第 3 章

面向对象设计原则

人们在软件应用中总结出许多宝贵的经验，经过归纳整理形成了若干个设计原则。设计原则是解决问题的基本思路，本章学习要点如下：

- 理解开闭原则是设计模式的基本原则；
- 掌握里氏代换原则和依赖倒置原则的要点；
- 掌握合成-聚合复用原则、单一职责原则、迪米特法则和接口隔离原则的要点。

3.1　面向对象设计原则的概述

面向对象设计原则为支持可维护性复用而诞生，这些原则蕴含在很多设计模式中，它们是从许多设计方案中总结出的指导性原则。设计原则也是学习软件设计模式（以下简称设计模式）的基础，每种设计模式都会符合若干设计原则。

面向对象的设计原则如表 3.1.1 所示。

表 3.1.1　面向对象的设计原则

名　称	简　介	重 要 性
开闭原则	软件实体对扩展是开放的，但对修改是关闭的，即在不修改软件实体的基础上扩展其功能	★★★★★
里氏代换原则	所有引用基类的地方必须能透明地使用其子类对象。或者说，一个可以接受基类对象的地方必然可以接受一个子类对象	★★★★★
依赖倒置原则	针对抽象层编程，而不应针对具体类编程	★★★★★
合成-聚合复用原则	在系统中应尽量多地使用组合和聚合的关联关系，少使用甚至不使用继承关系	★★★★
单一职责原则	类的职责要单一，不能将太多的职责放在一个类中	★★★★
迪米特法则	一个软件实体对其他实体的引用应越少越好。或者说，如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用，而是通过引入一个第三者发生间接交互	★★★
接口隔离原则	使用多个专门的接口来取代一个统一的接口	★★

注意：

（1）设计模式是针对重复出现的相似问题的解决方案。

（2）开闭原则是面向对象设计的根本原则，也是一个目标，并没有指明任何手段，而其他原则则是实现开闭原则的手段。

3.2 开 闭 原 则

开放封闭原则（Open-Closed Principle，OCP）简称开闭原则，是指一个软件实体应当对扩展开放、对修改关闭，即在设计一个模块时，应当使这个模块能在不被修改的前提下进行扩展。它是面向对象的可复用设计的基石。

随着软件规模的扩大，软件维护成本也变得越来越高。当软件系统需要面对新的需求时，应该尽量保证系统的设计框架是稳定的。那么如何判断软件设计水平的优劣呢？

（1）是否僵硬、难以应对改变。

（2）是否脆弱、改动一处就会影响其他无关地方。

（3）是否难以重用。

（4）是否存在不必要的重复。

（5）是否包含没有用的过度设计。

（6）是否很难阅读和理解，无法表达开发者的设计意图。

如果一个软件设计符合开闭原则，就可以非常方便地对系统进行扩展，而且在扩展时无须修改现有代码，使软件系统在拥有适应性和灵活性的同时，具备较好的稳定性和延续性。

注意：

（1）在开闭原则的定义中，软件实体可以指一个软件模块、一个由多个类组成的局部结构或一个独立的类。

（2）实现开闭原则的优点在于可以在不改动原有代码的前提下实现程序的扩展功能，既增加了程序的可扩展性，也降低了程序的维护成本。

为了满足开闭原则，需要对系统进行抽象化设计，抽象化是开闭原则的关键。在 Java 和 C#等编程语言中，可以为系统定义一个相对稳定的抽象层，将不同的行为移至具体的实现层中完成。

很多面向对象编程语言都提供了接口、抽象类等编程机制，可以先通过它们定义系统的抽象层，再通过具体类来进行扩展。修改系统时，无须对抽象层进行任何改动，通过增加具体类就可以实现在不修改已有代码的基础上扩展系统的功能，以达到开闭原则的要求。

本书将在第 4 章中介绍的工厂模式（工厂方法模式）就符合开闭原则。

3.3 里氏代换原则

里氏代换原则（Liskov Substitution Principle，LSP）是以提出者 Barbara Liskov 的名字命名的，它要求所有引用基类的地方必须能透明地使用其子类的对象，其原始定义如下。

如果对每一个类型为 T1 的对象 o1，都有类型为 T2 的对象 o2，当 T1 定义的程序 P 中所有的对象 o1 都代换成 o2 时，程序 P 的行为并不发生变化，则称类型 T2 是类型 T1 的子类型。

简单地说，一个程序 P(T1)，如果将 T1 替换为 T2 后，结果为 P(T1) = P(T2)，则称 T2 是 T1 的子类型。

注意：透明就意味着不感知，不受任何影响。

里氏代换原则要求在一个软件系统中，子类可以替换任何基类，且代码还能正常工作。

里氏代换原则是继承复用的基石。只有当子类可以替换父类，且软件单位的功能不受影响时，父类才能真正被复用，而子类也能够在基类的基础上增加新的行为。

里氏代换原则说明，在软件中将一个基类对象替换成其子类对象，程序将不会产生任何错误和异常，反过来则不成立。即一个软件实体能够使用一个子类对象，却不一定能够使用基类对象。例如，我喜欢动物，那我一定喜欢狗，因为狗是动物的子类。但是，我喜欢狗，不能据此断定我喜欢动物。因为我并不喜欢老鼠，尽管它也是动物。

再如有两个类，一个是 BaseClass 类，另一个是 SubClass 类，并且 SubClass 类是 BaseClass 类的子类。如果一个方法 method 可以接受一个 BaseClass 类型的基类对象 base 为参数，那么它必然可以接受一个 BaseClass 类型的子类对象 sub，即参数 method(sub)能够正常运行。反过来，这种代换就不一定能成立。

里氏代换原则是实现开闭原则的重要方式之一。由于使用基类对象的地方都可以使用子类对象，因此，在类方法设计时，方法参数应尽量使用抽象类型（接口或抽象类）。

3.4　依赖倒置原则

依赖倒置原则（Dependency Inversion Principle，DIP）是指抽象不应该依赖于具体类，而具体类应该依赖于抽象，即应针对接口编程，而不是针对实现编程。

依赖倒置原则要求在程序代码中传递参数时或在关联关系中，应尽量引用层次高的抽象层类，即使用接口和抽象类进行变量类型声明、参数类型声明、方法返回类型声明，以及数据类型的转换等，而不要用具体类来做这些事情。

注意：倒置概念是相对于建筑用语而言的，表达了高层依赖于低层的关系。

为了确保该原则的应用，一个具体类应只实现接口或抽象类中声明过的方法，而不要给出多余的方法。否则，将无法调用子类中新增的方法。

在引入抽象层后系统可具有很好的灵活性，因此在程序中使用抽象层进行编程时，可将具体类名称写在配置文件中。如果系统行为发生变化，则只需要对抽象层进行扩展，并修改配置文件，而无须修改原有系统的源代码即可实现在不修改的情况下扩展系统功能，以满足开闭原则的要求。

通过抽象来搭建框架，建立类和类的关联，以减少类间的耦合性。以抽象搭建的系统要比以具体实现搭建的系统更加稳定，且扩展性更高，同时也便于维护。

使用接口和抽象类进行变量类型声明、参数类型声明、方法返回类型声明，以及数据类型的转换等，而不是使用具体的类。在需要时，可将具体类的对象通过依赖注入（Dependency Injection，DI）的方式注入其他对象中。

注意：依赖注入是 Spring 框架的核心功能。

3.5　合成-聚合复用原则

合成-聚合复用原则（Composite/Aggregate Reuse Principle，CARP）是指在一个新的对象里使用一些已有的对象，使之成为新对象的一部分；新的对象通过向这些对象委派以达到复

用的目的。应首先使用合成-聚合功能，它能使系统变得灵活，其次再考虑继承，以达到复用的目的。

针对 1.1.3 节提出的问题——庞大的跨平台图像浏览系统，如果使用合成-聚合复用原则，可实现两个维度的独立变化，以解决类继承引起的类爆炸问题。改写后的代码如下：

```
package crp;
interface Image{    //图像接口
    public String parseMatrix();    //抽象的解析方法
}
abstract class OS{      //操作系统抽象类
    private Image image;      //聚合
    public Image getImage() {
        return image;
    }
    public void setImage(Image image) {
        this.image = image;
    }
    abstract public void show();    //抽象方法
}
class BMPImage implements Image{
    //实现接口方法
    public String parseMatrix() {
        return "按 BMP 格式解析";
    }
}
class JPGImage implements Image{
    //实现接口方法
    public String parseMatrix() {
        return "按 JPG 格式解析";
    }
}
class WindowsOS extends OS{
    @Override
    public void show() {
        System.out.println("在 Windows 环境下显示"+super.getImage().parseMatrix()+"图像文件");
    }
}
public class ImageBrowser2 {    //客户端
    public static void main(String[] args) {
        //面向抽象编程，两个维度可以独立变化
        Image image = new BMPImage();
        OS os = new WindowsOS();
        os.setImage(image);    //搭桥
        os.show();
```

```
    }
}
```

注意：

（1）从纵向考虑：抽象类 OS 包含 WindowsOS 等子类，接口 Image 包含 BMPImage 等实现类。从横向考虑：OS 聚合接口包含 Image 类型的对象。

（2）OS 的子类（如 WindowsOS）在重写基类的抽象方法时，调用了聚合对象的方法。

（3）新增（或删除）抽象类的子类或接口的实现类，对已有的类没有任何影响。

（4）客户端代码可以任意使用 OS 的子类和 Image 的实现类。

3.6　单一职责原则

单一职责原则（Single Responsibility Principle，SRP）是指一个类最好只做一件事，只有一个引起变化的原因。

单一职责强调的是职责的分离。如果一个类承担的职责过多，即把这些职责耦合在一起，就会削弱或抑制这个类完成其他职责的能力。这种耦合会导致脆弱的设计，在一定条件下，设计会遭受意想不到的破坏。

一个未使用单一职责原则的程序如下：

```
package singleresponsibility0;

public class UnSingleResponsibility {    //客户端
    public static void main(String[] args) {
        Vehicle vehicle = new Vehicle();
        vehicle.run("汽车");
        vehicle.run("飞机");    //运行时产生逻辑错误
        vehicle.run("轮船");    //运行时产生逻辑错误
    }
}

class Vehicle{    //交通工具
    public void run(String v){    //违反单一原则
        System.out.println(v+"在公路上行驶...");
    }
}
```

运行结果会出现飞机和轮船在公路上行驶的逻辑错误，是因为类 Vehicle 的职责太宽泛。对上面问题的改进措施是，将交通工具划分为 3 种不同的类型，相应的代码如下：

```
package singleresponsibility1;

public class SingleResponsibility {    //客户端
    public static void main(String[] args) {
        new RoadeVehicle().run("汽车");
        new AirVehicle().run("飞机");
        new WaterVehicle().run("轮船");
    }
}

class RoadVehicle{
```

```
        public void run(String s) {
            System.out.println(s+"在公路上行驶...");
        }
    }
    class AirVehicle{
        public void run(String s) {
            System.out.println(s+"在空中飞行...");
        }
    }
    class WaterVehicle{
        public void run(String s) {
            System.out.println(s+"在水上行驶...");
        }
    }
```

坚持单一职责原则的优点如下所述。

（1）可降低类的复杂性。实现每项职责都有清晰明确的定义。

（2）由于复杂性降低，可读性自然就提高了。

（3）随着可读性的提高，维护自然就容易了。

（4）可降低变更引起的风险。系统的扩展对已有的类没有影响。

3.7 迪米特法则

迪米特法则（Principle of Least Knowledge，PLK）是指一个软件实体应当尽可能少地与其他实体发生相互作用。

迪米特法则又叫最少知道原则，来自 1987 年美国东北大学（Northeastern University）的一个名为 Demeter 的研究项目。通俗地说，就是一个类对自己依赖的类知道得越少越好。对于被依赖的类来说，无论逻辑多么复杂，都应将逻辑封装在类的内部，对外除了提供 public 方法，不对外泄露任何信息。

迪米特法则还有一个更简单的定义，即只与直接的朋友通信。一个对象如果能满足下列条件之一，就是当前对象的“朋友”，否则就是“陌生人”。

- 当前对象（this）；
- 以参量形式传入当前对象方法中的对象；
- 当前对象的实例变量直接引用的对象；
- 当前对象的实例变量如果是一个聚集，那么聚集中的元素也都是朋友；
- 当前对象所创建的对象。

迪米特法则的示意图如图 3.7.1 所示。

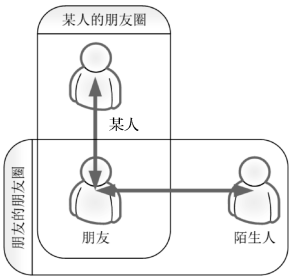

图 3.7.1　迪米特法则的示意图

注意：外观模式（详见 5.1.1 节）是迪米特法则的典型应用。其中，外观类角色充当了客户端和子系统间的第三者。

3.8　接口隔离原则

接口隔离原则（Interface Separate Principle，ISP）是指使用多个专门的接口比使用单一的总接口要好。也就是说，一个类对另一个类的依赖性应当建立在最小的接口上。接口隔离原则的根本在于不要强迫客户端程序依赖其不需要使用的方法。

根据接口隔离原则，当一个接口太大时，就需要将它分割成一些小的接口，让使用该接口的客户端仅需知道与之相关的方法即可，每个接口都应承担一种相对独立的角色。

在使用接口隔离原则时，需要注意控制接口的粒度，其接口不能太小，如果太小将会导致系统中接口泛滥，不利于维护。当然接口也不能太大，太大的接口将违背接口隔离原则，其灵活性较差，使用起来很不方便。

注意：

（1）接口隔离原则是对接口而言的，而单一职责原则则是对类而言的。

（2）避免同一个接口内包含不同类型的职责方法。

（3）只有接口责任划分明确，才符合高内聚、低耦合的思想。

习　题

一、判断题

1．一个软件实体如果使用的是一个子类的话，那么一定适用于其父类。

2．实现开闭原则的关键在于抽象化，抽象化是面向对象设计的第一个核心本质。

3. 将已有对象注入新对象中，使新对象可以调用已有对象的方法，从而实现行为的复用，这是迪米特法则的体现。

4．根据单一职责原则，一个类最好定义一个方法。

5．在软件中，将一个基类对象替换成其子类对象，程序不会产生任何错误，这是依赖倒置原则的体现。

6．陌生的类最好不要作为局部变量的形式出现在类的内部，这是接口隔离原则的体现。

7. 外观模式和中介者模式都体现了迪米特法则的应用。

二、选择题

1．下列叙述中，错误的是____。

A．要针对接口编程，不要针对实现编程

B．上层模块应该依赖于底层模块的细节

C．一个优良的系统设计，应强调模块间保持低耦合、高内聚的关系

D．一个软件实体应当对扩展开放，对修改关闭

2．要依赖于抽象，不要依赖于具体类，即要针对接口编程，不要针对实现编程。这体现____设计原则。

A．开闭原则　　B．接口隔离原则

C．里氏代换原则　　D．依赖倒置原则

3．以下关于面向对象设计的描述中，错误的是____。

A．抽象不应依赖于细节

B．细节可以依赖于抽象

C．高层模块不应依赖于低层模块

D．高层模块无法不依赖于低层模块

4．下面关于面向对象的描述中，正确的是____。

A．针对接口编程，而不是针对实现编程

B．针对实现编程，而不是针对接口编程

C．接口与实现不可分割

D．优先使用继承而非组合，因为组合破坏了封装性

5．在面向对象分析与设计中，____是指子类可以替换父类，并出现在父类能够出现的任何地方。

A．开闭原则　　B．里氏代换原则

C．依赖倒置原则　　D．单一职责原则

6．在迭代器模式中，将数据存储与数据遍历分离。数据存储由聚合类负责，数据遍历由

迭代器负责，这种设计方案是____的具体应用。

A．依赖倒置原则　　B．接口隔离原则

C．单一职责原则　　D．合成-聚合复用原则

三、填空题

1．在面向对象的设计原则中，_____原则是最重要的，能体现软件设计的总体要求。

2．_____设计原则强调对扩展开放，对修改关闭。

3．“不要和陌生人说话”是_____设计原则的通俗表述。

4．_____原则指出：一个类最好只做一件事，只有一个引起变化的原因。本原则强调职责的分离。如果一个类承担的职责过多，就等于把这些职责耦合在了一起。一个职责的变化可能会削弱或抑制这个类完成其他职责的能力。这种耦合会导致脆弱的设计，当变化发生时，设计会遭受意想不到的破坏。

5．_____原则是说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，要针对接口编程，不要针对实现编程。也就是说，应当使用接口和抽象类进行变量类型声明、参数类型声明、方法返回类型说明，以及数据类型的转换等，而不要用具体类。

实　验

一、实验目的

1．掌握开闭原则、里氏代换原则和依赖倒置原则及其联系。
2．掌握单一职责原则和接口隔离原则。
3．掌握合成-聚合复用原则。
4．掌握迪米特法则。

二、实验内容及步骤

【预备】访问上机实验网站 http://www.wustwzx.com/jdp/index.html，下载本章实验内容的案例，解压后得到文件夹 ch03。

1．开闭原则、里氏代换原则和依赖倒置原则的案例分析

（1）在 Eclipse 中，导入案例项目 sy2_DesignPrinciple，并选择 Outline 视图。
（2）查看包 no1_ocp_lsp_dip 里文件 Test.java 定义的一个抽象类及其两个子类。
（3）查看主类 Test 中 main()方法测试代码后的运行程序。
（4）验证可任意将 animal 定义为 Dog 或 Cat 类型，程序仍能正常运行。
（5）验证可将抽象类 Animal 改写为接口类型。
（6）验证增加新的子类（或实现类），不必修改已有类或接口，即符合开闭原则。
（7）体会里氏代换原则和依赖倒置原则是实现开闭原则的手段。

2．单一职责原则的案例分析

（1）查看包 no2_singleresponsibility0 里文件 UnSingleResponsibility.java 定义的两个类 UnSingleResponsibility 和 Vehicle。
（2）确认类 Vehicle 的方法 run()定义的职责过多。
（3）依次查看包 no2_singleresponsibility1 里文件 SingleResponsibility.java 定义的 3 个类 RoadVehicle、AirVehicle 和 WaterVehicle，确认其均具有单一职责。
（4）验证新增一个交通工具类，但不需要修改原有类，即符合开闭原则。

3．合成-聚合复用原则的案例分析

（1）查看包 no3_ImageBrowser 里文件 ImageBrowser.java 定义的类及其依赖关系。
（2）分析使用继承可能会引起类的爆炸。
（3）查看包 no2_ImageBrowser2 里文件 ImageBrowser2.java 定义的类及其关系。
（4）验证新增抽象类的子类或接口的实现类，对已有类没有任何影响，并做运行测试。
（5）分析是否坚持了接口隔离原则和迪米特法则。

三、实验小结及思考

（总结关键的知识点、上机实验中遇到的问题及其解决方案。）