普通高等教育"十二五"规划教材

软件设计原则与模式

郭双宙 ◎ 等编著

配电子课件

普通高等教育“十二五”规划教材

软件设计原则与模式

郭双宙　等编著

机械工业出版社

本书分为两大部分，共4章：第一部分（第1章）介绍软件设计原则，并简单介绍UML和设计模式；第二部分（第2~4章）详细介绍27种设计模式，每种设计模式都有一个与之对应的、浅显易懂的例子作为引子，并有详细的UML结构设计图和相对应的可运行程序，以帮助读者理解所学模式。

本书的主要特点是简单易懂，把设计模式的学习门槛降到最低，使初学者更容易理解和掌握27种设计模式。书中的每个程序都力求简洁明了，并采用最新的、成熟的Java技术编写，易学易用。

本书适合作为本科及高职院校软件专业的设计模式课程教材，也可供从事软件工程的初中级软件开发人员参考使用。

为方便教学，本书配备电子课件等教学资源。凡选用本书作为教材的教师均可登录机械工业出版社教育服务网 www. cmpedu. com 免费下载。如有问题请致信 cmpgaozhi@ sina. com，或致电 010-88379375 联系营销人员。

图书在版编目（CIP）数据

软件设计原则与模式 / 郭双宙等编著. —北京：机械工业出版社，2015. 8

普通高等教育“十二五”规划教材

ISBN 978 - 7 - 111 - 51002 - 4

Ⅰ. ①软…　Ⅱ. ①郭…　Ⅲ. ①软件设计-高等学校-教材　Ⅳ. ①TP311. 5

中国版本图书馆 CIP 数据核字（2015）第 172052 号

机械工业出版社（北京市百万庄大街 22 号　邮政编码 100037）

策划编辑：刘子峰　　责任编辑：刘子峰　陈瑞文

责任校对：王　欣　　封面设计：陈　沛

责任印制：李　洋

涿州市京南印刷厂印刷

2015 年 8 月第 1 版 · 第 1 次印刷

184mm × 260mm · 15. 5 印张 · 368 千字

0001 - 3000 册

标准书号：ISBN 978 - 7 - 111 - 51002 - 4

定价：33.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务

服务咨询热线：（010）88379833

读者购书热线：（010）88379649

网络服务

机 工 官 网：www. cmpbook. com

机 工 官 博：weibo. com/cmp1952

教育服务网：www. cmpedu. com

金 书 网：www. golden-book. com

封面无防伪标均为盗版

前　言

模式是对一种经常发生的问题提出的一种解决方案，这种方案经过无数人的测试和使用，经过千锤百炼后几乎是无懈可击的。现实情况中，很多人不使用模式是因为不了解其优点，认为这些模式很复杂。其实，设计模式的“复杂”就在于其要构造一个“万能钥匙”，目的是提出一种对所有锁的开锁方案。很多程序员在接触设计模式后，都有一种相见恨晚的感觉，有人形容学习了设计模式后感觉自己好像已经脱胎换骨，达到了新的境界；还有人甚至把是否了解设计模式作为划分程序员水平的标准。

本书的主要特色是简单易懂，把设计模式的学习门槛降到最低，目的就是让读者一看就会。同时，通过有趣的例子引入每种设计模式的含义，使初学者更容易理解和掌握27种设计模式。

本书分为两大部分，共4章：第一部分（第1章）介绍软件设计原则，并简单介绍UML和设计模式；第二部分（第2～4章）详细介绍27种设计模式，每种设计模式都有一个与之对应的、浅显易懂的例子作为引子，并有详细的UML结构设计图和相对应的可运行程序。

IT行业有个说法：“没写过10万行代码，就不要说你会一门语言”。本书提供的上万行代码可使读者既熟悉Java语言，又掌握用Java语言描述的27种设计模式，可谓一举两得。

本书由郭双宙、敖山、黄海波和郑哲共同编写。在编写过程中，参考并借鉴了一些相关书籍中的经典案例，在此一并向作者表示感谢。

由于作者水平有限，书中不足之处在所难免，恳请广大读者批评指正。

编　者

目 录

前言

第1章 软件设计原则与UML简介/1

1.1 “开—闭”原则/2

1.2 里氏代换原则/3

1.3 依赖倒置原则/4

1.4 接口隔离原则/4

1.5 合成/聚合复用原则/7

1.6 迪米特法则/8

1.7 单一职责原则/13

1.8 UML简介/17

1.9 设计模式简介/20

第2章 创建型模式/21

2.1 简单工厂模式/21

2.1.1 工厂模式的形态/21

2.1.2 简单工厂模式的结构/22

2.1.3 简单工厂模式的实现/23

2.1.4 简单工厂模式的使用实例/24

2.1.5 简单工厂模式的优点与缺点/27

2.2 工厂方法模式/31

2.2.1 工厂方法模式的结构/32

2.2.2 工厂方法模式的实现/34

2.2.3 工厂方法模式的实际应用/37

2.3 抽象工厂模式/42

2.3.1 抽象工厂模式的结构/44

2.3.2 抽象工厂模式的起源/46

2.3.3 抽象工厂模式的优点与缺点/48

2.3.4 抽象工厂模式的实现/48

2.4 单例模式/49

2.4.1 单例模式的结构/50

2.4.2 单例模式的类型/50

2.4.3 单例模式的进阶/53

2.5 多例模式/55

2.5.1 多例模式的结构/57

2.5.2 多例模式的实现/58

2.6 建造模式/60

2.6.1 建造模式的适用场景/60

2.6.2 建造模式的特点/61

2.6.3 建造模式的结构/61

2.7 原型模式/66

2.7.1 原型模式的结构/67

2.7.2 原型模式的优点与缺点/74

第3章 结构型模式/76

3.1 适配器模式/76

3.1.1 适配器模式的结构/76

3.1.2 适配器模式的实现/80

3.1.3 适配器模式的优点与缺点/83

3.2 缺省适配器模式/84

3.2.1 缺省适配器模式的结构/85

3.2.2 缺省适配器模式的实现/86

3.3 合成模式/88

3.3.1 合成模式的结构/89

3.3.2 合成模式的实现/95

3.4 装饰模式/98

3.4.1 装饰模式的结构/98

3.4.2 装饰模式的实现/101

3.4.3 装饰模式的简化/103

3.4.4 装饰模式的进阶/104

3.5 代理模式/106

3.5.1 代理模式的结构/107

3.5.2 代理模式的实现/108

3.6 享元模式/110

3.6.1 享元模式的结构/110

3.6.2 享元模式的优点与缺点/117

试读结束：需要全本请在线购买： www.ertongbook.com

3.7　门面模式/119
3.7.1　门面模式的结构/120
3.7.2　门面模式的实现/120
3.7.3　门面模式的进阶/122
3.8　桥梁模式/126
3.8.1　桥梁模式的定义/128
3.8.2　桥梁模式的结构/129
3.8.3　桥梁模式的优点/132
3.8.4　桥梁模式的实现/132

第4章　行为型模式/135
4.1　不变模式/135
4.1.1　“不变”和“只读”的区别/136
4.1.2　不变模式的结构/136
4.1.3　不变模式在Java中的应用/137
4.1.4　不变模式的安全应用/137
4.1.5　不变模式的优点与缺点/138
4.2　策略模式/138
4.2.1　策略模式的结构/139
4.2.2　策略模式的特点/140
4.2.3　策略模式的实现/141
4.2.4　策略模式的优点与缺点/142
4.3　模板方法模式/143
4.3.1　模板方法模式的结构/144
4.3.2　模板方法模式中的方法/145
4.3.3　模板方法模式的实现/147
4.3.4　模板方法模式的进阶/148
4.4　观察者模式/151
4.4.1　观察者模式的结构/151
4.4.2　观察者模式的模型/153
4.4.3　观察者模式的实现/157
4.4.4　推模型和拉模型的比较/160
4.5　迭代子模式/161
4.5.1　迭代子模式的结构/161
4.5.2　宽接口和窄接口/162
4.5.3　迭代子模式的实现/163
4.5.4　迭代子模式的优点与缺点/169
4.6　责任链模式/169
4.6.1　责任链模式的结构/169
4.6.2　责任链模式的实现/172
4.7　命令模式/174
4.7.1　命令模式的结构/175
4.7.2　命令模式的实现/175
4.7.3　命令模式的解析/177
4.7.4　命令模式的优点/178
4.8　备忘录模式/180
4.8.1　备忘录模式的结构/180
4.8.2　备忘录模式的实现/181
4.8.3　多重检查点/187
4.8.4　“自述历史”模式/191
4.9　状态模式/193
4.9.1　状态模式的结构/194
4.9.2　状态模式的实现/195
4.9.3　状态模式的解析/196
4.10　专题：分派/199
4.10.1　分派的概念/199
4.10.2　静态分派和动态分派/199
4.10.3　单分派和多分派/202
4.11　访问者模式/208
4.11.1　访问者模式的结构/208
4.11.2　访问者模式的实现/209
4.11.3　访问者模式的分析/212
4.11.4　访问者模式的进阶/217
4.11.5　访问者模式的优点与缺点/222
4.12　解释器模式/227
4.12.1　解释器模式的结构/227
4.12.2　解释器模式的实现/228
4.13　调停者模式/233
4.13.1　为什么需要调停者/233
4.13.2　调停者模式的结构/235
4.13.3　调停者模式的实现/235
4.13.4　调停者模式的优点与缺点/237

参考文献/239

第1章　软件设计原则与UML简介

软件设计的重要性表现在软件的质量上。软件设计描述了软件是如何被分解和集成为组件的，同时也描述了组件之间的接口以及组件之间是如何发挥软件构建功能的。如何设计才能保证软件的质量？这里给出软件设计的一般原则：

1）有分层的组织结构，便于对软件中的各个构件进行控制。

2）应形成具有独立功能特征的模块（模块化）。

3）应有性质不同、可区分的数据和过程描述（表达式）。

4）应使模块之间、模块与外部环境之间接口的复杂性尽量减小。

5）应利用软件需求分析中得到的信息和可重复的方法。

要想得到一个满意的设计结果，不仅要有基本设计原则的指导，而且还要有系统化的设计方法和科学严格的评审机制相结合，才能达到预期目的。

软件设计原则从宏观上指导着软件设计，但软件设计的具体实现还要遵循软件设计的基本准则。本章介绍7种常用的软件设计原则以及读懂本书中的UML图所必须掌握的知识。

对于每一个程序员来说，要想编写出满足要求的程序，首先应了解软件设计的设计原则。怎样评价设计的软件是合格的？通常有如下要求：

1）可靠性。软件可靠性指该软件在测试运行过程中避免可能发生故障的能力，且一旦发生故障后，解脱和排除故障的能力。

2）健壮性。健壮性又称为鲁棒性，是指软件对于规范要求以外的输入，能够判断此输入不符合规范要求，并有合理的处理方式。软件健壮性是一个比较模糊的概念，但是却是非常重要的软件外部量度标准。

3）可修改性。要求以科学的方法设计软件，使其有良好的结构和完备的文档，系统性能易于调整。

4）可理解性。软件的可理解性是其可靠性和可修改性的前提。它不仅是文档清晰可读的问题，更要求软件本身具有简单明了的结构。

5）程序简便。

6）可测试性。可测试性就是设计一个适当的数据集合，用来测试所建立的系统，并保证系统得到全面的检验。

7）效率性。软件的效率性一般用程序的执行时间和所占用的内存容量来度量。在达到原理要求功能指标的前提下，程序运行所需时间越短、占用存储容量越小，则效率越高。

8）标准化原则。在结构上实现开放，基于业界开放式标准，符合国家和信息产业部的规范。

9）先进性。满足客户需求，系统性能可靠，易于维护。

10）可扩展性。软件设计完要留有升级接口和升级空间。

满足了软件开发过程中常用的软件设计原则，就容易实现上述诸多要求。本书选择了7种常用的软件设计原则，在下面章节中逐一介绍。

1.1 “开—闭”原则

“开—闭”原则（Open-Closed Principle，OCP）是指一个软件实体应当对扩展开放，对修改关闭。这个原则说的是在设计模块时，应当使这个模块可以在不被修改的前提下进行扩展。换言之，应当可以在不修改源代码的情况下改变这个模块的行为，且在保持系统一定稳定性的基础上，对系统进行扩展。这是面向对象设计（OOD）的基石，也是最重要的原则。

所有的软件系统都有一个共同的性质，即对它们的需求会随时间的推移而发生变化。在软件系统面临新的需求时，系统的设计必须是稳定的。满足“开—闭”原则的设计可以给一个软件系统两个无可比拟的优势：

1）通过扩展已有的软件系统，可以提供新的行为，以满足对软件的新需求，使变化中的软件系统具有一定的适应性和灵活性。

2）已有的软件模块，特别是最重要的抽象层模块不能再修改，这就使变化中的软件系统具有一定的稳定性和延续性。

具有这两个优点的软件系统是一个在高层次上实现了复用的系统，也是一个易于维护的系统。

【示例】玉帝招安美猴王

当年大闹天宫的孙悟空是对玉帝天庭的新挑战。孙悟空说：“皇帝轮流做，明年到我家。只叫他搬出去，将天宫让与我。”对于这项挑战，太白金星给玉皇大帝提出的建议是：“……降一道招安圣旨，把他宣来上界……与他籍名在箓……一则不动众劳师，二则收仙有道也。”

换言之，不劳师动众、不破坏天规便是“闭”，收仙有道便是“开”。招安之法便是玉帝天庭的“开—闭”原则，通过给孙悟空封一个“弼马温”的官职，便使现有系统满足了变化的需求，而不必更改天庭的既有秩序，结构如图1-1-1所示。

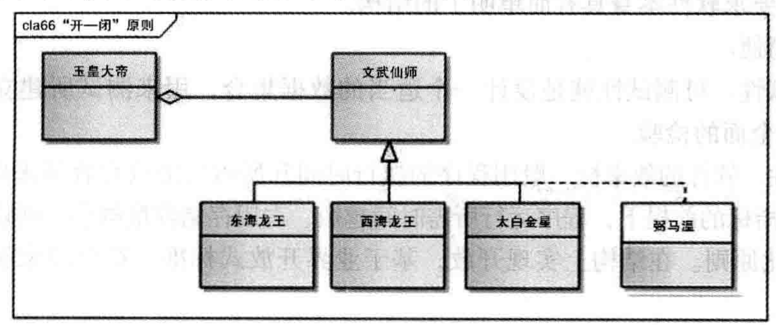

图1-1-1　玉帝招安美猴王

招安之法的关键是不允许更改现有的天庭秩序，但允许将孙悟空纳入现有秩序中，从而扩展了这一秩序。用面向对象的语言来讲，不允许更改的是系统的抽象层，而允许扩展的是系统的实现层。

1.2 里氏代换原则

里氏代换原则（Liskov Substitution Principle，LSP）由 Barbar Liskov 提出，是继承复用的基石。

里氏代换原则的意思是：程序中如果使用一个父类，那么一定可以用其子类替换，而且它根本不能察觉出父类对象和子类对象的区别。只有子类可以替换父类，软件单位的功能才能不受影响，父类才能真正被复用，而子类也能在父类的基础上增加新功能。反之代换不成立，即父类不能代替子类。

【示例】乘马说

《墨子·小取》中写："白马，马也；乘白马，乘马也。骊马（黑马），马也；乘骊马，乘马也。"结构如图 1-2-1 所示。

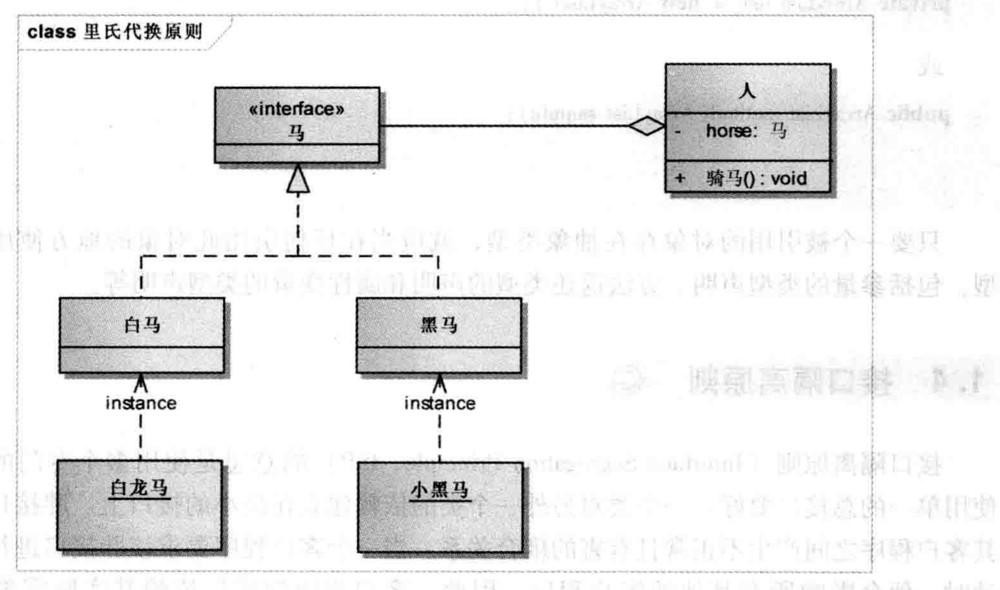

图 1-2-1　乘马说

对于人来说，骑小黑马和骑白龙马的结果是一样的，都是骑马，这就是里氏代换原则的一个实际应用。注意，应当尽量从抽象类继承，而不从具体类继承。一般而言，如果有两个具体类 A 和 B，它们有继承关系，那么一个最简单的修改方案是建立一个抽象类 C，然后让类 A 和类 B 成为抽象类 C 的子类，即如果有一个由继承关系形成的登记结构，则在等级结构的树形图中，所有的树叶节点都应是具体类，所有的树枝节点都应是抽象类或接口。

1.3 依赖倒置原则

依赖倒置原则（Dependence Inversion Principle，DIP）要求客户端依赖于抽象耦合。

依赖倒置原则表述一：抽象不应依赖于细节，细节应依赖于抽象。

依赖倒置原则表述二：要针对接口编程，不要针对实现编程。意思就是说，应当使用接口和抽象类进行变量的类型声明、参量的类型声明和方法的返还类型声明等。例如：

```
private List list = new ArrayList();
```

或

```
public List method(List sample){
    …
}
```

不要针对实现编程的意思是指不应当使用具体类进行变量的类型声明、参量类型声明和方法的返还类型声明等。例如：

```
private ArrayList list = new ArrayList();
```

或

```
public ArrayList method(ArrayList sample){
    …
}
```

只要一个被引用的对象存在抽象类型，就应当在任何引用此对象的地方使用抽象类型，包括参量的类型声明、方法返还类型的声明和属性变量的类型声明等。

1.4 接口隔离原则

接口隔离原则（Interface Segregation Principle，ISP）的意思是使用多个专门的接口比使用单一的总接口要好。一个类对另外一个类的依赖建立在最小的接口上。胖接口会导致其客户程序之间产生不正常且有害的耦合关系。当一个客户程序要求该胖接口进行一个改动时，便会影响所有其他的客户程序。因此，客户程序应该仅依赖其实际需要调用的方法。

接口隔离原则的使用场合：提供调用者需要的方法，屏蔽不需要的方法。

【示例】电子商务系统的订单类

例如，电子商务系统有一个订单类，有 3 个地方会使用此类：第一是门户，只能有查询方法；第二是外部系统，只有添加和查询订单方法；第三是管理后台，添加、删除、修改和查询都要用到。

根据接口隔离原则，一个类对另外一个类的依赖性应建立在最小的接口上，即对于门

户，它只能依赖有一个查询方法的接口。该示例系统的结构如图1-4-1所示。

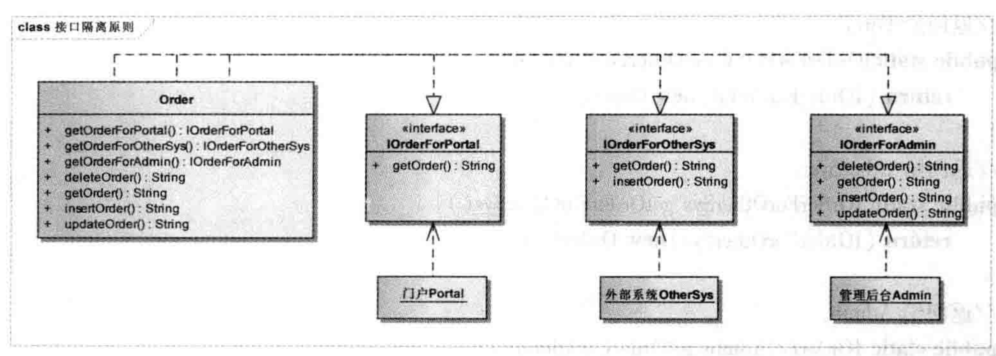

图1-4-1 电子商务系统的订单类

门户接口代码如下：

```
package com.接口隔离原则;
interface IOrderForPortal{
    String getOrder();
}
```

外部系统接口代码如下：

```
package com.接口隔离原则;
public interface IOrderForOtherSys {
    String getOrder();
    String updateOrder();
        String insertOrder();
}
```

管理后台接口代码如下：

```
package com.接口隔离原则;
interface IOrderForAdmin{
    String deleteOrder();
    String updateOrder();
    String insertOrder();
    String getOrder();
}
```

实现接口的订单类（Order）代码如下：

```
package com.接口隔离原则;
public class Order implements IOrderForPortal, IOrderForOtherSys, IOrderForAdmin{
    private Order(){
        //防止客户端直接new,然后访问它不需要的方法
    }
```

试读结束：需要全本请在线购买： www.ertongbook.com

```
//返回给 Portal
public static IOrderForPortal getOrderForPortal() {
    return (IOrderForPortal) new Order();
}
//返回给 OtherSys
public static IOrderForOtherSys getOrderForOtherSys() {
    return (IOrderForOtherSys) new Order();
}
//返回给 Admin
public static IOrderForAdmin getOrderForAdmin() {
    return (IOrderForAdmin) new Order();
}
//下面是接口方法的实现,这里只返回了一个 String,用于演示
public String getOrder() {
    return "getOrder 方法";
}
public String insertOrder() {
    return "insertOrder 方法";
}
public String updateOrder() {
    return "updateOrder 方法";
}
public String deleteOrder() {
    return "deleteOrder 方法";
}
}
```

测试代码如下:

```
package com.接口隔离原则;
public class Client {
    public static void main(String[] args) {
        IOrderForPortal orderForPortal = Order.getOrderForPortal();
        IOrderForOtherSys orderForOtherSys = Order.getOrderForOtherSys();
        IOrderForAdmin orderForAdmin = Order.getOrderForAdmin();
        System.out.println("门户可调用:\t" + orderForPortal.getOrder());
        System.out.println("外部系统可调用:\t" + orderForOtherSys.getOrder() + ";" +
orderForOtherSys.insertOrder() + ";" + orderForOtherSys.updateOrder());
        System.out.println("管理后台可调用:\t" + orderForAdmin.getOrder() + ";" +
orderForAdmin.insertOrder() + ";" + orderForAdmin.updateOrder() + ";" + orderForAdmin.
deleteOrder());
    }
}
```

运行结果如图 1-4-2 所示。

```
<terminated> Client (31) [Java Application] C:\Java\Genuitec\Common\binary\com.sun.java.jdk.win32.x86_1.6.0.013\
门户可调用：   getOrder方法
外部系统可调用：   getOrder方法;insertOrder方法;updateOrder方法
管理后台可调用：   getOrder方法;insertOrder方法;updateOrder方法;deleteOrder方法
```

图 1-4-2　运行结果

这样就能很好地满足接口隔离原则，调用者只能访问其自己的方法，不能访问不该访问的方法。

1.5　合成/聚合复用原则

合成/聚合复用原则（Composite/Aggregate Reuse Principle，CARP）的意思是要尽量使用合成/聚合，尽量不使用继承。合成/聚合和继承是实现复用的两个基本途径，合成复用原则是指尽量使用合成/聚合，而不使用继承。

合成和聚合都是对象建模中关联（Association）关系的一种。聚合表示整体与部分的关系，表示“含有”，整体由部分组合而成，部分可以脱离整体作为一个独立的个体存在。合成则是一种更强的聚合，由部分组成整体，且不可分割，部分不能脱离整体而单独存在。在合成关系中，部分和整体的生命周期一样，合成的新对象完全支配其组成部分，包括其创建和销毁。在一个合成关系中，成分对象不能与另外一个合成关系共享。

只有当下述条件全部满足时，才应使用继承关系：

1）子类是超类的一个特殊种类，而不是超类的一个角色，即区分“Has-A”和“Is-A”。只有“Is-A”关系才符合继承关系，“Has-A”关系应使用聚合来描述。

2）永远不会出现需要将子类换成另外一个类的子类的情况。如果不能肯定将来不会变成另外一个子类，则不要使用继承。

3）子类具有扩展超类的责任，而不是具有置换或注销超类的责任。如果一个子类有大量地置换超类的行为，那么这个类就不应该是这个超类的子类。

错误地使用继承而不是合成/聚合的一个常见原因是错误地把“Has-A”当成了“Is-A”。“Is-A”代表一个类是另一个类的一种；而“Has-A”代表一个类是另一个类的一个角色，而不是另一个类的特殊种类。

【示例】不使用合成/聚合复用原则的错误例子

例如，在图 1-5-1 中，一个人只能有一个角色，不能兼职。一个人无法同时拥有多个角色，是“雇员”就不能再是“学生”了，这个逻辑显然是不合理的。实际上，雇员、经理和学生描述的只是一种角色，如一个人是“经理”则必然是“雇员”，而另外一个人可能是“学生雇员”。

错误源于把“角色”的等级结构与“人”的等级结构相混淆，误把“Has-A”当作

"Is-A"。修正后的结构如图 1-5-2 所示。

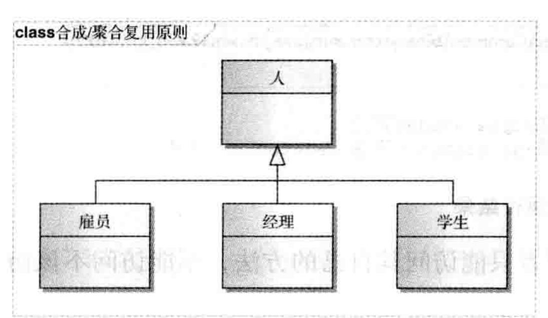

图 1-5-1　不使用合成/聚合复用原则的示例

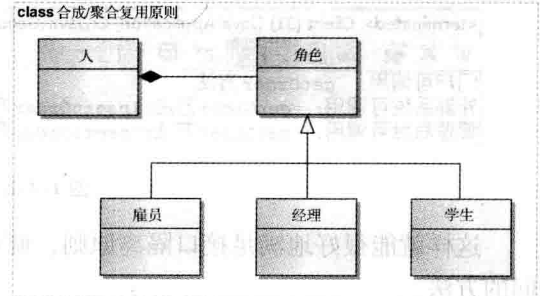

图 1-5-2　使用合成/聚合复用原则后的结构

现在编程就灵活多了，把角色作为人的属性，同一个人可以很方便地具有不同的角色。

1.6　迪米特法则

我们都知道软件结构设计的一个原则是"低耦合，高内聚"。无论是面向过程编程还是面向对象编程，只有使各个模块之间的耦合尽量低，才能提高代码的复用率。低耦合的优点不言而喻，但是怎样编程才能做到呢？这正是迪米特法则要完成的。

迪米特法则（Law of Demeter，LoD）又称为最少知识原则（Least Knowledge Principle，LKP），即一个对象应对其他对象有尽可能少的了解。该法则的核心观念就是类间解耦和弱耦合，只有弱耦合后，类的复用率才能提高。

没有任何一个其他的面向对象（OO）设计原则像迪米特法则这样有如此之多的表述方式，主要有以下几种：

1）只与你直接的朋友们通信（Only talk to your immediate friends）。

2）不要跟"陌生人"说话（Don't talk to strangers）。

3）每一个软件单位对其他的单位都只有最少的知识，而且仅局限于那些与本单位密切相关的软件单位。

也就是说，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，如果其中的一个类需要调用另一个类的某一个方法，则可以通过第三者转发这个调用。

【示例】不使用迪米特法则的错误例子

在安装软件时，我们经常有一个导向动作，第一步是确认是否安装，第二步是同意一些协议，然后再选择安装目录。这是一个典型的顺序执行动作，具体到程序中就是：调用一个或多个类，先执行第一个方法，然后执行第二个方法，根据返回的结果来判断是否调用第三个方法，其结构如图 1-6-1 所示。

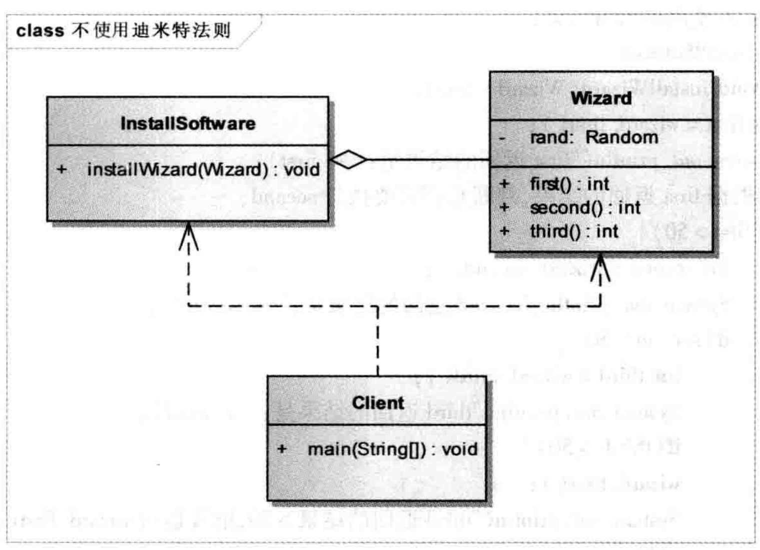

图 1-6-1　不使用迪米特法则的示例

这是一个很简单的结构图，实现软件安装的过程，其中 first 方法定义第一步做什么，second 方法定义第二步做什么，不使用迪米特法则的代码实现过程如下。

Wizard 类的代码如下：

```
package com. 不使用迪米特法则;
import java. util. Random;
public class Wizard {
    private Random rand = new Random(System. currentTimeMillis());
    //第一步
    public int first() {
        System. out. println("执行第一个方法...");
        return rand. nextInt(1010);
    }
    //第二步
    public int second() {
        System. out. println("执行第二个方法...");
        return rand. nextInt(100);
    }
    //第三步
    public int third() {
        System. out. println("执行第三个方法...");
        return rand. nextInt(100);
    }
}
```

InstallSoftware 类的代码如下：

```
package com.不使用迪米特法则;
public class InstallSoftware {
    public void installWizard( Wizard wizard) {
        int first = wizard.first( );
        System.out.println("first 返回的结果是:" + first);
        //根据 first 返回的结果,判断是否需要执行 second
        if( first > 50) {
            int second = wizard.second( );
            System.out.println("second 返回的结果是:" + second);
            if( second > 50) {
                int third = wizard.third( );
                System.out.println("third 返回的结果是:" + third);
                if( third  >50) {
                wizard.first( );
                System.out.println("third 返回的结果 > 50,继续运行 wizard.first( );" );
                } else
                  System.out.println("因 third 返回的结果 < 50,所以不再执行下一步操作。");
            } else
                System.out.println("因 third 返回的结果 < 50,所以不再执行下一步操作。");
        } else
            System.out.println("因 third 返回的结果 < 50,所以不再执行下一步操作。");
        }
}
```

测试代码如下:

```
package com.不使用迪米特法则;
public class Client {
    public static void main( String[ ] args) {
        Wizard wizard  =  new Wizard( );
        InstallSoftware IS  =  new InstallSoftware( );
        IS.installWizard( wizard);
    }
}
```

程序运行结果如图 1-6-2 所示。

```
Problems | @ Javadoc | Declaration | Co
<terminated> Client (32) [Java Application] C:\Java
执行第一个方法...
first返回的结果是: 319
执行第二个方法...
second返回的结果是: 57
执行第三个方法...
third返回的结果是: 46
因third返回的结果<50,所以不再执行下一
步操作。
```

图 1-6-2　程序运行结果

程序虽然简单，但其中隐藏的问题可不简单。Wizard 类把太多的方法暴露给了 InstallSoftware 类，两者的朋友关系太紧密了，耦合关系变得异常牢固。如果要将 Wizard 类中的 first 方法的返回值的类型由 int 改为 boolean，则需要修改 InstallSoftware 类，从而增加了修改变更的风险。因此，这样的耦合是极度不合适的，需要重新对设计进行重构，重构后的结构如图 1-6-3 所示。

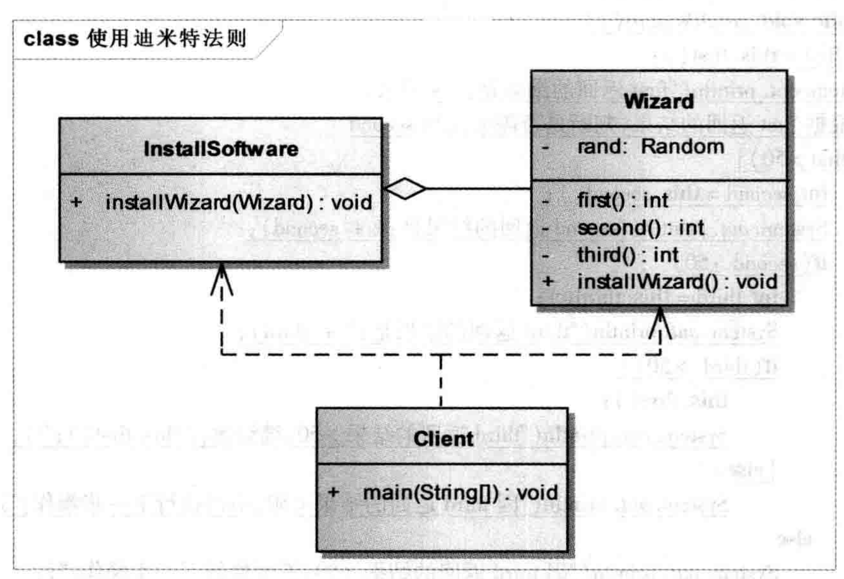

图 1-6-3　使用迪米特法则后的结构

注意，修改后使用迪米特法则的 Wizard 类的代码中的 private 方法与前面的不同了，并且增加了软件安装方法 public void installWizard()。

代码中的下画线部分显示了与前面代码的区别：

```
package com.迪米特法则;
import java.util.Random;
public class Wizard {
    private Random rand = new Random(System.currentTimeMillis());
    //第一步
    private int first() {
        System.out.println("执行第一个方法...");
        return rand.nextInt(1010);
    }
    //第二步
    private int second() {
        System.out.println("执行第二个方法...");
        return rand.nextInt(100);
    }
```

试读结束：需要全本请在线购买： www.ertongbook.com