计时会出现以下类似场景：

T 类负责两个不同的职责：职责 P1、职责 P2。当由于职责 P1 需求发生改变而需要修改类 T 时，有可能会导致原本运行正常的职责 P2 功能发生故障。也就是说，职责 P1 和 P2 被耦合在了一起。这种耦合会导致脆弱的设计并影响复用性。

其实，程序设计人员大都清楚应该写出高内聚、低耦合的程序，但是很多耦合常常发生在不经意之间，其原因就是：职责扩散，即因为某种原因，某一职责被分化为颗粒度更细的多个职责了。解决办法就是遵守单一职责原则，将不同的职责封装到不同的类或模块中。

例如，我们在设计一个电子商务系统的时候，通常需要计算订单总价。订单总价的计算通常包含计算货物总价和运费两部分。成熟的设计中，Sale 类用来计算货物总价，不应该再拿来计算运费。因为对于不同的产品、店铺、运送方式，运费的计算规则可能完全不同。

总之，单一职责原则解决的其实是类设计时的职责划分和粒度问题。每个类都是因为具有一定的职责才会存在，但是一个类也不应该分担过多的职责，每一个职责的变化都会引起类的变化。这就好比一个人身兼数职，这些职责可能互不相干。但是一旦有一个职责发生变化，他就必须重新安排自己的工作。类也一样。过多互不相关（或相关性不强）的职责集中在一个类中就会造成高耦合性、代码僵化。因此，应该把不同的职责分开到不同的类中。

3.3.2 开闭原则

开闭原则指的是一个软件实体应当对扩展开放，对修改关闭。即在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展。

“对扩展开放”意味着模块的行为是可以扩展的。当应用的需求改变时，我们可以对模块进行扩展，使其具有满足那些改变的新行为。也就是说，我们可以改变模块的功能。

“对修改关闭”意味着对模块行为进行扩展时，不必改动模块的源代码或者二进制代码。模块的二进制可执行版本，无论是可链接的库、DLL 或者. EXE 文件，都无须改动。

实现开闭原则的关键就在于“抽象”。把系统的所有可能的行为抽象成一个抽象底层，这个抽象底层规定出所有的具体实现必须提供的方法的特征。作为系统设计的抽象层，要预见所有可能的扩展，从而使得在任何扩展情况下，系统的抽象底层都不需修改；同时，由于可以从抽象底层导出一个或多个新的具体实现，可以改变系统的行为，因此系统设计对扩展是开放的。

我们在软件开发的过程中，一直都是提倡需求导向的。这就要求我们在设计的时候，要非常清楚地了解用户需求，判断需求中包含的可能的变化，从而明确在什么情况下使用开闭原则。但是，现实世界是在不断变化的，而且变化的速度越来越快，软件开发中唯一不会变的就是需求永远会变化。在实际开发过程的设计开始阶段，就要罗列出系统所有可能的行为，并把这些行为加入到抽象底层，根本就是不可能的，这样做也是不经济的。因此我们应该面对现实，接受修改，拥抱变化，使我们的代码可以对扩展开放，对修改关闭。

下面通过一个例子来说明开闭原则。

假设现在需要实现一个加法的功能，很简单，如图 3.1 所示。

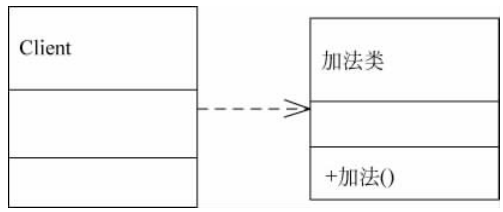

图 3.1　实现加法功能的类图

现在的问题是，需求变了，要求还要实现一个减法的功能，这也很简单，如图 3.2 所示。

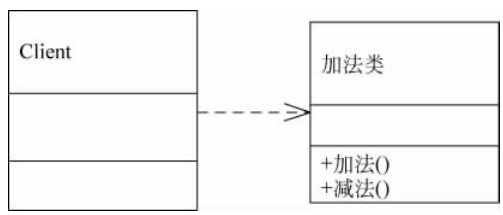

图 3.2　实现加、减法功能的类图

如果需求再变，还要求能实现乘法和除法的功能，依然简单，如图 3.3 所示。

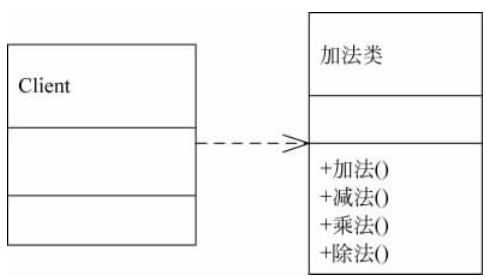

图 3.3　实现加、减、乘、除功能的类图

从如图 3.2 和图 3.3 所示的两步来看，很明显，在需求改变，需要引进新的功能的时候，具体做法是在已有的类的基础上通过新添方法来实现功能，假设在如图 3.2 所示的步骤之后，到图 3.3 的时候发现加法和减法功能最终都没有用到，反而乘法和除法需要用到，那么在图 3.3 的步骤的时候就要推翻之前的实现，在需求发生改变需要引进新的功能的时候，就要推翻整个之前的系统，很明显这样的做法是不可取的，说明设计上出现了问题，这个缺陷明显违反了“开闭原则”。需求总是在变的，如果可能，就要做到尽量不要去修改已有的实现，而应该通过扩展的手段来稳定需求的变动。可以使用“开闭原则”解决上述问题，如图 3.4 所示。

通过上面的例子可以看出：在最初编写代码的时候，我们假设变化不会发生，但最后变化发生的时候，可以通过创建抽象来隔离以后将要发生的同类变化。

“开闭原则”是面向对象设计中“可复用设计”的基础。从“开闭原则”中可以看出，解决问题的关键在于抽象化。对一个事物抽象化，实质上是在概括归纳总结它的本质。抽象让我们抓住最最重要的东西，从更高一层去思考。这降低了思考的复杂度，我们不用同时考虑那么多的东西，并且从抽象化中导出具体化，具体化可以有许多不同的版本，而每个不同的版本可以给出不同的实现。

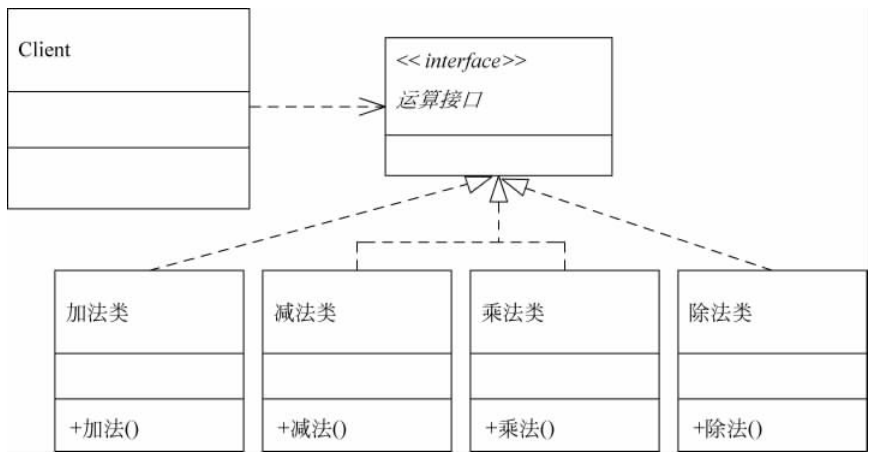

图 3.4　符合开闭原则的设计

3.3.3　里氏替换原则

里氏(Liskov)替换原则是 Barbara Liskov 于 1988 年提出来的，指的是“如果对于类型 S 的每个对象 O1 存在类型 T 的对象 O2，那么对于所有定义了 T 的程序 P 来说，当用 O1 替换 O2 并且 S 是 T 的子类型时，P 的行为不会改变。”通俗地讲，就是子类型能够完全替换父类型，而不会让调用父类型的客户程序从行为上有任何改变。

为什么要引入里氏替换原则呢？众所周知，在面向对象的语言中，继承是必不可少的、非常优秀的语言机制，它有如下优点：

(1) 代码共享，减少创建类的工作量，每个子类都拥有父类的方法和属性。

(2) 提高代码的重用性。

(3) 子类可以形似父类，但又异于父类，“龙生龙，凤生凤，老鼠生来会打洞”是说子拥有父的“种”，“世界上没有两片完全相同的叶子”是指明子与子、子与父的不同。

(4) 提高代码的可扩展性，很多开源框架的扩展接口都是通过继承父类来完成的。

(5) 提高产品或项目的开放性。

但是，任何事物都有两面性，继承也具有如下不容忽视的缺点：

(1) 继承是侵入性的。只要继承，就必须拥有父类的所有属性和方法。

(2) 降低代码的灵活性。子类必须拥有父类的属性和方法，让子类自由的世界中多了些约束。

(3) 增强了耦合性。当父类的常量、变量和方法被修改时，必须要考虑子类的修改，而且在缺乏规范的环境下，这种修改可能带来非常糟糕的结果——大段的代码需要重构。

尽管如此，从整体上来看，使用继承还是利大于弊，怎么才能让“利”的因素发挥最大的作用，同时减少“弊”带来的麻烦呢？解决方案是引入里氏替换原则。从里氏替换原则的定义可以看出：只要父类能出现的地方子类就可以出现，而且替换为子类还不产生任何错误或异常，使用者可能根本就不需要知道是父类还是子类。但是，反过来就不行了，有子类出现的地方，父类未必就能适用。

下面举一个例子来说明里氏替换原则。

“鸵鸟非鸟”是一个理解里氏替换原则的经典的例子。“鸵鸟非鸟”的另一个版本是“企鹅非鸟”，这两种说法本质上没有区别，前提条件都是这种鸟不会飞。生物学中对于鸟类的定义：“恒温动物，卵生，全身披有羽毛，身体呈流线型，有角质的喙，眼在头的两侧。前肢退化成翼，后肢有鳞状外皮，有四趾。”所以，从生物学角度来看，鸵鸟肯定是一种鸟。

我们设计一个与鸟有关的系统，鸵鸟类顺理成章地由鸟类派生，鸟类所有的特性和行为都被鸵鸟类继承。大多数的鸟类在人们的印象中都是会飞的，所以，我们给鸟类设计了一个名字为 fly 的方法，还给出了与飞行相关的一些属性，例如飞行速度(velocity)。

鸟类 Bird：

```
class Bird {
   double velocity;
   public fly() { //I am flying; };

   public setVelocity(double velocity) { this.velocity = velocity; };
   public getVelocity() { return this.velocity; };
}
```

鸵鸟不会飞怎么办？我们就让它扇扇翅膀表示一下吧，在 fly 方法里什么都不做。至于它的飞行速度，不会飞就只能设定为 0 了，于是我们就有了鸵鸟类的设计。

鸵鸟类 Ostrich：

```
class Ostrich extends Bird {
    public fly() { //I do nothing; };
    public setVelocity(double velocity) { this.velocity = 0; };
    public getVelocity() { return 0; };
}
```

好了，所有的类都设计完成，我们把类 Bird 提供给了其他的代码(消费者)使用。现在，消费者使用 Bird 类完成这样一个需求：计算鸟飞越黄河所需的时间。

对于 Bird 类的消费者而言，它只看到了 Bird 类中有 fly 和 getVelocity 两个方法，至于里面的实现细节，它不关心，而且也无须关心，于是给出了实现代码。

测试类 TestBird：

```
class TestBird {
   public calcFlyTime(Bird bird) {
   try{
     double riverWidth = 3000;
     System.out.println(riverWidth / bird.getVelocity());
   }catch(Exception err){
     System.out.println("An error occured!");
   }
   };
}
```

如果我们拿一种飞鸟来测试这段代码，没有问题，结果正确，符合我们的预期，系统输出了飞鸟飞越黄河所需要的时间；如果我们再拿鸵鸟来测试这段代码，结果代码发生了系统除零的异常，明显不符合我们的预期。

对于 TestBird 类而言，它只是 Bird 类的一个消费者，它在使用 Bird 类的时候，只需要根据 Bird 类提供的方法进行相应的使用，根本不会关心鸵鸟会不会飞这样的问题，而且也无须知道。它就是要按照“所需时间＝黄河的宽度/鸟的飞行速度”的规则来计算鸟飞越黄河所需要的时间。

我们得出结论：在 calcFlyTime 方法中，Bird 类型的参数是不能被 Ostrich 类型的参数所代替，如果进行了替换就得不到预期结果。因此，Ostrich 类和 Bird 类之间的继承关系违反了里氏替换原则，它们之间的继承关系不成立，鸵鸟不是鸟。

那么“鸵鸟到底是不是鸟?”，鸵鸟是鸟也不是鸟，这个结论似乎就是个悖论。产生这种混乱有两方面的原因。

原因一：对类的继承关系的定义没有搞清楚。

面向对象的设计关注的是对象的行为，它是使用“行为”来对对象进行分类的，只有行为一致的对象才能抽象出一个类来。我们经常说类的继承关系是一种“Is-A”关系，实际上指的是行为上的“Is-A”关系，可以把它描述为“Act-As”。我们一讲到鸟，就认为它能飞，有的鸟确实能飞，但不是所有的鸟都能飞。问题就是出在这里。如果以“飞”的行为作为衡量“鸟”的标准的话，鸵鸟显然不是鸟；如果按照生物学的划分标准：有翅膀、有羽毛等特性作为衡量“鸟”的标准的话，鸵鸟理所当然就是鸟了。鸵鸟没有“飞”的行为，我们强行给它加上了这个行为，所以在面对“飞越黄河”的需求时，代码就会出现运行期故障。

原因二：设计要依赖于用户的要求和具体环境。

继承关系要求子类要具有基类全部的行为，这里的行为是指落在需求范围内的行为。

A 需求期望鸟类提供与飞翔有关的行为，即使鸵鸟跟普通的鸟在外观上是 100％的相像，但在 A 需求范围内，鸵鸟在飞翔这一点上跟其他普通的鸟是不一致的，它没有这个能力，所以，鸵鸟类无法从鸟类派生，鸵鸟不是鸟。

B 需求期望鸟类提供与羽毛有关的行为，那么鸵鸟在这一点上跟其他普通的鸟是一致的。虽然它不会飞，但是这一点不在 B 需求范围内，所以，它具备了鸟类全部的行为特征，鸵鸟类就能够从鸟类派生，鸵鸟就是鸟。

所有派生类的行为功能必须和使用者对其基类的期望保持一致，如果派生类达不到这一点，那么必然违反里氏替换原则。在实际的开发过程中，不正确的派生关系是非常有害的。伴随着软件开发规模的扩大，参与的开发人员也越来越多，每个人都在使用别人提供的组件，也会为别人提供组件。最终，所有人的开发的组件经过层层包装和不断组合，被集成为一个完整的系统。每个开发人员在使用别人的组件时，只需知道组件的对外裸露的接口，那就是它全部行为的集合，至于内部到底是怎么实现的，无法知道，也无须知道。所以，对于使用者而言，它只能通过接口实现自己的预期，如果组件接口提供的行为与使用者的预期不符，错误便产生了。里氏替换原则就是在设计时避免出现派生类与基类不一致的行为。

简言之，里氏替换原则为继承定义了一个规范，简单地概括为 4 层含义：

(1) 子类必须完全实现父类的方法，且方法对子类是有意义的；

(2) 子类可以有自己的个性；

(3) 覆盖或者实现父类方法时输入参数可以被放大；

(4) 覆盖或者实现父类方法时输出参数可以被缩小。

3.3.4 接口隔离原则

接口隔离原则指的是“客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上”。就是说，“不应该强迫客户依赖于他们不用的方法。”再通俗点说，不要强迫客户使用他们不用的方法，如果强迫客户使用他们不需要使用的方法，那么这些客户就会面临由于这些不使用的方法的改变所带来的改变。

下面举一个违反了接口隔离原则的例子。首先看下面的 Java 代码：

```
// interface segregation principle - bad example

interface IWorker {
    public void work();
    public void eat();
}

class Worker implements IWorker {
    public void work() {
        // ...working
    }
    public void eat() {
        // ... eating in launch break
    }
}

class SuperWorker implements IWorker{
    public void work() {
        //... working much more
    }

    public void eat() {
        //... eating in launch break
    }
}

class Manager {
    IWorker worker;
    public void setWorker(IWorker w) {
        worker = w;
    }
    public void manage() {
        worker.work();
    }
}
```

在这个例子中，我们使用 Manager 类代表一个管理工人的管理者。有两种类型的工人：普通的和高效的，这两种工人都需要吃午饭。但是，现在来了一批机器人，它们同样为公司工作，但是它们不需要吃午饭。一方面 Robot 类需要实现 IWoker 接口，因为它们要工作，另一方面，它们又不需要实现 IWorker 接口，因为它们不需要吃饭。在这种情况下

IWorker 就被认为是一个被污染了的接口。如果我们保持上面那样的代码设计，那么 Robot 类将被迫实现 eat()方法，我们可以写一个哑类它什么也不做（例如它只用一秒钟的时间吃午饭），但是这会对程序造成不可预料的结果（例如管理者看到的报表中显示被带走的午餐多于实际的人数）。

根据接口隔离原则，一个灵活的设计不应该包含被污染的接口。对于这个例子，我们应该把 IWorker 分离成两个接口。

下面是遵循接口隔离原则的代码。通过把 IWorker 分离成两个接口，Robot 类就不需要再被强迫实现 eat()方法。如果需要为 Robot 类添加其他的功能，例如重新充电，我们可以创建一个新的 IRechargeable 接口，其中包含一个重新充电的方法 recharge。

```
//interface segregation principle - good example

interface IWorkable {
    public void work();
}

interface IFeedable{
    public void eat();
}
class Worker implements IWorkable, IFeedable {
    public void work() {
        // ...working
    }

    public void eat() {
        //... eating in launch break
    }
}

class SuperWorker implements IWorkable, IFeedable{
    public void work() {
        //... working much more
    }

    public void eat() {
        //... eating in launch break
    }
}

class Robot implements IWorkable{
    public void work() {
    // ...working
    }
}

class Manager {
    IWorkable worker;
    public void setWorker(IWorkable w) {
```

```
            worker = w;
        }
        public void manage() {
            worker.work();
        }
    }
```

总之，接口隔离原则是对接口进行规范约束，其包含以下四层含义：

(1) 接口尽量要小。这是接口隔离原则的核心定义，不出现臃肿的接口(Fat Interface)。

(2) 接口要高内聚。什么是高内聚？高内聚就是提高接口、类、模块的处理能力，减少对外的交互。要求在接口中尽量少公布 public 方法，接口是对外的承诺，承诺越少对系统的开发越有利，变更的风险也就越少，同时也有利于降低成本。

(3) 定制服务。只提供访问者需要的方法。

(4) 接口设计是有限度的。接口的设计粒度越小系统越灵活，这是不争的事实，但是这也带来结构的复杂化，使得开发难度增加、可维护性降低，这不是一个项目或产品所期望看到的，所有接口设计一定要注意适度，适度的“度”怎么来判断的呢？根据经验和常识判断！

至此，可能很多人会觉得接口隔离原则跟单一职责原则很相似，其实不然。其一，单一职责原则注重的是职责；而接口隔离原则注重对接口依赖的隔离。其二，单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；而接口隔离原则主要约束接口，主要针对抽象，针对程序整体框架的构建。

3.3.5 依赖倒置原则

依赖倒置原则指的是“高层模块不应该依赖于低层模块，二者都应该依赖于抽象；抽象不应该依赖于细节，细节应该依赖于抽象”。

举个生活中的例子。用 ATM 机取过钱的人都知道，只要我们手上有一张银行卡，就可以到各个银行的 ATM 机上去取款，在这个场景中 ATM 机器属于高层模块，我们手上的银行卡属于底层模块。在 ATM 机上提供了一个卡槽插口(接口)，供各种银行卡插入使用，在这里 ATM 机不依赖于具体的哪种银行卡，它只规定了银行卡的规格，只要我们手上的银行卡满足这个规格参数，我们就可以使用它。

在早期面向过程的开发中，上层调用下层，上层依赖于下层，当下层剧烈变化时，上层也要跟着变化，这就会导致模块的复用性降低而且大大提高了开发的成本。

面向对象的开发很好地解决了这个问题，一般情况下抽象的变化概率很小，让用户程序依赖于抽象，实现的细节也依赖于抽象。即使实现细节不断变化，只要抽象不变，客户程序就不需要变化。这大大降低了客户程序与实现细节的耦合度。

例如，图 3.5 就是一个不符合 DIP 原则的例子。

从图 3.5 中可以看出，开关类 ToggleSwitch 依赖于具体的灯 Light 类，使得这两个类耦合紧密，一旦某一盏灯发生变化必然导致 ToggleSwitch 也将随之发生变化。

解决的方法是将 Light 类设计成抽象类，然后让具体的灯类继承自 Light，如图 3.6 所示。

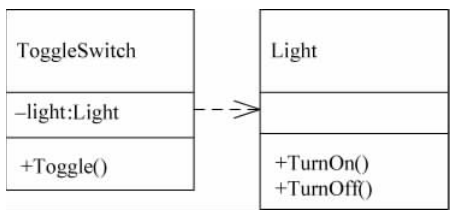

图 3.5 一个不符合 DIP 原则的例子

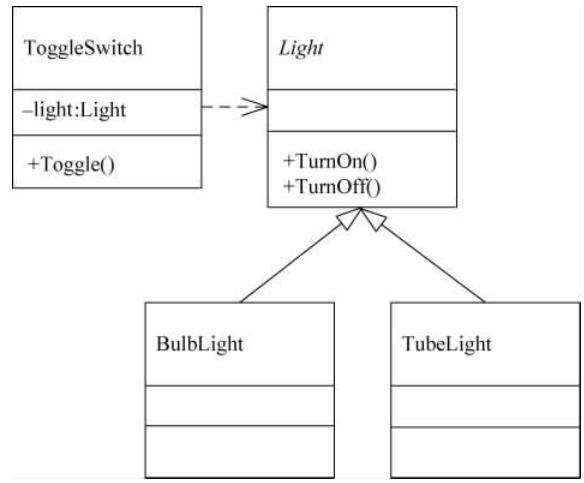

图 3.6 将 Light 类设计成抽象类

这种设计的优点是 ToggleSwitch 依赖于抽象类 Light，具有更高的稳定性，而 BulbLight 与 TubeLight 继承自 Light，可以根据“开放封闭”原则进行扩展。只要 Light 不发生变化，BulbLight 与 TubeLight 的变化就不会影响 ToggleSwitch。

但是这种设计仍然存在问题，在目前的设计中，ToggleSwitch 类可以控制灯，但是控制一台电视就很困难，因为无法让电视机继承自 Light。怎么办呢？可以对上述设计进一步改进，如图 3.7 所示。

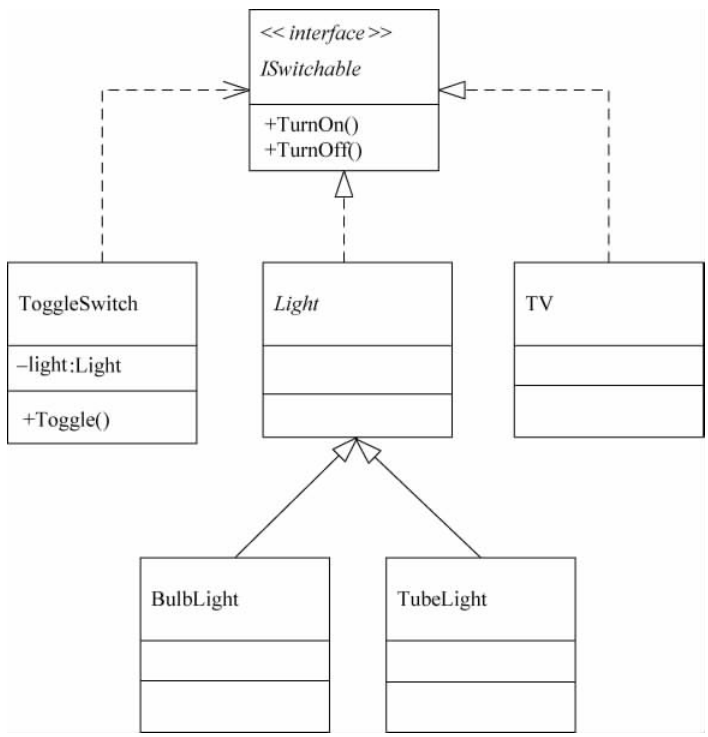

图 3.7 遵循依赖倒置原则的设计

很显然这种设计遵循了依赖倒置原则，更为通用、更为稳定。

综上所述，依赖倒置原则的核心就是要我们面向接口编程，理解了面向接口编程，也就理解了依赖倒置。在实际编程中，一般需要做到如下 3 点：

(1) 低层模块尽量都要有抽象类或接口，或者两者都有。

(2) 变量的声明类型尽量是抽象类或接口。

(3) 使用继承时遵循里氏替换原则。

3.4 本章习题

一、选择题

1. 开闭原则的含义是一个软件实体(　　)。

A. 应当对扩展开放，对修改关闭　　B. 应当对修改开放，对扩展关闭

C. 应当对继承开放，对修改关闭　　D. 以上都不对

2. 要依赖于抽象，不要依赖于具体。即针对接口编程，不要针对实现编程，是(　　)的表述。

A. 开闭原则　　B. 接口隔离原则

C. 里氏替换原则　　D. 依赖倒置原则

3. 以下对“开闭原则”的一些描述错误的是(　　)。

A. “开闭原则”与“对可变性的封装原则”没有相似性

B. 找到一个系统的可变元素，将它封装起来

C. 对修改关闭是其原则之一

D. 从抽象层导出一个或多个新的具体类可以改变系统的行为，是其原则之一

4. (　　)是实现“一种接口，多种方法”的机制。

A. 抽象　　B. 封装　　C. 多态　　D. 继承

5. 下列属于面向对象基本原则的是(　　)。

A. 继承　　B. 封装　　C. 里氏代换　　D. 都不是

二、填空题

1. 子类可以在任何地方替换它的父类。这指的是面向对象的(　　)原则。

2. 单一职责的含义是：类的职责单一，引起类变化的(　　)单一。

3. 依赖于(　　)是面向对象设计的精髓，也是依赖倒置原则的核心。

4. 软件设计说到底追求的目标就是封装(　　)、降低(　　)。

5. 多态最主要的作用在于(　　)。

三、判断题

1. 接口是对外的承诺，承诺越少对系统的开发越有利，变更的风险也就越少，同时也有利于降低成本。

2. 里氏替换原则指的是父类型和子类型之间可以互相替换。

3. 依赖于抽象，就是依赖于细节并对实现编程。

4. 只有遵循了里氏替换原则，才能保证继承复用是可靠的。

5. 对修改关闭对扩展开放使得软件无法修改、难以维护。

四、应用题

1. 某画图程序目前需要绘制矩形和圆形，未来版本可能增加诸如三角形等更多的图形。要求设计符合开放封闭原则。请回答以下问题：

(1) 分析上诉应用需求，为该程序当前需求设计对象类，确定对象类之间的关系，以UML类图的形式说明对象类设计，并用注释说明每个类代表哪个事物。

(2) 用Java类定义等形式给出每个类中属性和方法的详细声明，并给出必要的注释。不需要给出方法内部的具体实现代码。

(3) 说明你的设计符合开放封闭原则的理由。

2. 举例说明面向对象技术的三大机制。