杨　帆　王钧玉　孙更新　编著

设计模式从入门到精通

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

设计模式从入门到精通

杨　帆　王钧玉　孙更新　编著

電子工業出版社
Publishing House of Electronics Industry
北京・BEIJING

内 容 简 介

本书使用Java语言来描述经典的GoF的23种设计模式，在讲解过程中涉及了JDK 6.0中的新特性，全书采用案例驱动的形式，以一个完整的超市系统案例统领了全部知识点。本书以案例项目工程为主线，以应用为目的，循序渐进地讲解了设计模式的具体应用方法，易学易用，并且结合案例驱动形式，可以使读者将各种设计模式真正运用到实际开发中，避免理论与实践脱节的问题。

本书适用于对设计模式不甚了解的初学者，同时也适合具有一定编程基础、需要提高实践技术的程序员作为参考用书。本书还可作为高等院校计算机等专业及相关培训学校的指导教材。

图书在版编目（CIP）数据

设计模式从入门到精通/杨帆，王钧玉，孙更新编著.—北京：电子工业出版社，2010.8
ISBN 978-7-121-11560-8

Ⅰ. ①设… Ⅱ. ①杨…②王…③孙… Ⅲ. ①Java语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字（2010）第154726号

责任编辑：李红玉
文字编辑：易 昆
印 刷：北京天竺颖华印刷厂
装 订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
北京市海淀区万寿路173信箱 邮编：100036
北京市海淀区翠微东里甲2号 邮编：100036
开 本：787×1092 1/16 印张：33.25 字数：851.2千字
印 次：2010年8月第1次印刷
定 价：62.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

自GoF推出《设计模式》这本经典书籍以来，在软件设计界，学习使用模式的风潮就没有停止过，设计模式是前人对于软件设计经验的总结，非常具有学习的价值。

本书的作者具有丰富的实际开发经验和培训经验，鉴于多年的培训工作和软件开发经历，他们能很好地把握初学者对于模式的学习需求。本书是设计模式的实用性入门和进阶书籍，内容安排上注重实用，可以使初学者迅速学以致用。本书非常适合熟悉Java编程但是对设计模式经验相对较少的读者阅读。

本书采用案例驱动的形式，用一套完整的超市系统统领全书。书中第1章～第3章介绍了面向对象的设计方法以及设计模式的起源和优点，讲解了UML的发展历史以及常见的关系图。

第4章～第8章详细讲解了创建型模式的各个模式，包括了超市中各种需要创建的对象，解决了商品上架、捆绑销售、连锁店加盟、总店地位等问题。

第9章～第15章详细讲解了结构型模式的各个模式，解决的问题主要有：电源插座问题、游戏机配件问题、手机展示问题、宣传海报设计问题等。在案例中还穿插使用了创建型模式以加深读者对两大类型的模式的理解。

第16章～第26章详细讲解了行为型模式的各个模式，解决的主要问题有：连锁店客服专线问题、超市促销宣传资料的问题、商品进货审批中的问题、设计灵活高效的收银程序问题、超市在不同时段的运营状态问题、商品打折问题、设计不同糕点的制作流程问题等。在示例中综合运用了创建型模式、结构型模式、行为型模式，可以使读者对不同模式之间的合作加深印象。

第27章对所有模式进行了总结并提出了对未来模式的学习方向。

总体来说，本书具有以下特点：

- 内容全面，实例丰富，讲解循序渐进。
- 以基础知识紧密结合典型实例，具有很强的操作性和实用性。
- 从实际应用的角度出发，可以帮助读者以最快的速度学会使用Java语言开发设计模式，全面提高程序员的开发技术水平和面向对象的设计能力。

• 采用案例驱动模式，不仅能使读者掌握好知识，更能使其在实际开发中学以致用。

• 整体案例中贯穿了全部章节的知识，读者可以以此更深刻地理解模式的相互合作编程法。

本书在编写过程中，得到了很多专家和同事的大力支持，在此一并表示感谢。同时由于作者水平有限，加上时间仓促，书中难免存在疏漏和不妥之处，欢迎广大读者批评指正。

为方便读者阅读，若需要本书配套资料，请登录“北京美迪亚电子信息有限公司”（http://www.medias.com.cn），在“资料下载”页面进行下载。

目　录

第1章　设计模式初见

设计模式（Design Pattern）是一套经过分类的、被反复使用的软件代码设计经验的总结。使用设计模式是为了可复用代码，让代码更容易被理解，保证代码的可靠性。通常来说，设计模式是软件复用的基础理论，它使代码编制真正工程化。

设计模式最初是在建筑学中被提出的，建筑师克里斯托佛·亚历山大在1970年代编撰了一本汇集设计模式的书，但是设计模式的思想在建筑设计领域里的影响远没有后来在软件开发领域里传播得广泛和深远。

软件设计中的设计模式是在GoF（“四人帮”，指Gamma、Helm、Johnson & Vlissides、Addison-Wesley四人）合著的《设计模式》一书中第一次提出的，随后被规范化。本书提出的23种基本设计模式便属于《设计模式》中所提及的经典的模式。

1.1　一切从某个小超市开始

在软件工程领域中研究一种具体的技术，通常都会借助一个具体的案例来分析和学习，在本书中也不例外，在每一章节的学习过程中，读者除了要学习和分析模式的案例，还将学习如何使用设计模式来解决一个现实工程中存在的问题。

各章节的案例都来源于一个“超市”案例，因为“超市”对于大众来说都比较熟悉，其中发生的问题也比较容易理解。

软件设计中的超市是什么样子的呢？其实本软件设计中就是使用软件来模拟人经营一个超市，现实生活中的超市中发生的各种情况在软件环境中都要提及并加以处理。比如要进行商品的上架、仓库进货、打折销售、客户服务、广告宣传等。

在学习每一个模式时，为了达到良好的学习效果，读者最好能了解一下每一章涉及的超市问题发生的原因及需求，这对理解模式的意图是十分关键的。

1.2　为何使用设计模式

要回答为何要使用设计模式这个问题，必须要知道设计模式的优点。设计模式的优点如下：

复用解决方案

在代码设计中通常会遇到需要设计的方案和以前设计的某个方案类似的问题，比如之前已经设计过一个论坛系统，现在又要设计一个讨论版系统。这时，比较好的解决方案就是最大限度地利用之前设计的代码。

设计模式的主要思想就是“复用”，通过复用已经确认的设计，能够在解决问题的过程中使用最小的成本获得最大的效益，而且可以在学习他人经验的过程中获利，不用再为那些总是会重复出现的问题重复设计解决方案。

设计模式将设计方法标准化

开发团队中的交流和协作通常都要使用共同的词汇和要对问题达成共识。设计模式在项目

的分析和设计阶段提供了共同的基准点。如果项目团队中的所有成员都熟悉设计模式就可以产生很高的沟通效率，比如团队成员A说，我认为这个地方适合使用“策略模式”来实现算法改变。团队成员B说，我不同意你的观点，我认为这个地方的算法改变是跟对象状态相关的，应该使用“状态模式”。这样表述起来非常简明扼要。

设计模式可以提高个人和团队的设计能力

在开发团队中使用设计模式的经验证明，设计模式既可以帮助开发人员个人提高个人水平，也可以帮助团队提高整体实力。这是因为，经验少的团队成员亲眼看到已经掌握设计模式的资深开发人员如何快速从中获益后，他们会更加自发、主动地去学习这些强大的知识。

设计模式使软件更容易修改和维护

设计模式能够使软件更容易修改和维护的原因在于：这些模式都是久经考验的解决方案。所以，它们的结构都是长期发展形成的，比新构思的解决方案更善于应对变化。而且，这些设计模式所用的代码往往更易于理解，从而使代码更易于维护。使用设计模式对开发人员、维护人员、测试人员都有好处，所以使用设计模式是一种“多赢”的设计方法。

如果说学习某种具体的编程语言是练习程序设计的“招式”的话，那么学习设计模式无疑就是练习程序设计的“内功”，而“内功”通常才是一个高手制胜的关键。

1.3 设计模式分类

经典的GoF的23种设计模式分为三个类别。

- 创建型模式：用于创建对象。对象的创建会消耗掉系统的很多资源，所以单独对对象的创建进行研究，以便能够高效地创建对象就是创建型模式要探讨的问题。
- 结构型模式：用于构建类间的关系。如何设计对象的结构、继承和依赖关系会影响到后续程序的维护性，代码的健壮性、耦合性等。这些因素需要使用结构型模式来优化。
- 行为型模式：用于控制对象的行为。如果对象的行为设计得好，那么对象的行为就会更清晰，它们之间的协作效率就会更高。

每一类设计模式都有多种具体模式，如表1-1所示。

表1-1 设计模式分类

模式分类	模式名称
创建型模式	工厂方法模式
	抽象工厂模式
	建造者模式
	原型模式
	单例模式
结构型模式	适配器模式
	桥接模式
	代理模式
	外观模式
	装饰模式
	组合模式
	享元模式

（续表）

模式分类	模式名称
行为型模式	命令模式
	观察者模式
	责任链模式
	迭代器模式
	访问者模式
	状态模式
	备忘录模式
	策略模式
	调停者模式
	模板方法模式
	解释器模式

1.4 阅读建议与学习资源

下面推荐的是公认的设计模式经典书籍：

《设计模式：可复用面向对象软件的基础》（*Design Patterns: Elements of Reusable Object-Oriented Software*）；

《设计模式精解》（*Design Patterns Explained*）；

《重构——改善既有代码的设计》（*Refactoring: Improving the Design of Existing Code*）；

《软件复用技术：在系统开发过程中考虑复用》（*Software Reuse Techniques Adding Reuse to the Systems Development Process*）；

《软件复用：结构、过程和组织》（*Software Reuse Architecture, Process and Organization for Business Success*）；

《企业应用架构模式》（*Patterns of Enterprise Application Architecture*）；

《UML面向对象设计基础》（*Fundamentals of Object-Oriented Design in UML*）；

《敏捷软件开发——原则、模式与实践》（*Agile Software Development: Principles, Patterns, and Practices*）；

《软件工程——实践者的研究方法（原书第5版）》（*Software Engineering: A Practitioner's Approach, Fifth Edition*）；

《计算机程序设计艺术》（*The Art of Computer Programming*）。

学习设计模式的经典网站资源：

维基百科：http://en.wikipedia.org或http://zh.wikipedia.org。

UML软件工程组织：http://www.uml.org.cn/。

J道：http://www.jdon.com/。

CSDN综合社区：http://www.csdn.net。

WeLie模式频道：http://www.welie.com/patterns/。

第2章　面向对象设计原则

毫无疑问，当前最流行的软件设计方式就是面向对象的设计方式。在程序设计的过程中，使用面向对象的设计方式会给大规模的软件开发带来清新的活力，提高编程的效率，但是如果对面向对象的设计方法掌握不好也会带来很多的问题，本章就介绍一下面向对象的设计原则，这也是设计模式需要遵守的原则。

2.1　软件的维护代价

软件设计通常来说具有长期性，设计出软件之后还需要根据需求的变更对其进行维护。维护通常分为几类：功能扩充、功能修改、功能删除。设计不当通常会带来如下问题：

- 僵化性（Rigidity）：设计难以改变。具有这种特性的软件将无法维护。
- 脆弱性（Fragility）：设计易于遭到破坏。这种特性使得其他程序员在使用设计好的类时有可能将其破坏。
- 牢固性（Immobility）：设计难以重用。无法重用的软件模块将会导致大量不必要的劳动。
- 不必要的复杂性（Needless Complexity）：过分设计导致设计复杂，影响开发效率和性能。
- 不必要的重复（Needless Repetition）：过多的重复违反了不要重复自己的DRY（don't repeat yourself）原则。
- 晦涩性（Opacity）：混乱的表达导致程序不可读。

如果设计的应用程序具备以上问题，很显然，维护的成本将是很高的，但是如果采用正确的面向对象设计原则，将会解决这些问题。

2.2　面向对象设计原则

良好的面向对象设计方法需要遵循一些基本原则，如单一职责原则（SRP）、开—闭原则（OCP）、里氏替换原则（LSP）、依赖倒置原则（DIP）、接口隔离原则（ISP）等。下面依次进行举例介绍。

2.2.1　基础原则：“开—闭”原则（OCP）

“开—闭”原则的含义是：一个软件实体应当对扩展开放，对修改关闭。

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。从另外一个角度讲，就是所谓的“对可变性封装原则”。“对可变性封装原则”意味着两点：

- 一种可变性不应当分散于很多代码片段中，而应当被封装到一个对象里面。同一种可变性的不同表象意味着同一个继承等级结构中的具体子类。
- 一种可变性不应当与另一种可变性混合在一起。类的设计应该具备特定的可变性而不是众多的可变性。

下面的例子将说明此原则的意义，假设要用代码描述一个汽车类Car，其中要定义汽车的名称和发动机类型，代码如下：

代码片段1 Car.java

```
1  package cn.steven.oo.ocp;
2
3  /**
4   * 违反开—闭原则的汽车类
5   */
6  public class Car {
7
8      /**
9       * 汽车名称
10      */
11     private String name;
12
13     /**
14      * 汽车引擎
15      */
16     private String eng;
17
18     public String getName() {
19         return name;
20     }
21
22     public void setName(String name) {
23         this.name = name;
24     }
25
26     public String getEng() {
27         return eng;
28     }
29
30     public void setEng(String eng) {
31         this.eng = eng;
32     }
33
34     public void print() {
35         System.out.println("引擎是" + eng + "的" + name);
36     }
37
38     public static void main(String[] args) {
39         Car car = new Car();
40         car.setEng("V4发动机");
41         car.setName("皮卡");
42         car.print();
43         //运行结果
44         //引擎是V4发动机的皮卡
45     }
```

```
46
47  }
```

可见，虽然这个类的使用是没有问题的，但是要更换其中的发动机类型将变得很复杂（上面例子中使用的是字符串，效果不明显，假设发动机是一个单独的类这个问题就明显了），所以发动机类型的这种变化应该被独立设定出来。

修改后的代码如下：

代码片段2　CarModi.java

```
1   package cn.steven.oo.ocp;
2
3   /**
4    * 符合开—闭原则的汽车类
5    */
6   public class CarModi {
7
8       /**
9        * 汽车名称
10       */
11      private String name;
12
13      /**
14       * 汽车引擎
15       */
16      private IEng eng;
17
18      public String getName() {
19          return name;
20      }
21
22      public void setName(String name) {
23          this.name = name;
24      }
25
26      public IEng getEng() {
27          return eng;
28      }
29
30      public void setEng(IEng eng) {
31          this.eng = eng;
32      }
33
34      public void print() {
35          System.out.println("引擎是" + eng.getEng() + "的"
36                  + name);
37      }
38
39      public static void main(String[] args) {
40          CarModi car = new CarModi();
41          car.setEng(new V4Eng());
```

```
42              car.setName("皮卡");
43              car.print();
44              car.setEng(new V8Eng());
45              car.print();
46              // 运行结果
47              // 引擎是V4的皮卡
48              // 引擎是V8的皮卡
49          }
50
51  }
52
53  /**
54   * 引擎接口
55   */
56  interface IEng {
57      public String getEng();
58  }
59
60  /**
61   * V4引擎
62   */
63  class V4Eng implements IEng {
64      private String eng = "V4";
65
66      public String getEng() {
67          return eng;
68      }
69  }
70
71  /**
72   * V4引擎
73   */
74  class V8Eng implements IEng {
75      private String eng = "V8";
76
77      public String getEng() {
78          return eng;
79      }
80  }
```

由代码可见，现在已经可以自由地更换引擎而不用修改汽车对象本身了。

2.2.2 单一职责原则（SRP）

单一职责原则的含义是：就一个类而言，应该仅有一个引起它变化的原因。

在构造对象时，应将对象的不同职责分离至多个类中，从而确保引起该类变化的原因只有一个。使用此原则可以提高内聚，降低耦合度。

比如有一个文档类，这个类目前的职责是负责维护文档内容和打印，代码如下：

代码片段3 Document.java

```
1  package cn.steven.oo.srp;
```

```
2
3  /**
4   * 违反SRP的文档类
5   */
6  public class Document {
7
8      /**
9       * 文档内容
10      */
11     private String content;
12
13     public String getContent() {
14         return content;
15     }
16
17     public void setContent(String content) {
18         this.content = content;
19     }
20
21     /**
22      * 打印方法
23      */
24     public void printMethod() {
25         System.out.println("将文档内容打印至控制台");
26     }
27 }
```

由代码可见，Document类有两种使其改变的因素，一个是文档内容的改变，这个是此类的本分职责；另一个是，如果现在需要将文档传送至文件或者网络进行打印，那么势必要修改Document类中的print方法。这样一来，Document类就有了两种使其改变的因素，违反了SRP原则。修改方法就是将print方法放在另一个职责中：

代码片段4　DocumentModi.java

```
1  package cn.steven.oo.srp;
2
3  /**
4   * 符合单一职责原则的文档类
5   */
6  public class DocumentModi {
7
8      /**
9       * 文档内容
10      */
11     private String content;
12
13     public String getContent() {
14         return content;
15     }
16
17     public void setContent(String content) {
```