

大话设计模式

◎ 感悟百态人生
◎ 深谙模式思想
◎ 愉悦、风趣中感受对象思维
◎ 轻松、诙谐中顿悟模式思想



程序的关键不是程序本身，而是程序所体现的设计模式理念。

吴强 / 编著



在金融危机下股票还挣钱

——外观模式

蜡笔与毛笔

——桥接模式

放风者与偷窃者

——观察者模式

中介公司

——中介者模式

高老庄的故事

——代理模式

麦当劳的鸡腿

——抽象工厂模式

月光宝盒

——备忘录模式

多功能的手机

——扩展型模式

三明治

——装饰器模式

企业管理出版社
ENTERPRISE MANAGEMENT PUBLISHING HOUSE

版权信息

书名：大话设计模式

作者：吴强

出版社：企业管理出版社

ISBN：9787802555372

排版：安然

制作：豆子书坊

本书由北京斯坦威图书有限责任公司策划出版并全球制作发行中文电子版

版权所有·侵权必究



本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质
电子书下载！！！！

前言

现代科学技术迅猛发展，计算机信息技术发挥着巨大的作用，并已经渗透到各行各业，推动着这些行业的迅速发展。很多读者朋友熟练地掌握了语言，却对设计模式不够重视，导致在实际的工作中没有真正应用到起关键作用的设计模式。

而了解了设计模式，在宏观上就能把握面向对象编程的精髓。对于大多数不懂编程的朋友来说，了解了设计模式，也就是体会到了编程世界的一个大的框架。

究竟什么是设计模式呢？

设计模式就是由某些需要严密整合的具体接口开始，最后过渡到一种通用的结构。不管最后所选取的设计模式是什么，最初的目的都是相同的，就是为了解决一个设计问题。它所创造的一系列词汇可以帮助我们同其他开发者相互交流。

设计模式体现的是一种思想，思想是指导行为的一切。理解和掌握设计模式，记住23种或者更多的设计场景和解决策略是不够的，更要接受一种思想的熏陶和洗礼。用这种思想进行设计和开发，这才是重要的。

本书通过故事讲述程序如何设计。希望能给渴望了解面向对象程序设计的初学者及困惑、无法复用的代码编程体验者一些好的建议和提示。

本书主要采用Java语言介绍设计模式中比较常见的23种设计模式，分29章具体

介绍，以现实生活中常见的事情为例来具体分析讲解。在本书中，以“男人和女人通过媒人约会”为例来说明。这样比只告诉概念性的内容更加容易理解和记忆。为了让读者能够更好地理解这23种常见的设计模式，本书还举了许多的例子，如我们大学生毕业面临的问题：大学毕业了怎么办？参加招聘会或是大学生毕业后会选择什么样的路？等等。每种设计模式都以一个现实生活中的故事为例，引入该模式的概念，目的是使概念能够通俗易懂，然后是举一至两个较简单易懂的代码例子来具体体现该模式。之所以会采用Java语言是因为Java比C++计算机语言简单，没有像C++语言中有一些不容易理解或容易出错的概念和语法。Java是一种较新的计算机语言，所以它在面向对象和多线程特性上比其他现有计算机语言显得更纯粹一些，在网络平台无关性和安全性方面的优点也比大部分计算机语言更显突出。而Java语言本身是一种可以满足这种需求的计算机语言。学习Java程序设计，应用Java语言实现算法也比较容易，从而节省编程时间，编写出来的Java代码比较容易得到复用和移植。

本书是一些基础性的内容，不适合有多年面向对象开发经验和对常用的设计模式了如指掌的人。它所面向的读者是那些想提高的初中级Java程序员。

■ 本书人物及背景

小A：原名李华，22岁，广东人，广州某大学计算机专业大学三年级学生，成绩一般，但是好学上进。

大B：原名黄大远，29岁，广东人，广州某大学毕业，是小A的师兄。毕业后长期从事软件开发和管理工作，住在小A家附近，小A以向大B学习为由，经常找大B聊天。大B也很欣赏小A的好学上进，所以也常常鼓励小A，帮小A解决学习上遇到的困

难。

■ 本书结构

本书主要分为七个部分来讲述23种常见设计模式。

第一部分主要是第一章设计模式的概述。

第二部分是（第2章~第6章）介绍接口型模式。主要包括：适配器模式、外观模式、组合模式、桥接模式。

第三部分是（第7章~第12章）介绍责任型模式。主要包括：单体模式、观察者模式、中介者模式、代理模式、享元模式。

第四部分是（第13章~第18章）介绍构造型模式。主要包括生成器模式、工厂方法模式、抽象工厂模式、原型模式、备忘录模式。

第五部分是（第19章~第24章）介绍操作型模式。主要包括模板方法模式、状态模式、策略模式、命令模式、解释器模式。

第六部分是（第25~第29章）介绍扩展型模式。主要包括装饰器模式、迭代器模式、访问者模式和设计模式总结。

第七部分是附录。

编 者

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质

电子书下载！！！！

第一部分

设计模式概述

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第一章 大学毕业了怎么办？——设计模式

概述

1.1 大学毕业了怎么办

大B和小A是同一所大学的师兄弟，都是学计算机编程专业。大B在C大毕业在从事软件开发工作，大B是小A的校友兼师兄，大B在大学四年学了不少软件开发方面的东西，也学着编了些小程序，小A经常会找师兄学习一些关于编程方面的问题。

时间：12月2日20点 地点：大B房间 人物：大B，小A

这天，小A问大B，大学毕业了怎么办？

小A：“常听人说：‘大学毕业=失业’！”

大B：“不尽然吧！事实上还是有好多同学挺希望毕业的，有的人觉得在学校里学不到什么东西，或因为希望自己早点独立可以减轻家里负担啊什么的。”

小A：“毕业了，就是成人了。应该对自己负责了，又觉得还不能独自去面对社会。”

大B：“这就是为什么我们大学毕业后会觉得痛苦，觉得自己没有就业能力吧！

不敢面对社会。有能力的人到哪里都不愁找不到好工作，相反欠缺工作经验的年轻人，如果没有一个正确的职业规划、良好的求职动机、成熟的求职技巧，可能到哪都会遇到不少困难和挫折。”

小A：“我平时在学校都很努力学习啊，毕业后找工作真的很难吗？”

大B：“就现在社会形式而言，找到工作其实并不难的，难的是找到自己喜欢的工作。现在的年轻人，大多都是喜欢‘钱多、活少、离家近、坐坐办公室’的工作，但是在现实中是不太可能的。我个人建议，先找一份适合自己发展，能累积到很多实践经验的基层工作，有了工作经验，再找更理想的工作，或是在原有岗位往更高的岗位发展就不再那么难了。”

小A：“那么师兄，你说如何才能找到适合自己的工作呢？”

大B：“我认为呐：首先还是要了解下自己的综合实力，再就是要密切留意下社会上的岗位需求，总得说起来有三点：1、我想做什么？ 2、我能做什么？ 3、市场要什么？”

1.2 什么是设计模式

这天大B问小A：“怎样设计可复用的面向对象软件？”

小A：“靠，师兄你这是考我么？”

大B：“啥啊？我这是想看你在学校是不是真学到了东西。”

小A：“得得得，那我就说说吧。设计模式是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。”

大B：“我再考考你，用C++、Java、C#或VB.NET任意一种面向对象语言实现一个简单程序。”

小A不到几分钟就给大B一个程序。

```
/*
 * @(#)Blah.java          1.82 99/03/18
 * Copyright (c) 1994-1999 Sun Microsystems, Inc.
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information"). You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 */
package java.blah;
import java.blah.blahdy.BlahBlah;
/**
 * Class description goes here.
 *
 * @version 1.82 18 Mar 1999
 * @author  Firstname Lastname
```

```

*/
public class Blah extends SomeClass {
/* A class implementation comment can go here. */
*
* classVar2 documentation comment that happens to be
* more than one line long
*/

private static Object classVar2;
/** instanceVar1 documentation comment */
public Object instanceVar1;
/** instanceVar2 documentation comment */
protected int instanceVar2;
/** instanceVar3 documentation comment */
private Object[] instanceVar3;
/**

public class Blah extends SomeClass {
/* A class implementation comment can go here. */
*
* classVar2 documentation comment that happens to be
* more than one line long
*/

private static Object classVar2;
/** instanceVar1 documentation comment */
public Object instanceVar1;
/** instanceVar2 documentation comment */
protected int instanceVar2;
/** instanceVar3 documentation comment */
private Object[] instanceVar3;
/**

* ...constructor Blah documentation comment...
*/

public Blah() {

```

```
// ...implementation goes here...
}
/**
 * ...method doSomething documentation comment...
 */
public void doSomething() {
// ...implementation goes here...
}
/**
 * ...method doSomethingElse documentation comment...
 * @param someParam description
 */
public void doSomethingElse(Object someParam) {
// ...implementation goes here...
}
}
```

1.3 代码规范

大B：“呵呵呵，写得不错，不亏是C大的高材生。”

小A：“师兄你这不是在取笑我嘛。我还有好多问题得请教你哩！”

大B：“行行行.....没问题。小师弟好学，师兄哪能袖手旁观的呐！对了，你写代码的时候一定要注要代码的规范。”

小A：“代码规范？”

大B：“嗯，来来来，我说给你听。首先是要注意注释文档的格式，注释文档将用来生成HTML格式的代码报告，所以注释文档必须书写在类、域、构造函数、方法、定义之前。注释文档由两部分组成——描述、块标记。”

例如：

注释

```
@author LEI
@version 1.10 2005-09-01
public void doGet (HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    doPost(request, response);
}
```

大B：“看！前两行为描述，描述完毕后，由@符号起头为块标记。”

大B：“然后是注释的种类。有文件头注释、构造函数注释、域注释方法注释、和定义注释。”

小A：“什么？注释还有那么多的种类的？我以前都没有用心去记过它的喔。师兄能不能给我讲讲每一种注释的功能和要求啊？”

大B：“这有什么问题哩！举些例子来讲，那样你就容易理解多了。”

小A：“嘿嘿！那感情好。”

大B：“文件头注释已结束，需要注明该文件创建时间，文件名，命名空间信息。”

例如：

描述部分用来书写该类的作用或者相关信息，块标记部分必须注明作者和版本。

如：

```
class Window extends BaseWindow {  
  
...  
  
}
```

大B：“构造函数注释采用，描述部分注明构造函数的作用，不一定有块标记部分。域注释可以出现在注释文档里面，也可以不出现在注释文档里面。用的域注释将会被认为是注释文档出现在最终生成的HTML报告里面，而使用的注释会被忽略。这个应该注意！”

例如：

```
boolean isTrigerSuccess = false;
```

又例如：

```
boolean isTrigerSuccess = false;
```

再例如：

```
int x = 1263732;
```

大B：“还有就是方法注释，方法注释采用描述部分注明方法的功能，块标记部分注明方法的参数，返回值，异常等信息。定义注释，规则同域注释。”

小A：“喔.....原注释还真的是有那么多种类，每个种类都有不同的功能哩！嘿嘿！看来以后我得更认识学习才行哩！”

大B：“呵呵.....你能这么想就好喽。对了，小A，注释块标记你知道不？”

小A：“注释块标？嘿嘿，这个.....也不知道.....”

大B：“没关系，我给告诉你。首先是标记的顺序。”

块标记将采用如下顺序：

```
...
*
* @param (classes, interfaces, methods and constructors only)
* @return (methods only)
* @exception (@throws is a synonym added in Javadoc 1.2)
* @author (classes and interfaces only, required)
* @version (classes and interfaces only, required. See footnote 1)
* @see
* @since
* @serial (or @serialField or @serialData)
* @deprecated (see How and When To Deprecate APIs)
* ...
```

一个块标记可以根据需要重复出现多次，多次出现的标记按照如下顺序：

@author 按照时间先后顺序 (chronological)
@param 按照参数定义顺序 (declaration)
@throws 按照异常名字的字母顺序 (alphabetically)
@see 按照如下顺序：
@see #field
@see #Constructor(Type, Type...)
@see #Constructor(Type id, Type id...)
@see #method(Type, Type,...)
@see #method(Type id, Type, id...)
@see Class
@see Class#field
@see Class#Constructor(Type, Type...)
@see Class#Constructor(Type id, Type id)
@see Class#method(Type, Type,...)
@see Class#method(Type id, Type id,...)
@see package.Class
@see package.Class#field
@see package.Class#Constructor(Type, Type...)
@see package.Class#Constructor(Type id, Type id)
@see package.Class#method(Type, Type,...)
@see package.Class#method(Type id, Type, id)
@see package

小A：“哇噻！块标记还可以按照时间先后顺序，按照参数定义顺序，按照异常名字的字母顺序哩！师兄，你讲得真详细。看来我得好好得花点心思把它们都记下来才好哩！”

大B：“能记住当然好哩，我再给你讲下标记介绍。”

@param标记

@param后面空格后跟着参数的变量名字（不是类型），空格后跟着对该参数的描述。

在描述中第一个名字为该变量的数据类型，表示数据类型的名次前面可以有一个冠词如：a, an, the。如果是int类型的参数则不需要注明数据类型。例如：

```
...
* @param ch the char 用来.....
* @param _image the image 用来.....
* @param _num 一个数字.....
...
```

大B：“对于参数的描述如果只是一短语，最好不要首字母大写，结尾也不要句号。对于参数的描述是一个句子，最好不要首字母大写，如果出现了句号这说明你的描述不止一句话。如果非要首字母大写的话，必须用句号来结束句子。（英文的句号）”

公司内部添加ByRef和ByVal两个标记，例如：

```
* @param _image the image ByRef 用来.....
```

说明该参数是引用传递（指针），ByVal可以省略，表示是值传递。

大B：“代码规范大概要点就是这些了。听起来好像很多，只要用心，其实也很容易记的。”

小A：“嘿嘿！师兄，你太厉害了！”

1.4 初学代码者常犯的错误

小A：“师兄，我经常在写代码的时候犯错，你能不能教教我呐！”

大B：“好啊！当出现这些错误的时候，要仔细的看看错误提示，找出问题所在，避免以后不再发生同样的错误。在这个过程当中我们的经验和水平也在不断的提升。Java错误，主要包括两方面，一种是语法错误，另一种是逻辑错误。语法错误，也就是我们的编码不符合Java 规范，在编译的时候无法通过。通常，我们都是用javac 编译我们的源程序，如果代码中存在语法错误，比如某个表达式后缺少分号的时候，编译器就会告诉我们错误信息，编译就此停止。逻辑错误，也就是我们常说的Bug，一般存在逻辑错误的程序都是可以顺利的被编译器编译产生相应的字节码文件，也就是class文件。但是，在执行的时候，也就是java ourClass的时候，得出的结果并不是我们所希望的。”

小A：“对！我在写代码的时候就是经常出现这样的问题，经常都解决不了。”

大B：“失败是成功的他妈！学习编程是没有什么捷径可以走的，要在不断的学习和编码的过程中，逐渐的积累经验，从开始的模仿者变成最后的创作者。和学习其它的编程语言一样，在开始编码的时候也会出现很多很多的错误，而且有的错误可能也是不断的出现。”

小A：“嗯嗯嗯.....师兄说得是！”

1.5 面向对象编程

小A：“师兄，用任意一种面向对象语言实现，就是要用面向对象的编程方法去实现，对吗？”

大B：“一般编程初学者都会遇到这样的问题，碰到问题就直觉地用计算机能够理解的逻辑来描述和表达待解决的问题及具体的求解过程。其实这是用计算机的方式去考虑它，就好比计算器这个程序，先输入两个数和运算符号，再根据运算符号判断选择如何运算，得出结果。这样是对的。但这样的想法却使得程序只为满足实现当前的需求，而程序就不容易维护，不容易扩展，也更不容易复用。也就达不到高质量代码的要求了。”

小A：“师兄，你这样一讲我又不懂了，那怎么程序才能容易维护，容易扩展，也容易复用哩？”

大B：“我再跟你讲细点吧！顺便也举些例子，理解一点。发广告邮件，广告邮件列表存在数据库里面。倘若用C来写的话，一般会这样思考，先把邮件内容读入，然后连接数据库，循环取邮件地址，调用本机的qmail的sendmail命令发送。然后考虑用Java来实现，既然是OOP，就不能什么代码都塞到main过程里面，于是就设计了三个类：一个类是负责读取数据库，取邮件地址，调用qmail的sendmail命令发送；一个类是读邮件内容，MIME编码成HTML格式的，再加上邮件头；一个主类负责从命令读参数，处理命令行参数，调用发email的类。把一件工作按照功能划分为3个模块分别处理，每个类完成一件模块任务。仔细的分析一下，你就会发现这样的设计完全是从程序员实现程序功能的角度来设计的，或者说，设计类的时候，是自底向上的，从机器的角度到现实世界的角度来分析问题的。因此在设计的时候，就已经把程序编程实现的细节都考虑进去了，企图从底层实现程序这样的出发点来达到满足现实世界的软件需求的目标。这样的分析方法其实是不适用于Java这样面向

对象的编程语言。”

小A：“为什么？”

大B：“因为，如果改用C语言，封装两个C函数，都会比Java实现起来轻松的多，逻辑上也清楚的多。”

小A：“我倒觉得面向对象的精髓在于考虑问题的思路是从现实世界的人类思维习惯出发的，只要领会了这一点，就领会了面向对象的思维方法。”

大B：“这样吧，我再举一个非常简单的例子：假使现在需要写一个网页计数器，客户访问一次页面，网页计数器加1，计数器是这样来访问的
如：`http://hostname/count.cgi?id=xxx` 后台有一个数据库表，保存每个id（一个id对应一个被统计访问次数的页面）的计数器当前值，请求页面一次，对应id的计数器的字段加1（这里我们忽略并发更新数据库表，出现的表锁定的问题）。”

大B：“如果按照一般从程序实现的角度来分析，我们会这样考虑：首先是从HTTP GET请求取到id，然后按照id查数据库表，获得某id对应的访问计数值，然后加1，更新数据库，最后向页面显示访问计数。

小A：“现在假设一个没有程序设计经验的人，要怎样来思考这个问题的呢？会提出什么样的需求呢？”

大B：“你很可能会这样想：我需要有一个计数器，这个计数器应该有这样的功能，刷新一次页面，访问量就会加1，另外最好还有一个计数器清0的功能，当然计数器如果有一个可以设为任意值的功能的话，我就可以作弊了。做为一个没有程序设计经验的人来说，他完全不会想到对数据库应该如何操作，对于HTTP变量该如何

传递，他考虑问题的角度就是有什么需求，我的业务逻辑是什么，软件应该有什么功能。”

按照这样的思路需要有一个计数器类 Counter，有一个必须的和两个可选的方法：

getCount() // 取计数器值方法

resetCounter() // 计数器清0方法

setCount() // 设计数为相应的值方法

把Counter类完整的定义如下：

```
public class Counter {  
    public int getCount(int id) {}  
    public void resetCounter(int id) {}  
    public void setCount(int id, int currentCount) {}  
}
```

解决问题的框架已经有了，来看一下如何使用Counter。在count.cgi里面调用Counter来计数，程序片段如下：

```
// 这里从HTTP环境里面取id值  
...  
Counter myCounter = new Counter(); // 获得计数器  
int currentCount = myCounter.getCount(id); // 从计数器中取计数  
// 这里向客户浏览器输出  
...  

```

程序的框架全都写好了，剩下的就是实现Counter类方法里面具体的代码了，此时才去考虑具体的程序语言实现的细节。

面向对象的思维方法其实就是在现实生活中习惯的思维方式，是从人类考虑问题的角度出发，把人类解决问题的思维方式逐步翻译成程序能够理解的思维方式的过程，在这个翻译的过程中，软件也就逐步被设计好了。

大B：“在运用面向对象的思维方法进行软件设计的过程中，最容易犯的错误就是开始分析的时候，就想到了程序代码实现的细节，因此封装的类完全是基于程序实现逻辑，而不是基于解决问题的业务逻辑。”

1.6 面向对象

大B “我给你个通俗的例子：面向对象OO = 面向对象的分析OOA + 面向对象的设计OOD + 面向对象的编程OOP；通俗的解释就是万物皆对象，把所有的事物都看作一个个可以独立的对象(单元)，它们可以自己完成自己的功能，而不是像C那样分成一个个函数；现在纯正的OO语言主要是Java和C#，C++也支持OO，C是面向过程的；你看这样好理解多了吧？”

小A：“是啊，这样的话一下子就看懂了！呵呵.....”

1.7 面向对象的特征

大B：“那小师弟，面向对象它都有些什么特征哩？”

小A：“面向对象的三个基本特征是：封装、继承、多态。”

大B：“嗯，是的。那你不能用图来说明？”

小A：“用图来说明？可以的。下面这个是我写的图（如图1-1面向对象基本特征所示），还得请师兄多指教。”

大B：“吼，不错嘛。”

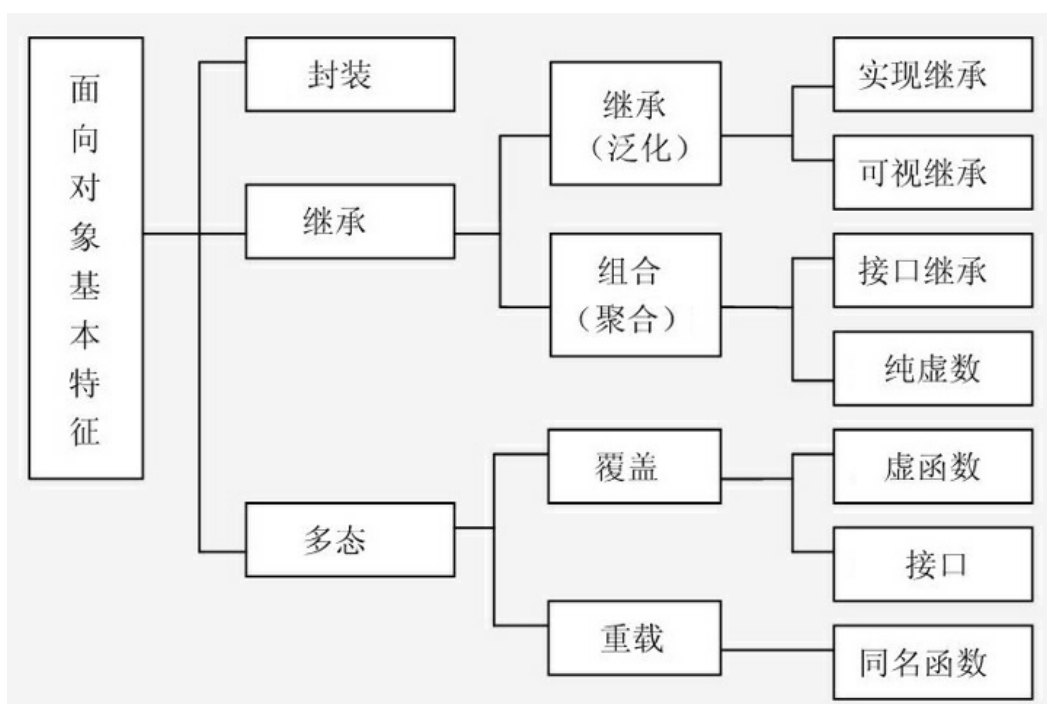


图1-1 面向对象基本特征

1.8 面向对象的优势

大B：“其实学习编程也没什么很难的，我一开始也不知道的，不过学做了软件开发几年后，遇多了，还有更改最初想法的事件，就慢慢明白了它的道理。”

小A：“呵呵.....这就是经验所得嘛！”

大B：“像你这么好学，一定能学好的！在我们生活中接触得最多是‘面向对象编程技术’，而‘面向对象编程技术’也是面向对象技术中的一个组成部分。面向对象技术需要面向对象的分析，设计和编程技术，也需要借助必要的建模和开发工具。”

小A：“师兄，能不能给我讲讲面向对象的优点具体有哪些呐？”

大B想，好学的小师弟，想想自己当年要是也能向小师弟这么好学的话，那肯定比现在学得好。

大B：“行呐，我讲给你听。1、要符合人们习惯的思维方法，便于分解大型的复杂多变的问题。由于对象对应于现实世界中的实体，因而可以很自然地按照现实世界中处理实体的方法来处理对象，软件开发可以很方便地与问题提出者进行沟通和交流。2、易于软件的维护和功能的增减。对象的封装性及对象之间的松散组合，都给软件的修改和维护带来了方便。3、可重用性好。重复使用一个类（类是对象的定义，对象是类的实例化），可以比较方便地构造出软件系统，加上继承的方式，极大地提高了软件开发的效率。4、与可视化技术相结合，改善了工作界面。随着基于图形界面操作系统的流行，面向对象的程序设计方法也将深入人心。它与可视化技术相结合，使人机界面进入GUI时代。”

小A：“就如Java语言，它都有哪些优点呐？”

大B：“Java是目前最流行的语言不是没有道理的。1、最为显著的优点是它与平台无关。Java依靠它的运行库(Run Time Library)获得了以往任何一种语言都没有的平台无关性。同样的代码可以不用改动就可在Windows、Solaris、Unix等各

种软硬件平台上运行。 2、另外一个显著的优点是Java的类C++语法。Java从C++发展而来，对于当今世界上众多的c++程序员来说，Java显得并不陌生。 3、面向对象。Java语言是完全面向对象的，区别于C++的“半面向对象”。目前面向对象技术已经取代早期的结构化程序设计方法而成为计算机界的标准技术，因为事实证明面向对象技术处理复杂问题的优势远非其他方法所能及。 4、健壮。Java自己操纵内存减少了内存出错的可能性。Java还实现了真数组，避免了覆盖数据的可能。这些功能特征大大缩短了开发Java应用程序的周期。Java提供Null指针检测数组边界检测异常出口字节代码校验。 5、安全。Java最重要的一点保证是：Java的安全体系架构。Java的安全性可从两个方面得到保证。一方面，在Java语言里，象指针和释放内存等C++功能被删除，避免了非法内存操作。另一方面，当Java用来创建浏览器时，语言功能和浏览器本身提供的功能结合起来，使它更安全。 6、多线程。简言之为一项任务多点开工，多线程带来的更大的好处是更好的交互性能和实时控制性能。在Java里，你可用一个单线程来调一副图片，而你可以访问HTML里的其它信息而不必等它。 7、动态。Java的动态特性是其面向对象设计方法的发展。它允许程序动态地装入运行过程中所需要的类，这是C++语言进行面向对象程序设计所无法实现的。”

小A：“哇噻！你不说我还真不知道Java语言还有这么多优点哩！嘿嘿！也真难怪现在最流行它了。”

大B：“是啊！不管哪种语言都有各自的优缺点，Java的缺点就是编译、执行的速度太慢，所以Java私塾建议你如果想学编程，不要总是问这个好不好，那个难不难，只要下定决心学就对了。”

1.9 类、对象、方法和实例变量

这天，大B问小A，“小师弟，你知道什么是类，对象，方法和实例变量吗？”

小A：“师兄，你问得我早都学过，不信，我说给你听。类是一种复杂的数据类型，它是将不同类型的数据和与这些数据相关的操作封闭在一起的集合体。类是对一组事物的抽象，是对事物的特性和功能的描述。类是一种模板，并不代表具体的事物。对象是类的实例，即类的变量。方法是指实现对象所具有的功能操作的代码。每个对象中一般包括若干种方法，每个方法有方法名和对应的一组代码。方法体现了对象的一种行为能力。实例变量.....实例变量.....”

大B：“哈哈！不记得了吧？”

小A：“实例变量？”

大B：“让师兄来告诉你吧，实例变量就是说某一实例具有的状态,比如说圆的半径,汽车的颜色。”

小A：“喔.....我明白了，嘿嘿，其实这个我学过的，只是.....只是一时想不起来了.....”

大B：“没事，我这次问你，你不知道，下次遇到，你不就想起来了。对于初学者来说要理解类、对象、和对象变量不是一件很容易的事。现以美眉为例来说明。假设你的学校或者是赚大米的地方有很多美眉，为了和这些妹妹中的一部分或者全部建立良好的关系，你需要建立一个Java类:Meimei。那么学校种的美眉们就是类Meimei，而对象就是类的一个实例，那么其中任何一个美眉就是对象。”

如：

```
meimei1("Jennifer", ...);  
meimei2("Lucy",...);  
meimei3("Danny",...);  
.....
```

假如你想让其中一个Meimei类实例成为你的‘超友谊朋友’，另外一个Meimei类实例成为你的‘女朋友’，那么‘超友谊朋友’和‘女朋友’就是一个Meimei类对象变量；‘超友谊朋友’和‘女朋友’这两个对象变量就引用其中一个Meimei对象。如现在你的超友谊朋友是meimei1，你的女朋友是meimei2,那么：

```
超友谊朋友 = meimei1;  
女朋友 = meimei2;  
meimei1.name = "Jennifer";  
meimei2.name = "Lucy";  
meimei3.name = "Danny"  
超友谊朋友.name = "Jennifer";  
女朋友.name = "Lucy";
```

三个月以后你的超友谊朋友是meimei3，你的女朋友是meimei1,那么：

```
超友谊朋友 = meimei3;  
女朋友 = meimei1;  
meimei1.name = "Jennifer";  
meimei2.name = "Lucy";  
meimei3.name = "Danny";
```

```
超友谊朋友.name = "Danny";
```

```
女朋友.name = "Jennifer";
```

大B问小A，：“这下你该对类，对象，方法又有新的认识了吧？”

小A：“嘿嘿！简直就是重新认识啊！哈哈！”

1.10 继承

大B：“那我再来问你，你来讲讲继承。”

小A：“继承是指一个对象直接使用另一对象的属性和方法。”

大B：“嗯！事实上，我们遇到的很多实体都有继承的含义。例如，若把汽车看成一个实体，它可以分成多个子实体，如：卡车、公共汽车等。这些子实体都具有汽车的特性，因此，汽车是它们的‘父亲’，而这些子实体则是汽车的‘孩子’。”

1.11 接口

大B：“你知道什么是接口吗？”

小A：“这个我知道，接口用来定义一种程序的协定。实现接口的类或者结构要与接口的定义严格一致。有了这个协定，就可以抛开编程语言的限制（理论上）。

接口可以从多个基接口继承，而类或结构可以实现多个接口。接口可以包含方法、属性、事件和索引器。接口本身不提供它所定义的成员的实现。接口只指定实现该接口的类或接口必须提供的成员。”

看到小师弟还学得挺好的嘛！大B越来越欣赏小师弟了。

大B说：“接口好比一种模版，这种模版定义了对象必须实现的方法，其目的就是让这些方法可以作为接口实例被引用。接口不能被实例化。类可以实现多个接口并且通过这些实现的接口被索引。接口变量只能索引实现该接口的类的实例。”

1.12 UML图

小A：“师兄，我想请你帮我总结和理解一下类图，因为我学了那么久的编程，类图就是学不好，简单的类图我还可以看懂，有些标记很容易混淆。你能给我讲讲吧！”

大B：“先看看UML的定义：统一建模语言（Unified Modeling Language，UML）是一种绘制软件蓝图的标准语言。”

小A：“那它有什么特性？”

大B：“顾名思义，它具备语言的特性：

- 标准性：元素、规则、机制
- 逻辑性：严谨

- 灵活性：同样的事情，不同的正确表述
- 方言性：利益驱动；翻译版本的混乱
- 不可盲目模仿性：避免片面借鉴，抓住事务本质和思想灵魂”

小A：“嘿嘿，标准性、逻辑性我能理解，什么是灵活性，方言性和不可盲目模仿性？”

大B：“方言性，在一方面是由于软件商家（如微软）追求商业利益、行业标准的制定权和话语权，造成了一些CASE工具未完全遵从UML标准这一混乱现象；另一方面，由于国内翻译的参考教材中文字晦涩难懂、不统一，造成目前的UML的学习门槛高、入门困难的局面。其实，真的，这东西没有这么高深。说到方言性，不得不补充一句，不建议使用VISIO做为UML的CASE工具，UML的三个爸爸早在94、95年分别加入Rational公司，没有理由不使用Rational Rose啊。最值得一提的是它的灵活性、不可盲目模仿性。举个例子吧！”

用例场景：

张无忌，出生于冰火岛，父亲张翠山，母亲殷素素。张无忌的武功大全：武当长拳、九阳神功、武当梯云纵、乾坤大挪移、少林擒龙手、崆峒七伤拳、太极拳剑、圣火令武功。

先画个用例图，描述张无忌他家的情况。如图1-2用例图所示

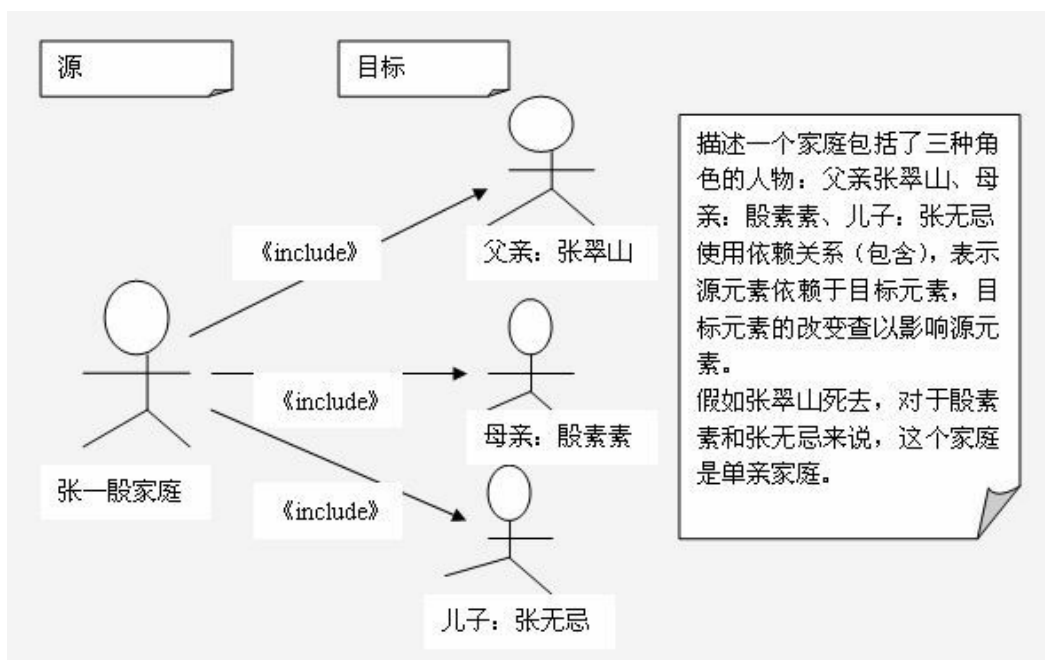


图1-2 用例图

2) 以泛化关系表示继承的时候，用例描述中需要强调描述：儿子继承了爸爸的什么，爸爸有哪些没有被儿子继承，儿子还新增了什么。如图1-3用例图所示

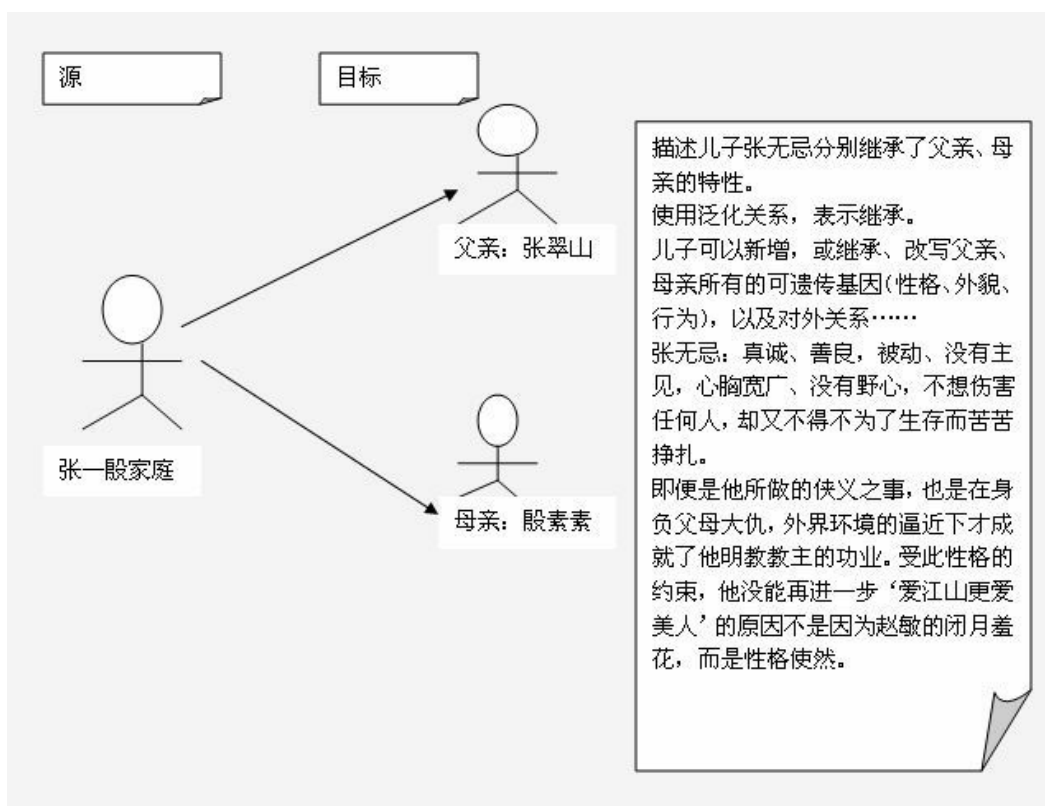


图1-3 用例图

3) 这是一个有问题的用例图，看看错误在哪里？提示：聚合关系是一种松散的关系，目标元素可有可无。如图1-4用例图所示

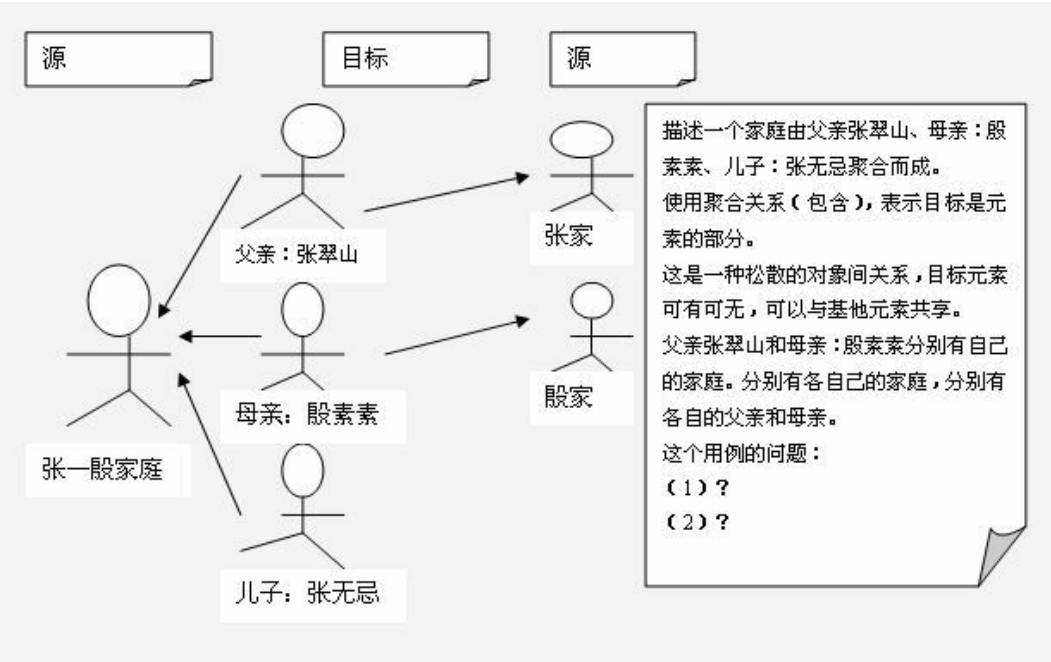


图1-4 用例图

想一想，人亡家还在么？所以以上关系用聚合是完全错误的！那么，上面用例场景描述中，哪些可以使用聚合关系描述呢？

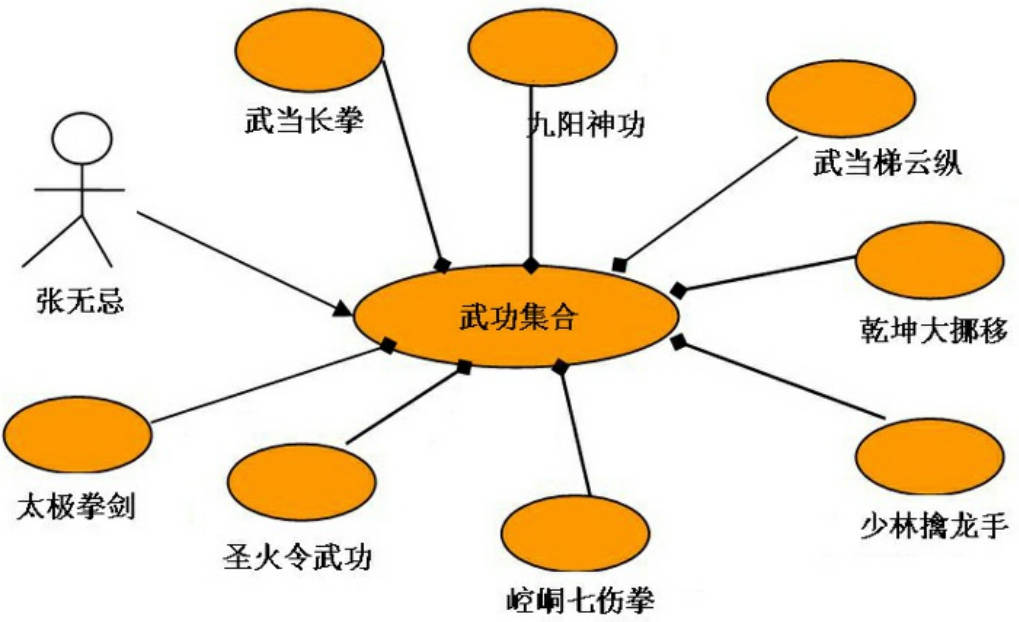


图1-5 用例图

没错，张无忌可以不会任何武功，也可以会N多武功；张无忌学习九阳神功，并不妨碍杨过、郭靖去学习嘛！如图1-5用例图所示

4) 这还是一个有问题的用例，看看错误在哪里？提示：组合关系是一种强关联，它有一个重要的特性：部分在某一时刻仅仅只能属于一个整体。如图1-6用例图所示

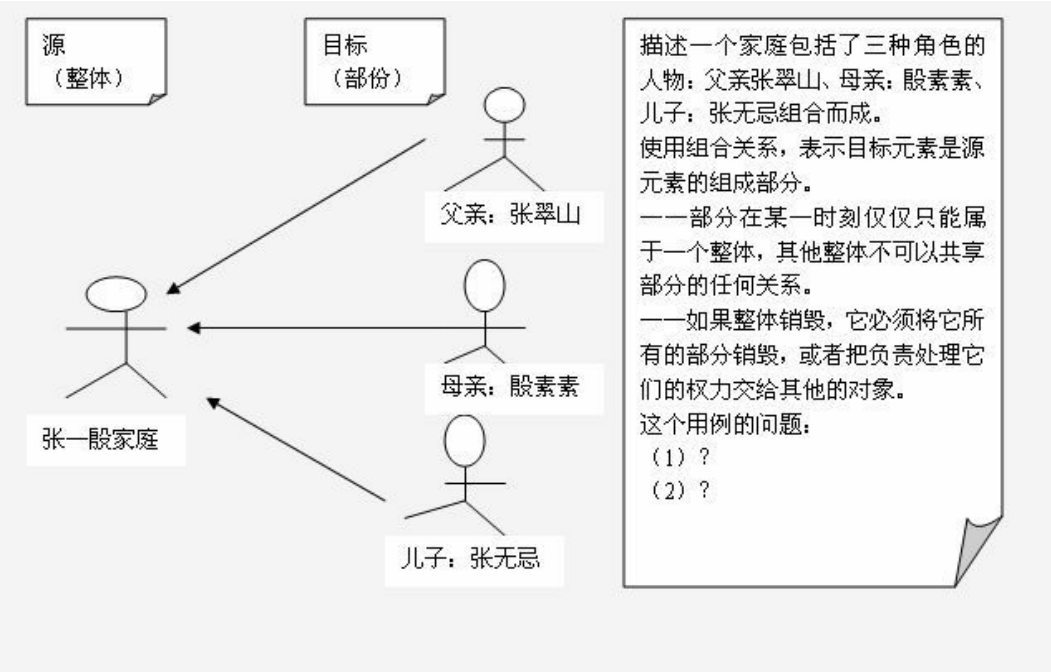


图1-6 用例图

想一想，殷素素不但是张翠山的老婆，还是殷家的女儿，张家的儿媳妇。有人问，这张用例图中并未讲到张家和殷家啊，所以这般描述好像也不错。非也，用例图描述的是实际的、真实的关系，跟“人为地”画或不画是没有任何关系的。所以上面关系用组合是完全错误的！那么，上面用例场景描述中，哪些可以使用组合关系描述呢？

我挖，我挖，我使劲挖，挖掘需求。5分钟过后。突然想到：冰火岛上有树，树上长叶子，嘿，有了！如图1-7用例图所示

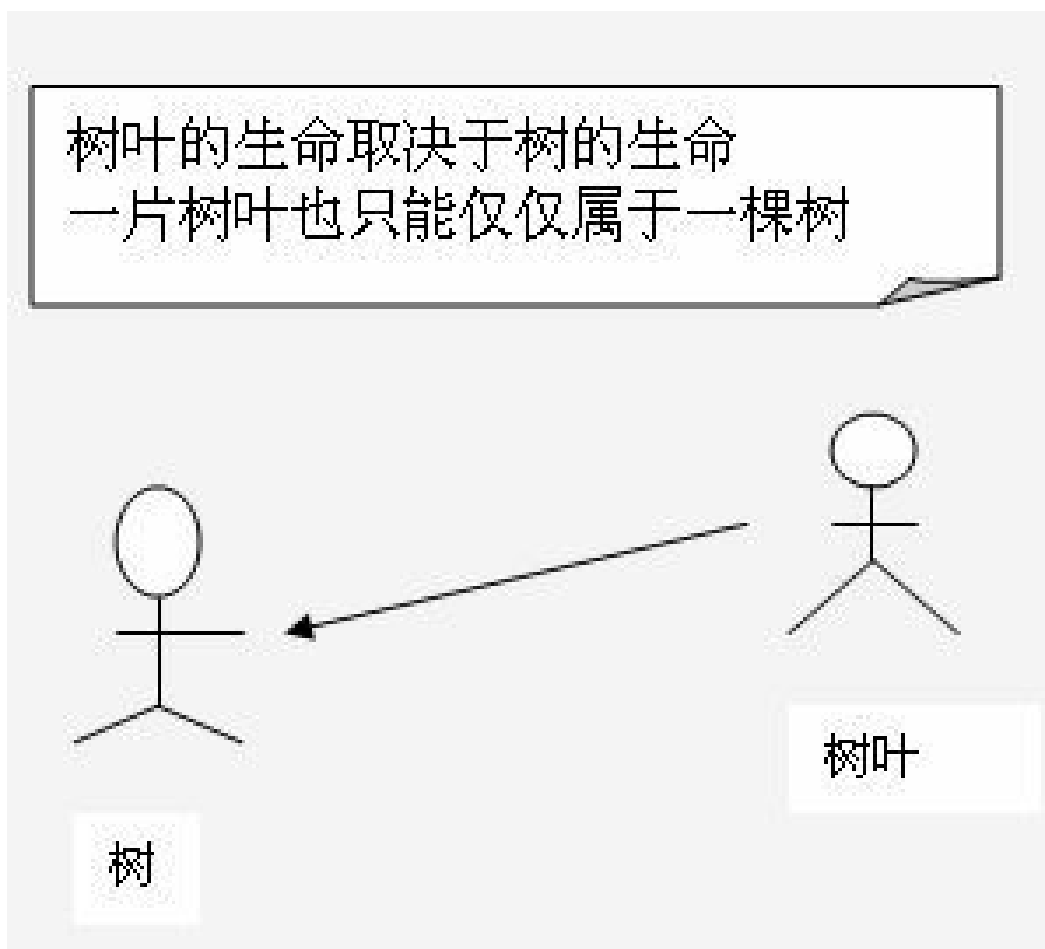
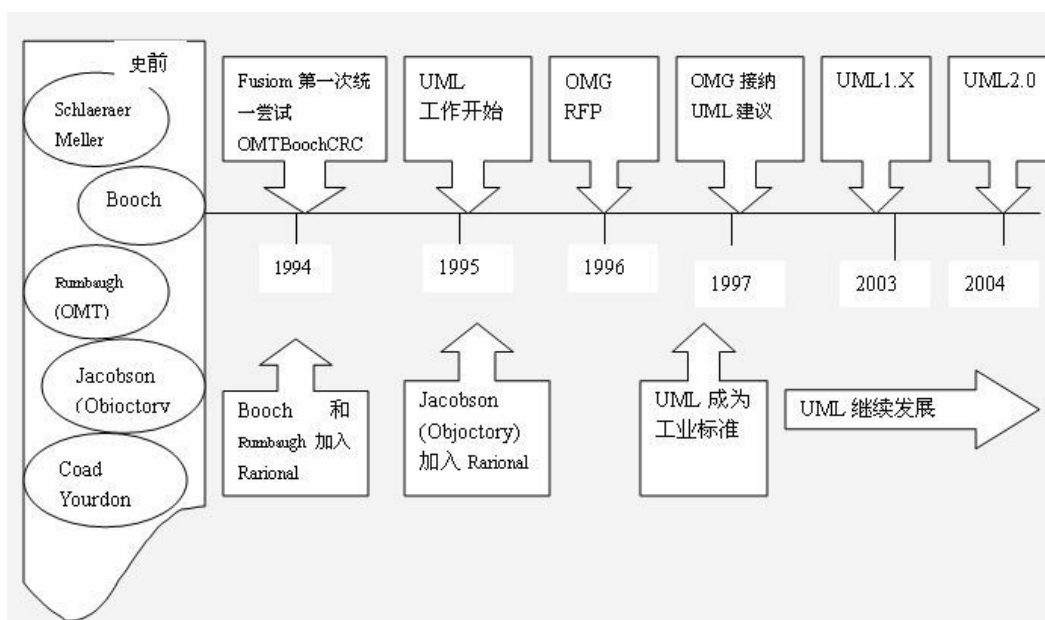


图1-7 用例图

现在是不是感觉UML很好玩？想不想了解一下UML的成长历程，想不想认识一下它的三个爸爸？如图1-8用例图所示



UML之父：Grady Booch、James Rumbaugh、Ivar Jacobson

Grady Booch 在他的一本书中说：“如果你有好的思想，那么它也是我们的”。这其实从一方面概括了UML的哲学——它吸取已有的精华并且在其上进行OOA/D（面向对象分析和设计）整合和构造。这是最广泛意义上的复用。

大B：“在正式上手去应用UML之前，再了解一下应用UML的三种方式，包括：

- UML作为草图

非正式的、不完整的图（通常是在白板上手绘草图），借助可视化语言的功能，用于探讨问题或解决方案空间的复杂部分。

- UML作为蓝图（主要方式）

这是UML更加正式和精确的用法，使用UML用于详细规定软件系统。UML模型可被维护，并成为软件的一个重要交付成果。用于：正向工程，逆向工程。这种方法需要使用如Rational Rose建模工具。

- UML作为编程语言

使用模型驱动构架（Model Driven Architecture，MDA），给UML模型添加足够的细节，使得能够从模型中编译生成系统。这是UML最正式和精确的用法，是软件开发的未来。但目前在理论、工具的健壮性和可用性方面仍处于发展阶段。”

大B：“还有就是，类图这东西你以后看多了，用多了自然就熟悉了。”

小A：“看来UML类图也不太难嘛！嘿嘿！”

大B：“就是啊！以后就要记住了哦！编程是一门技术，也是一门艺术。不能只满足于写代码运行结果正确，要考虑如何让代码更加简练，更加容易维护，容易扩展和复用，这样才可以真正得到提高。UML类图不是一学就会的，要有一个慢慢熟练的过程。学无止境，理解面向对象的才是真正学习编程的开始！”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第二部分

接口型模式

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第二章 学校招聘会——接口型模式介绍

2.1 学校招聘会

这个星期六我校会举办校园招聘会。面对如此严峻的就业形式，小A虽然还不是毕业班，但是他想去参加这个星期六的招聘会。他想见识和感受一下现场招聘的气氛，顺便了解一下情况，提前做好准备工作。

时间：12月13日8点 地点：校东区体育场

学校招聘会就设在我校东区体育场，今天小A早早地来到这里。招聘的单位早已把整个操场占的满满的，但来应聘的大学生还是把一个个单位围的水泻不通。

这里可是我们天天打球，上体育课的地方呐！这里有我们的快乐和欢笑。如今，这里却是处处严峻！

转了一上午，投了几份简历，虽然不是真的要来找工作的，但是今天的招聘会的确让小A见识到了不少。小A想毕业后也会找一个工作，也会参加这样的招聘会。觉得很不可思议，跟想像的差的太远了，还没准备好接受这些。但总有毕业的那天吧，总要面对这些的，生活真的很不容易.....

2.2 接口型模式

自从参加了学校招聘会，小A就更加努力地学习，他知道现在的社会，不但是要讲究学历，工作经验，更讲究的是个人的能力。

小A把参加了学校招聘会的事给大B讲了讲。

大B：“现实中学校操场的多功能，就如在系统的设计时刻常常遇到这样一个问题：类Client的实例instanceClient希望使用另一个对象instanceX提供的服务service，但在设计时，我们并不能确定对象instanceX究竟属于哪个类。”

小A：“那遇到这些情况的时候，我们应该怎么办呐？”

大B：“当遇到这些情况，常见的解决办法是：将对象instanceX提供的服务service抽象为一个接口ServiceProvider，然后让对象instanceClient通过持有接口ServiceProvider的实例来使用服务service。这种通过接口间接获得服务的解决方案就是接口模式。”

小A：“喔.....”

大B：“接口模式还可以有一些变化的形式：不止用一个接口抽象一个对象提供的服务，还可以用一组接口抽象一群对象的交互。”

2.3 接口型模式包括哪些模式

大B：“你知道接口型有哪些模式吗？”

小A：“接口型模式包括：适配器模式，外观模式，合成模式以及桥接模式等。”

大B：“对！就是这些.....要记住它喔！”

2.4 接口和抽象类的区别

小A：“在Java语言中，`abstract class` 和 `interface` 是支持抽象类定义的两机制。那抽象类和接口有什么区别啊？”

大B：“`abstract class`和`interface`之间在对于抽象类定义的支持方面具有很大的相似性，甚至可以相互替换，因此很多开发者在进行抽象类定义时对于`abstract class`和`interface`的选择显得比较随意。其实，两者之间还是有很大的区别的，对于它们的选择甚至反映出对于问题领域本质的理解、对于设计意图的理解是否正确、合理。”

大B：“那你先来理解一下抽象类。”

小A心想师兄怎么问我这么简单的问题。“`abstract class`和`interface`在Java语言中都是用来进行抽象类定义的。”

大B：“那么什么是抽象类，使用抽象类能为我们带来什么好处呢？”

小A：“好处？”

大B：“嗯哼，对，好处。说说看？”

小A：“在面向对象的概念中，我们知道所有的对象都是通过类来描绘的，但是反过来却不是这样。并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。”

大B：“嗯。不错。抽象类往往用来表征我们在对问题领域进行分析、设计中得出的抽象概念，是对一系列看上去不同，但是本质上相同的具体概念的抽象。比如：如果我们进行一个图形编辑软件的开发，就会发现问题领域存在着圆、三角形这样一些具体概念，它们是不同的，但是它们又都属于形状这样一个概念，形状这个概念在问题领域是不存在的，它就是一个抽象概念。正是因为抽象的概念在问题领域没有对应的具体概念，所以用以表征抽象概念的抽象类是不能够实例化的。在面向对象领域，抽象类主要用来进行类型隐藏。”

小A：“师兄，你这样一讲，我就不是很懂了。”

大B：“没关系，我跟你举个例子先，在语法层面，Java语言对于abstract class和interface给出了不同的定义方式，下面以定义一个名为Demo的抽象类为例来说明这种不同。使用abstract class的方式定义Demo抽象类的方式如下：

```
abstract class Demo {  
    abstract void method1();  
    abstract void method2();  
    ...  
}
```

使用interface的方式定义Demo抽象类的方式如下：

```
interface Demo{  
    void method1();  
    void method2();  
    ...  
}
```

大B：“你要知道，在abstract class方式中，Demo可以有自己的数据成员，也可以有非abstract的成员方法，而在interface方式的实现中，Demo只能够有静态的不能被修改的数据成员（也就是必须是static final 的，不过在interface中一般不定义数据成员），所有的成员方法都是abstract的。从某种意义上说，interface是一种特殊形式的 abstract class。从编程的角度来看，abstract class和interface都可以用来实现 ‘design by contract’ 的思想。但是在具体的使用上面还是有一些区别的。”

小A：“那它都有哪些区别呢？”这回小A可是不敢再把抽象型模式当是个简单的问题了。

大B：“首先，abstract class 在 Java 语言中表示的是一种继承关系，一个类只能使用一次继承关系(因为Java不支持多继承 -- 转注)。但是，一个类却可以实现多个interface。也许，这是Java语言的设计者在考虑Java对于多重继承的支持方面的一种折中考虑吧。其次，在abstract class的定义中，我们可以赋予方法的默认行为。但是在interface的定义中，方法却不能拥有默认行为，为了绕过这个限制，必须使用委托，但是这会增加一些复杂性，有时会造成很大的麻烦。在抽象类中不能定义默认行为还存在另一个比较严重的问题，那就是可能会造成维护上的麻烦。因为如果后来想修改类的界面（一般通过 abstract class 或者

interface来表示)以适应新的情况(比如,添加新的方法或者给已用的方法中添加新的参数)时,就会非常的麻烦,可能要花费很多的时间(对于派生类很多的情况,尤为如此)。但是如果界面是通过abstract class来实现的,那么可能就只需要修改定义在abstract class中的默认行为就可以了。同样,如果不能在抽象类中定义默认行为,就会导致同样的方法实现出现在该抽象类的每一个派生类中,违反了‘one rule, one place’原则,造成代码重复,同样不利于以后的维护。因此,在abstract class和interface间进行选择时要非常的小心。”

小A:“嗯,好。下次我遇到abstract class和interface时我一定会特别小心选择的。”

大B其实一开始是看到小A对抽象类和接口的区别不是很看重,也看到他对它们是有一定的理解,但是他还是想挫挫小A的锐气。好让他知道学习编程可是重重要认真,就像做事一样,不得马虎。

大B:“刚才我跟你讲了从语法定义和编程的角度论述了abstract class和interface的区别,这些层面的区别是比较低层次的、非本质的。”

大B:“来,考虑这样一个例子,假设在我们的问题领域中有一个关于Door的抽象概念,该Door具有执行两个动作open和close,此时我们可以通过abstract class或者interface来定义一个表示该抽象概念的类型。”

定义方式分别如下所示:

使用abstract class方式定义Door:

```
abstract class Door{
```

```
abstract void open();
abstract void close();
}
```

使用interface方式定义Door：

```
interface Door{
void open();
void close();
}
```

大B：“其他具体的Door类型可以extends使用abstract class方式定义的Door或者implements使用interface方式定义的Door。看起来好像使用abstract class和interface没有大的区别。如果现在要求Door还要具有报警的功能。”

小A：“我们该如何设计针对该例子的类结构呢？”

大B：“可以在Door的定义中增加一个alarm方法。”

在Door的定义中增加一个alarm方法

```
abstract class Door{
abstract void open();
abstract void close();
abstract void alarm();
}
```

或者

```
interface Door{  
void open();  
void close();  
void alarm();  
}
```

具有报警功能的AlarmDoor的定义方式如下：

```
class AlarmDoor extends Door{  
void open(){...}  
void close(){...}  
void alarm(){...}  
}
```

或者

```
class AlarmDoor implements Door {  
void open(){...}  
void close(){...}  
void alarm(){...}  
}
```

大B：“你觉得这样做行得通吗？”

小A：“起码我是这样认为的！”

大B：“好，有自信。不错！但是，我还是得和你讲讲你这样做错在哪里了。你

这种方法违反了面向对象设计中的一个核心原则 ISP (Interface Segregation Principle), 在Door的定义中把Door概念本身固有的行为方法和另外一个概念 ‘报警器’ 的行为方法混在了一起。这样引起的一个问题是那些仅仅依赖于Door这个概念的模块会因为 ‘报警器’ 这个概念的改变 (比如: 修改alarm方法的参数) 而改变, 反之亦然。”

小A这回可是真的 “受伤” 了, 诚心地向师兄请教: “师兄, 那你认为应该怎么做会比较好哩? 请指教。”

大B: “呵呵。孺子可教也! 好。我跟你讲讲。既然open、close和alarm属于两个不同的概念, 根据ISP原则应该把它们分别定义在代表这两个概念的抽象类中。定义方式有: 这两个概念都使用abstract class方式定义; 两个概念都使用interface方式定义; 一个概念使用 abstract class 方式定义, 另一个概念使用interface方式定义。显然, 由于Java语言不支持多重继承, 所以两个概念都使用abstract class方式定义是不可行的。后面两种方式都是可行的, 但是对于它们的选择却反映出对于问题领域中的概念本质的理解、对于设计意图的反映是否正确、合理。”

小A还是似懂非懂的样子。大B接着说:

“我们——来分析、说明。如果两个概念都使用interface方式来定义, 那么就反映出两个问题: 1、我们可能没有理解清楚问题领域, AlarmDoor在概念本质上到底是Door还是报警器? 2、如果我们对于问题领域的理解没有问题, 比如: 我们通过对问题领域的分析发现AlarmDoor在概念本质上和Door是一致的, 那么我们在实现时就没有能够正确的揭示我们的设计意图, 因为在这两个概念的定义上 (均使用 interface方式定义) 反映不出上述含义。如果我们对于问题领域的理解是:

AlarmDoor在概念本质上是Door，同时它具有具有报警的功能。我们该如何来设计、实现来明确的反映出我们的意思呢？前面已经说过，abstract class在Java语言中表示一种继承关系，而继承关系在本质上是‘is-a’关系。所以对于Door这个概念，我们应该使用abstract class方式来定义。另外，AlarmDoor又具有报警功能，说明它又能够完成报警概念中定义的行为，所以报警概念可以通过interface方式定义。”

如下所示：

```
abstract class Door{
    abstract void open();
    abstract void close();
}
interface Alarm{
    void alarm();
}
class Alarm Door extends Door implements Alarm{
    void open(){...}
    void close(){...}
    void alarm(){...}
}
```

大B：“这种实现方式基本上能够明确的反映出我们对于问题领域的理解，正确的揭示我们的设计意图。其实abstract class表示的是‘is-a’关系，interface表示的是‘like-a’关系，大家在选择时可以作为一个依据，当然这是建立在对问题领域的理解上的，比如：如果我们认为AlarmDoor在概念本质上是‘报警器’，同时又具有Door的功能，那么上述的定义方式就要反过来了。”

小A：“这回我是明白了。”

大B还是不是很放心，接着说：“`abstract class` 和 `interface` 是Java语言中的两种定义抽象类的方式，它们之间有很大的相似性。但是对于它们的选择却又往往反映出对于问题领域中的概念本质的理解、对于设计意图的反映是否正确、合理，因为它们表现了概念间的不同的关系（虽然都能够实现需求的功能）。这其实也是语言的一种的惯用法，这样吧，我把刚才讲的都总结成几点，你记得时候方便一点，也记得牢一点。

- 1、`abstract class`在Java语言中表示的是一种继承关系，一个类只能使用一次继承关系。但是，一个类却可以实现多个 `interface`。
- 2、在 `abstract class` 中可以有自己的数据成员，也可以有非 `abstract` 的成员方法，而在 `interface` 中，只能够有静态的不能被修改的数据成员（也就是必须是 `static final` 的，不过在 `interface` 中一般不定义数据成员），所有的成员方法都是 `abstract` 的。
- 3、`abstract class` 和 `interface` 所反映出的设计理念不同。其实 `abstract class` 表示的是 ‘is-a’ 关系，`interface` 表示的是 “like-a” 关系。
- 4、实现抽象类和接口的类必须实现其中的所有方法。抽象类中可以有非抽象方法。接口中则不能有实现方法。
- 5、接口中定义的变量默认是 `public static final` 型，且必须给其初值，所以实现类中不能重新定义，也不能改变其值。
- 6、抽象类中的变量默认是 `friendly` 型，其值可以在子类中重新定义，也可以重新赋值。
- 7、接口中的方法默认都是 `public, abstract` 类型的。”

小A：“我把这些都记下来，回去一定好好记住它！”

2.5 接口和委托的区别

大B：“那你知道接口和委托的区别吗？”

小A：“接口可以包含属性，索引，方法以及事件。但委托不能包含事件。”

大B：“这回是对了。呵呵.....别伤心，其实你学得很好，再加倍努力一定可以学得更好的！”

小A今天可真是受益非浅呐。对大B师兄更多了一份敬意。

小A：“谢谢师兄！你放心，我一定会更加努力！”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第三章 我们班来了位新同学——适配器模式

3.1 我们班来了位新同学——适配器模式

时间：12月15日 地点：教室 人物：06信管全班同学

大二上学期的一天.....

早上到教室，依然是在响铃前3分钟。今天是上那讨厌的汇编，学了我大半个学期，到现在还不会编。看来今天这个节我又得晕呼着过去了。

正在我叹气的时候，教室里出现了骚动，还有好多人在窃窃私语哩。不少男生还吹起了口哨。怪了，今天有啥好事哩？正在我嘀咕的时候，班主任进来了，教室门口走进来一位美女？嘿嘿.....还是金发的！

不等我回神，班主任来了：“同学们，借大家上课前几分钟的时间，给大家说个事。今天，我们班来了位新同学，以后她会和大家一起学习。Anne，来，给大家做个自我介绍。”

台下一片骚动.....

只见坐在我前面的金发MM站起来了。非常有礼貌地说

“Good morning everyone, My name is Anne, I am from Canada, please take care of!”

妈呀，英语？头晕.....

只见她甜甜地笑着，我正晕呼，她接着说：“大家早上好，我叫Anne，我来自加拿大，请多关照！”

这回可听懂了，人家叫安妮哩！人漂亮名字也好听，这回可得迷倒我们班男生了。也难怪，在我理工院校，女生少之又少，这回突然来了位金发MM，我们班男生可有福了。不过那个MM也真是的，都会中文，为啥来个英语啊！

看来接下来我们的大学生活就热闹了.....

3.2 适配器模式

时间：12月16日 地点：大B房间 人物：大B，小A

都说学好普通话，走遍中国都不怕。就好像Anne来到我们班，如果说她只会说英文，那我们好大一部份同学都不一定听得懂。可是她还会说中文，那以后和大家在一起沟通就好多了。就好比 we 讲的适配器。

大B：“师弟，你知道怎样是适配器模式吗？”

小A：“就是把一个类的借口转换成客户端所期待的另一种接口，从而使原接口

不匹配而无法在一起工作的两个类能在一起工作。”

大B：“从功能上讲这些接口不兼容的类一般具有相同或相似的功能。通常我们通过修改该类的接口来解决这种接口不兼容的情形，但是如果我们不愿意为了一个应用而修改各原有的接口，或者我们压根就没有原有对象的源代码那该怎么办呢？此时Adapter模式就会派上大用场了。你能不能用代码来实现呐？”

小A：“好。我试一下。”

如果有两个编译好的(无源代码)类，类A有某些功能，但是需要一个xml读取模块才能工作，

这个模块要实现这个接口：

```
public interface XmlReader{  
    public InputStream xmlReader();  
}
```

你的另一个类B恰好有这个功能，但是B实现的是这个接口：

```
public interface ReaderXml{  
    public InputStream readerXml();  
}
```

这个时候我们的做法是写个适配器：

```
public class Adapter implements XmlReader extends B{
```

```
public InputStream xmlReader(){  
    return readerXml();  
}  
}
```

这个就是适配器模式了。

适配器模式还有另外一种实现方式：

```
public class Adapter implements XmlReader  
ReaderXml b = new B();  
public InputStream xmlReader(){  
    return b.readerXml();  
}  
}
```

大B：“对，没错！上次有个朋友从美国给我带回一个微波炉，但因为美国的生活用电电压是110V，而中国的电压是220V，所以我不能使用，幸好朋友有先见之明，给我带回一个变压器，能把220V电压转换成110V电压，我才可以放心使用了。”

小A：“嘿嘿！师兄你那位朋友还挺有心的嘛！一定是位很要好的朋友吧？”

大B：“还不就是那个大学时候的死党老E，那鸟人大学毕业后一直在国外，听说最近要回来一趟哩！”

小A：“是吗？那到时你们可爽了，又可以一起喝酒啦！”

大B：“嘿嘿！是啊！毕业几年一直没见过那鸟人，跑国外喝了几年养墨水，回

来一定得好好宰他一顿。”

小A：“那是要的啦！”

大B：“对了，你编程学得不错，能不能把刚才我说的，也就是微波炉电压转换用代码表示？”

小A：“好。通过适配，使c220类能在c110类中使用。”

程序代码：

```
#include
class c220v
{
public:
void DianYa220v()
{
cout<<"220v电压!"<<DianYa220v();
cout<<"经变压器转换成"110v电压"
```

运行结果：

220v电压！

经变压器转换成

110v电压

大B：“吼！非常不错喔！”

3.3 适配器模式的几个要素

大B：“那适配器模式有几个要素？”

小A：“我道还没有注意，给我说说适配器模式所涉及的角色有哪些吧！”

大B：“适配器模式所涉及的角色包括：目标、客户、被适配者、适配器。”

小A：“那这些要素主要都做些什么？”

大B：“目标（CTarget）：定义一个客户端使用的特定接口。客户（CClient）：使用目标接口，与和目标接口一致的对象合作。被适配者（CAdaptee）：一个现存需要匹配的接口。适配器（CAdapter）：负责将CAdaptee的接口转换成CTarget的接口。适配器是一个具体的类，这是本模式的核心。由此可见，但客户端调用Adapter接口时候，Adapter便会调用Adaptee的操作相应请求，该模式就完成了接口的适配过程。”

3.4 优势和缺陷

小A：“那适配器模式好在哪里？它又有什么缺陷呐？给我讲讲吧？”

大B：“适配器模式可以将一个类的接口和另一个类的接口匹配起来，使用的前提是你不能或不想修改原来的适配器母接口。例如，你向第三方购买了一些类、控件，但没有源程序，这时，使用适配器模式，你可以统一对象访问接口，但客户调用可能需要变动。”

3.5 何时使用适配器模式

小A：“使用一个已经存在的类，如果它的接口，方法和你的要求不相同的时候，可以考虑用适配器模式吗？”

大B：“可以啊！如果两个类所做的事情相同或相似，但是他们有不同的接口的时候要使用它。类都是共享同一个接口，那你想客户代码要怎么样才行？”

小A：“客户代码只要统一调用同一接口就行了，是不是这样简单，直接，更紧凑？”

大B：“是的，软件都是需要维护的，维护可能会因不同的开发人员，不同的产品，不同的厂家，造成功能类似但是接口不同，这时就可以使用适配器。”

小A：“你是说，在软件开发后期或维护的时候再考虑使用适配器？”

大B：“在设计阶段没必要把类似的功能类的接口设计的不同。”

小A：“可是不同的程序员定义方法的名称也可能是不同的呀！”

大B：“那也是，但是在一般公司内部，类和方法的命名是有规范的，做好前期就设计，接口不相同的时候，第一时间不应该考虑用适配器，而是考虑通过重构统一接口。”

小A：“也就是说要在双方都不太容易修改的时候，这个时候再使用适配器模式适配？不是一出现不同时就使用它？会不会有在设计初就考虑用适配器模式的情况哩？”

大B：“有，就好像在设计一个系统时使用第三方开发组件，这个组件的接口与系统接口不相同，这个时候就不用为了迎合它去改动自己的接口，在这种情况下，虽然是在开发设计阶段，解决接口不同的问题也可以用适配器模式。”

小A：“这样呐！”

大B：“有人举过这样一个例子：虎与飞禽是没有直接关联的两类动物，但是现在出来了个“飞虎”，它同时具有虎肉食动物跟飞禽会飞的特质，要在飞禽这个类系中添加一个成员类“飞虎”，除了直接实现“飞虎”类，还有一种简单的办法是实现一个Adapter类，在其中包容一个虎的对象，同时实现飞禽的接口即可。当然，对于这个问题，多继承或者实现多接口可能是一个更直观的作法，在实际应用中，可视具体需要确定采用何种作法。”

3.6 适配器总体上可以分为哪两类

小A：“适配器还要有类别之分的吗？”

大B：“是啊，根据重用使用方式的不同一般将适配器模式分为两类。”

小A：“喔。是吗？哪两类哩？”

大B：“类适配器和对象适配器。”

3.7 类适配器 VS 对象适配器

大B：“我还是跟你讲讲类适配器和对象适配器吧！举些例子，这样你就明白了。”

小A：“好！”

大B：“要正确地区别这两种适配器的区别，我们还是从一个简单的例子开始吧！我们的系统中有一个具有某个特定功能的类Adaptee，一个客户类Client——他需要一个实现Target接口的对象，和一个Target接口。” 以下是他们的源码：

```
//Adaptee.java
public class Adaptee{
    public void specialRequest(){
        System.out.println("Called SpecificRequest() in Adaptee ");
    }
}

//Client.java
public class Client {
    public static void main(String[] args){
        Target t = ..... //new Adapter() ;
        t.request();
    }
}

//Target.java
public interface Target{
    public void request();
}
```

“根据上一小节的分析我们知道此时需要一个Adapter对象，该对象实现Target接口，同时他又重用现有的Adaptee类。任何有一点点OO（面向对象）知识

的人都会想到通过继承可以达到重用的目的。”

下面是通过继承实现Adaptee类重用的例子：

```
//Adapter.java
public class Adapter extends Adaptee implements Target{
    public void request(){
        this.specialRequest();
    }
}
```

大B：“看，简单明了吧！现在的过程就是：客户调用Target接口的request方法，实际就是调用其父类Adaptee的specialRequest方法。”

小A：“嘿嘿！这样一说倒真的是简单喔！”

大B：“对啊！这就是大家通常说的类的适配器！类适配器具有以下的两个特点：1、适配器类（Adapter）实现Target接口；2、适配器类（Adapter）通过继承来实现对Adaptee类的重用。”

下面是一个通过组合关系实现继承的例子，以下是源码：

```
//Adapter.java
public class Adapter implements Target{
    Adaptee adaptee = new Adaptee();
    public void request(){
        adaptee.specialRequest();
    }
}
```

对于这两者不同的适配器客户代码其实是完全一样的。以下是客户的代码：

```
//Client.java
public class Client {
    public static void main(String[] args){
        Target t = new Adapter() ;
        t.request();
    }
}
```

大B：“现在可是全明白了吧？”

小A：“嘿嘿！现在都懂了。”

3.8 类适配器和对象适配器的哪些不同？

小A：“那类适配器和对象适配器有哪些不同哩？”

大B：“类适配器是通过继承类适配者类（Adaptee Class）实现的，另外类适配器实现客户类所需要的接口。当客户对象调用适配器类方法的时候，适配器内部调用它所继承的适配者的方法。对象适配器包含一个适配器者的引用（reference），与类适配器相同，对象适配器也实现了客户类需要的接口。当客户对象调用对象适配器的方法的时候，对象适配器调它所包含的适配器者实例的适当方法。”

3.9 日常生活中的适配器

大B：“适配器的例子在日常生活中随处可见。”

小A：“喔？”

大B：“新的电脑鼠标一般都是USB接口，而旧的电脑机箱上根本就没有USB接口，而只有一个PS2接口，这时就必须有一个PS2转USB的适配器，将PS2接口适配为USB接口。一般家庭中电源插座有的是两个孔（两项式）的，也有三个孔（三项式）的。很多时候我们可能更多地使用三个引脚的插头，但是那种两孔的插座就不能满足我们的需求，此时我们一般会买一个拖线板，该拖线板的插头是两脚插头，这样就可以插入原先的两孔插座，同时拖线板上带有很多两孔、三孔的插座！这样不仅可以扩容，更主要的是将两孔的插座转变为三孔的插座！”

小A：“嘿嘿！仔细想来好像是有好多适配器的例子喔！”

大B：“设计模式里的适配器模式和日常生活中的适配器的作用是完全一样的。”

3.10 电脑电源适配器

小A想去电脑城买个笔记本电脑电源适配器，这天他来找师兄先了解一些关于电脑电源适配器。

小A：“师兄，什么样的的是笔记本电脑的电源适配器？”

大B：“多数笔记本电脑的电源适配器可以自动检测100~240V交流电(50/60Hz)。基本上所有的笔记本电脑都把电源外置，用一条线和主机连接，这样可以缩小主机的体积和重量，只有极少数的机型把电源内置在主机内。在电源适配器上都有一个铭牌，上面标示着功率，输入输出电压和电流量等指标，特别要注意输入电压的范围，这就是所谓的‘旅行电源适配器’，如果到市电电压只有100V的国家时，这个特性就很有用了，有些水货笔记本电脑是只在原产地销售的，没有这种设计，甚至只有100V的单一输入电压，在我国的220V市电电压下插上就会烧毁。”

小A：“电源适配器？”

大B：“嗯，对。电源适配器是小型便携式电子设备及电子电器的供电电源变换设备，一般由外壳、电源变压器和整流电路组成，按其输出类型可分为交流输出型和直流输出型；按连接方式可分为插墙式和桌面式。广泛配套于电话子母机、游戏机、语言复读机、随身听、笔记本计算机、蜂窝电话等设备中。”

小A：“师兄，手提电脑电源适配器要怎么买才好啊？”

大B：“呵呵！这样说起来要注意的问题就多了喔！”

小A：“是么？那电源适配器的标称电压和电流是什么意思啊？”

大B：“首先，一般电源适配器（以下简称电源）标称的电压，是指开路输出的电压，也就是外面不接任何负载，没有电流输出时候的电压，所以也可以理解为，此电压就是电源输出电压的上限。对于电源内部使用了主动稳压的元件的情况下，即使市电电压有所波动，其输出也是恒定值，象市面上一般的小变压器，比如随身听之类配电源，如果市电波动，该电源的输出也会随之波动的。一般来讲普通电

源适配器的真正空载电压也不一定和标称电压完全一致，因为电子元件的特性不可能完全一致，所以有一定的误差，误差越小，对电子元件的一致性要求越高，生产的成本就高了，所以价格也就贵一些了。另外，关于标称的电流值，无论任何电源都有一定的内阻，因此当电源输出电流的时候，会在内部产生压降，导致两件事情，一个是产生热量，所以电源会热，另一个是导致输出电压降低，相当于内部消耗。”

小A：“那都是同样标称电压的电源，输出电流不同，能不能用在同一台本本上？”

大B：“电源电压一样，输出电流不同，能不能用在同一台本本上。基本的原则是大标称电流的电源可以代替小标称电流的电源。估计有人会这样想，觉得大标称电流的电源会烧坏本本，因为电流大了嘛。实际上电流多大在电压相同的情况下取决于负载，当本本高负荷运转的时候，电流大些，本本进入待机的时候，电流就小些，大标称电流的电源有足够的电流余量。反之，有人用56w的电源代替72w的用起来也没什么问题，原因是通常电源适配器的设计留有一定的余量，负载功率都要小于电源功率，所以这种代替在一般使用上是可行的，但是剩余的电源功率余量就很少了，一旦你的本本接了很多外设，比如两块usb硬盘，然后cpu全速运转，再有一个底座，上面来个光驱全速读盘，再加上同时给电池充电，估计就危险了，要随时用手摸摸你的电源是不是已经可以煮鸡蛋了。所以最好不要用小电流电源代替大电流电源。”

小A：“师兄呐，为什么一模一样的机器，别人的电源温温的，我的总是很烫？”

大B：“先不要怀疑你的电源有问题，先看看你的本本在干什么，是不是像上面

说的两块USB硬盘，CPU全速运转，硬盘疯狂读写，光驱全速读盘，同时给电池充电，大声放着音乐，屏幕亮度最大，无线网卡一直在侦测信号等等，善用电源管理，根据任务合理调整本本的工作状态是很重要的。”

小A：“那电源标称电压比我的本本电池电压高很多，不会出事吧？”

大B：“首先，要知道的是，电源给本本供电与电池给本本供电是不同的。电池供电，电池的输出是纯直流，干净得很，电池的电压既不可能也不需要设计得很高，锂电池的化学特性决定了一节电芯的输出电压只能在3.6V左右，所以很多电池都是采用三级串联的方式，10.8V也就成了很流行的电池电压。有些电池的标称值比3.6V的整数倍稍大一些，比如3.7V或者11.2V等等，其实是为了保护电池。电源供电，情况就复杂一些，首先需要对加入电压进行进一步的稳压滤波，以保证在电源性能不很好的情况下稳定工作，稳压后的电压分成两个部分，一路给本本工作供电，另一路给电池充电，给本本供电的那部分同电池供电的时候相同，而给电池充电的那部分需要通过电池的充电控制电路才可以加在电芯上，控制电路可以很复杂，所以电源电压必须大于电芯电压才有充分的能力供应给充电控制电路的各单元。最后真正加到电芯上的电压决不会是你的电源标称的电压。放心好了。另外搜索了一下，大家看看电池的充电控制电路包括些什么东西，应该包括初级稳压、精密可调谐稳压、可控硅调节脉动输出、稳压输出、电流反馈、芯片充电过程记录与运算、充电程序自反馈调节参数等等，呵呵，是不是感觉还是很需要消耗一部分电能的啊。”

小A：“为什么理论上原配的电源通常比非原配的电源要好？”

大B：“理论上来说。原装的电源肯定好一些，但是，实际使用可能感觉不到差别。通常我们的设备都有一个电压输入的安全范围，比如一个2.5的移动硬盘，它是

要求5V加减5%，过高或者过低，保护电路会停止设备的工作。如果保护电路启动，那说明在这之前，你的机器已经接近或者超过了它设计所能承受的上限或者下限，这对机器的寿命都是有影响的。（当然，这里提到的非原装电源，还是做工好的，劣质产品，就不要提了。）更重要的问题可能是本本的数据安全，突然的自动保护而停止工作对于计算机来说是很恐怖的事情，尤其是很多不用电池的朋友。有的时候计算机莫名其妙的重新启动也和此有关。对于原装的电源来说，厂家很清楚自己的电源需要有多大的负载能力，计算出来的安全的标称电压电流肯定准确的多。然而如果使用的是非原配适配器，比如通用型的变压器之类，上述问题不能得到认真考虑，这是用户就只能从电源参数上尽量想办法获得兼容，但是每种适配器的内阻是不同的，标称电压的允许误差可能不同，标称电流输出下电压的变化范围的也可能有所不同，如果你不是仔细测量了相关数值，肯定，是有风险的。这就是原装与非原装的区别，这里可没说到劣质电源啊，这样的货色，肯定是更不保险的。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第四章 金融危机股票还挣钱？——外观模式

4.1 金融危机股票还挣钱？

时间：12月17日 地点：大B房间 人物：大B，小A

小A：“大B，你今年有炒股票吗？”

大B：“有啊！买了一些。”

大B很奇怪为什么小A突然问这个问题：“怎么会这么问哩？”

小A：“我们班上好多同学都炒了，不过大部份都还是新手，都不太懂的。”

大B：“那他们的股票今年都怎么样呐？”

小A：“都亏了，今年什么形式呐！”

大B：“今年很多炒股的人都亏了。”

小A：“他们都是刚入市不久的，什么都不懂。对了，好像有个还可以，赚了不少钱，具体是怎么样也不太清楚。只是天天下课后，在教室里，那些买了股票的同

学就会聚在一起讨论今天股市的走势情况，有的边上课还边有手机上网关注股市哩！”

大B：“呵呵.....都是新股民呐，其实按今年的行情买股票是应该很小心的。”

小A：“嘿嘿！都是学生，买了一点股票，天天在关心，不过也是，自己的买了股票，就今年的行市，谁能不关心呐！特别是前段时间，从6000多点降到了2000点以下。几多人伤心，几多人愁呐！”

大B：“现在的学生是有不少在炒股票的。不过应该买得都不多吧？”

小A：“是啊。大部份都不多的。都还是学生嘛！对了，师兄，你的股票今年走势如何啊？”

大B：“我的啊？今年还好，今年还是挣钱的。”

小A：“是么？呀！师兄，你真厉害！”

大B：“我只是炒短线，平时对股票也比较在研究，刚好上次跌的时候我的原来买的卖了，后来一直跌，我都是小心地进仓，挣得不多，但还算是可以的。你有没有买股票呐？”

小A：“股票我是没有买，买了一些基金。唉，我也是个新手，想着不会炒股，就买个基金算了，结果遇到今年形式不好，还是亏得只有一半了。”

大B：“基金？挺好啊！它将投资者分散的资金集中起来，交由专业的经理人进行管理，投资于股票，债券，外汇等领域。而基金投资的收益归持有人所有。管理机构只收取一定比例的托管费用。不过今年金融风暴横扫全球，全世界经济受到严

重影响。”

小A：“是啊，所以现在也没办法。师兄，你怎么在现在这样金融危机股票还挣钱？”

大B：“因为我时时都在关注时事，而且我主要了解股票的各种信息，结合现今时事预测它的未来。还要买入和卖出的时机合适，这是很难帮到的。”

小A：“投资买股票，做不好的原因和软件开发类中的什么类似？而投资者买基金，基金经理人拿这些钱去投资，然后获利后分给大家，这体现了什么？”

大B：“由于大多投资者对众多股票的联系太多，反而不利于股票操作，这在软件中叫耦合性过高。而基金，是众多投资者只和基金打交道，只关心基金的上涨和下跌就可以了，而实际在操作的是基金经理人。”

4.2 外观模式意图

股市有风险入市须小心啊！这就是外观模式。

大B：“师弟，你现在明白什么是外观模式没有啊？”

小A：“明白了。外观模式定义了一个将子系统的一组接口集成在一起的高层接口，以提供一个一致的界面。通过这个界面，其他系统可以方便地调用子系统功能，而忽略子系统内部发生的变化。”

大B：“是的。”

4.3 使用场合

大B：“说说在什么情况下可以使用它吧！”

小A：“1、为一个比较复杂的子系统提供一个简单的接口。2、将客户程序与子系统的实现部分分离，提高子系统的独立性和可移植性。3、简化子系统间的依赖关系。”

大B：“外观模式 (Facade pattern) 涉及到子系统的一些类。所谓子系统，是为提供一系列相关的特征 (功能) 而紧密关联的一组类。例如，一个Account类、Address类和CreditCard类相互关联，成为子系统的一部分，提供在线客户的特征。在真实的应用系统中，一个子系统可能由很多类组成。子系统的客户为了它们的需要，需要和子系统中的一些类进行交互。客户和子系统的类进行直接的交互会导致客户端对象和子系统 (Figure1) 之间高度耦合。任何的类似于对子系统中类的接口的修改，会对依赖于它的所有的客户类造成影响。如图4-1所示”

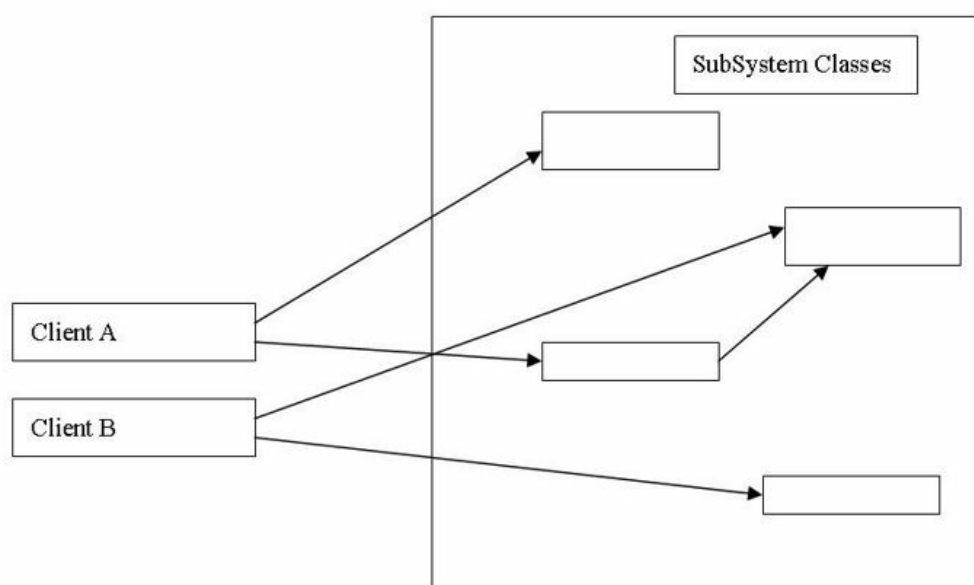


图4-1 外观模式

小A：“这样说来，外观模式（Facade pattern）很适用于在上述情况。”

大B：“是啊。外观模式（Facade pattern）为子系统提供了一个更高层次、更简单的接口，从而降低了子系统的复杂度和依赖。这使得子系统更易于使用和管理。外观是一个能为子系统和客户提供简单接口的类。当正确的应用外观，客户不再直接和子系统类交互，而是与外观交互。外观承担与子系统中类交互的责任。实际上，外观是子系统与客户的接口，这样外观模式降低了子系统和客户的耦合度(如图4-2所示)。”

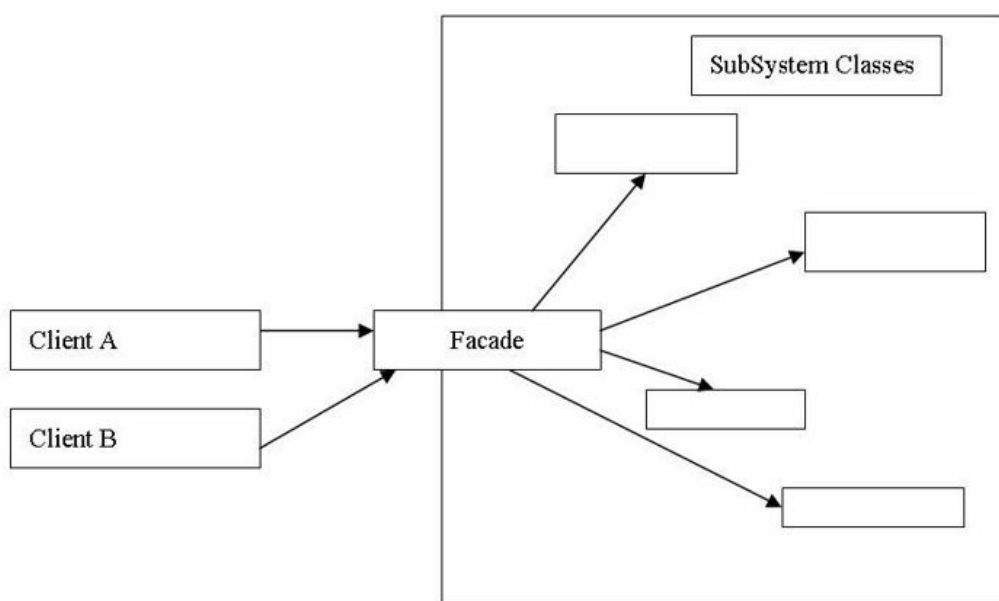


图4-2 外观模式降低了子系统和客户的耦合度

大B：“从图4-2中我们可以看到：外观对象隔离了客户和子系统对象，从而降低了耦合度。当子系统类中的类进行改变时，客户端不会像以前一样受到影响。尽管客户使用由外观提供的简单接口，但是当需要的时候，客户端还是可以视外观不存在，直接访问子系统类中的底层次的接口。这种情况下，它们之间的依赖/耦合度和原来一样。”

4.4 例子

大B：“我给你举个例子来说明吧。”

小A：“嗯。好的。”

大B：“让我们建立一个应用：1、接受客户的详细资料（账户、地址和信用卡信息）2、验证输入的信息3、保存输入的信息到相应的文件中。这个应用有三个类：Account、Address和CreditCard。每一个类都有自己的验证和保存数据的方法。”

```
Listing1: AccountClass
public class Account {
    String firstName;
    String lastName;
    final String ACCOUNT_DATA_FILE = "AccountData.txt";
    public Account(String fname, String lname) {
        firstName = fname;
        lastName = lname;
    }
    public boolean isValid() {
        /*
        Let's go with simpler validation
        here to keep the example simpler.
        */
    }
    public boolean save() {
        FileUtil futil = new FileUtil();
        String dataLine = getLastName() + " ," + getFirstName();
        return futil.writeToFile(ACCOUNT_DATA_FILE, dataLine,true, true);
    }
}
```

```

}
public String getFirstName() {
return firstName;
}
public String getLastName() {
return lastName;
}
}

```

Listing2: Address Class

```

public class Address {
String address;
String city;
String state;
final String ADDRESS_DATA_FILE = "Address.txt";
public Address(String add, String cty, String st) {
address = add;
city = cty;
state = st;
}
public boolean isValid() {
/*
The address validation algorithm
could be complex in real-world
applications.
Let's go with simpler validation
here to keep the example simpler.
*/
if (getState().trim().length() < 2)
return false;
return true;
}
public boolean save() {

```



```

FileUtil futil = new FileUtil();
String dataLine = getAddress() + " ," + getCity() + " ," + getState();
return futil.writeToFile(ADDRESS_DATA_FILE, dataLine,true, true);
}

public String getAddress() {
return address;
}

public String getCity() {
return city;
}

public String getState() {
return state;
}
}

```

Listing3: CreditCard Class

```

public class CreditCard {
String cardType;
String cardNumber;
String cardExpDate;
final String CC_DATA_FILE = "CC.txt";
public CreditCard(String ccType, String ccNumber,
String ccExpDate) {
cardType = ccType;
cardNumber = ccNumber;
cardExpDate = ccExpDate;
}

public boolean isValid() {
/*
Let's go with simpler validation
here to keep the example simpler.
*/
if (getCardType().equals(AccountManager.VISA)) {

```

```
return (getCardNumber().trim().length() == 16);
}
if (getCardType().equals(AccountManager.DISCOVER)) {
return (getCardNumber().trim().length() == 15);
}
if (getCardType().equals(AccountManager.MASTER)) {
return (getCardNumber().trim().length() == 16);
}
return false;
}
public boolean save() {
FileUtil futil = new FileUtil();
String dataLine = getCardType() + , " " + getCardNumber() + " , " + getCardExpDate();
return futil.writeToFile(CC_DATA_FILE, dataLine, true, true);
}
public String getCardType() {
return cardType;
}
public String getCardNumber() {
return cardNumber;
}
public String getCardExpDate() {
return cardExpDate;
}
}
```

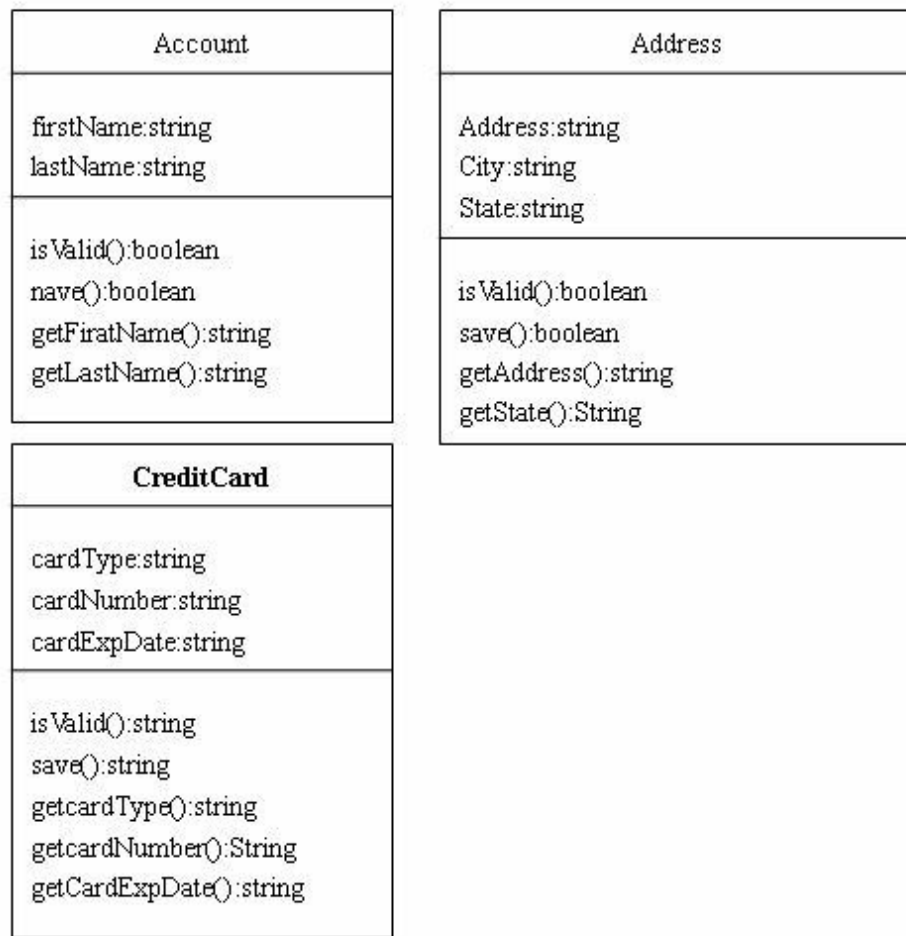


图4-3 验证并保存客户数据

大B：“建立一个客户AccountManager，它提供用户输入数据的用户界面。”

Listing4: Client AccountManager Class

```

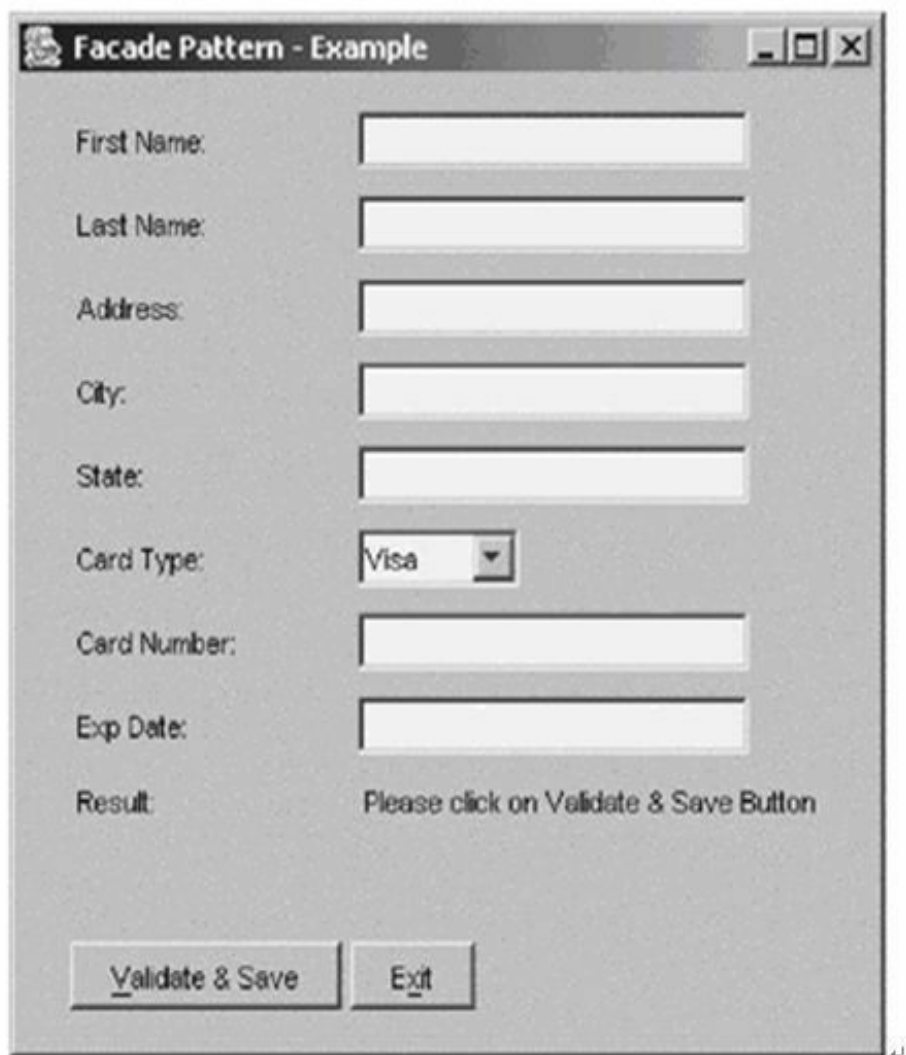
public class AccountManager extends JFrame {
    public static final -0*963String newline = "\n";
    public static final String VALIDATE_SAVE = "Validate & Save";
    public AccountManager() {
        super(" Facade Pattern - Example ");
        cmbCardType = new JComboBox();
        cmbCardType.addItem(AccountManager.VISA);
        cmbCardType.addItem(AccountManager.MASTER);
        cmbCardType.addItem(AccountManager.DISCOVER);
        //Create buttons
    }
}
  
```

```

JButton validateSaveButton = new JButton(AccountManager.VALIDATE_SAVE);
}
public String getFirstName() {
return txtFirstName.getText();
}
}
} //End of class AccountManager

```

当客户AccountManage运行的时候，展示的用户接口如下：



The image shows a Java Swing window titled "Facade Pattern - Example". The window contains a form with the following elements:

- First Name:
- Last Name:
- Address:
- City:
- State:
- Card Type: (dropdown menu)
- Card Number:
- Exp Date:
- Result: Please click on Validate & Save Button
- Buttons: and

图4-4 用户接口

大B：“为了验证和保存输入的数据，客户AccountManager需要：1、建立Account、Address和CreditCard对象。2、用这些对象验证输入的数据3、用这

些对象保存输入的数据。”

图4-5是对象间的交互顺序图：

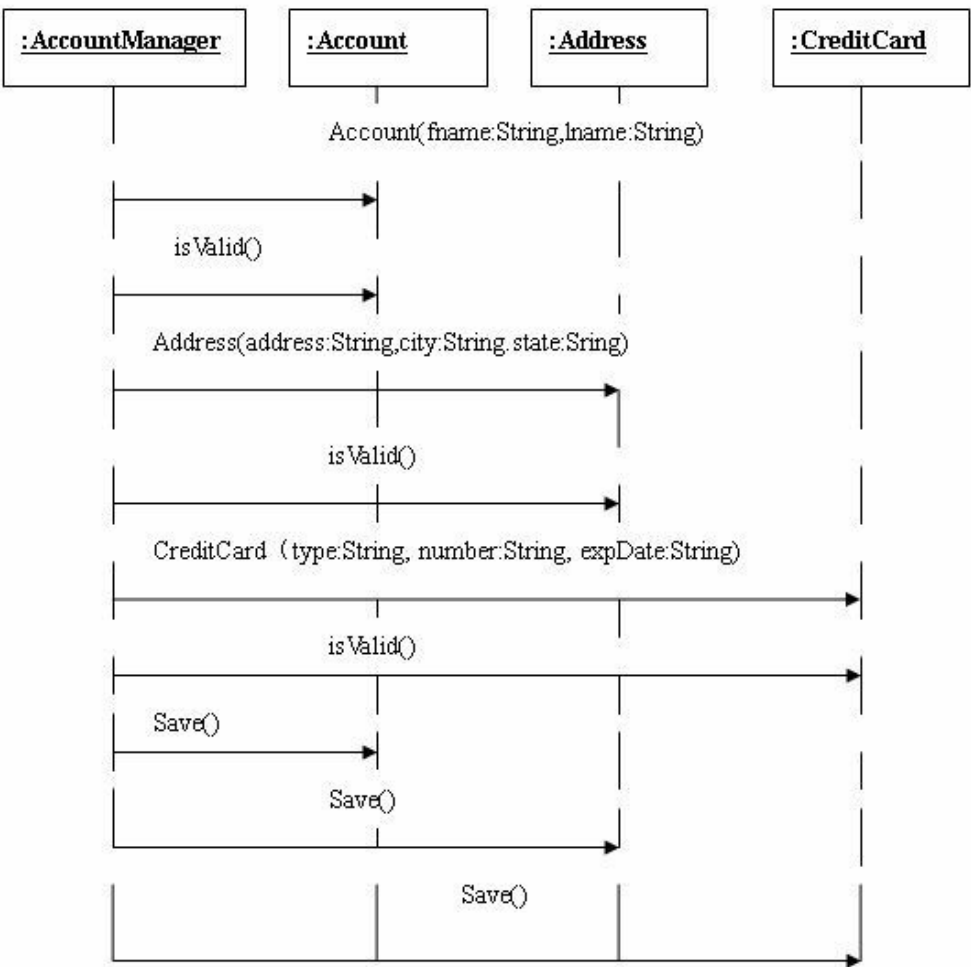


图4-5 对象间的交互顺序图

大B：“在这个例子中应用外观模式是一个很好的设计，它可以降低客户和子系统组件（Address、Account和CreditCard）之间的耦合度。应用外观模式，定义一个外观类CustomerFacade（Figure6 and Listing5）。它为由客户数据处理类（Address、Account和CreditCard）所组成的子系统提供一个高层次的、简单的接口。”

```
CustomerFacade
address:String
```

```

city:String
state:String
cardType:String
cardNumber:String
cardExpDate:String
fname:String
lname:String
setAddress(inAddress:String)
setCity(inCity:String)
setState(inState:String)
setCardType(inCardType:String)
setCardNumber(inCardNumber:String)
setCardExpDate(inCardExpDate:String)
setFName(inFName:String)
setLName(inLName:String)
saveCustomerData()

```

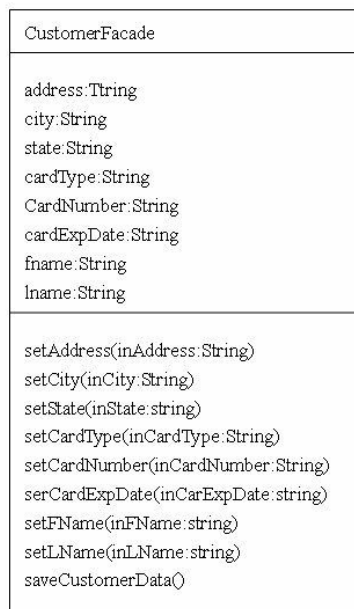


图4-6 外观类

Listing5: CustomerFacade Class

```

public class CustomerFacade {

```

```
private String address;
private String city;
private String state;
private String cardType;
private String cardNumber;
private String cardExpDate;
private String fname;
private String lname;
public void setAddress(String inAddress) {
    address = inAddress;
}
public void setCity(String inCity) {
    city = inCity;
}
public void setState(String inState) {
    state = inState;
}
public void setFName(String inFName) {
    fname = inFName;
}
public void setLName(String inLName) {
    lname = inLName;
}
public void setCardType(String inCardType) {
    cardType = inCardType;
}
public void setCardNumber(String inCardNumber) {
    cardNumber = inCardNumber;
}
public void setCardExpDate(String inCardExpDate) {
    cardExpDate = inCardExpDate;
}
```

```

public boolean saveCustomerData() {
    Address objAddress;
    Account objAccount;
    CreditCard objCreditCard;
    /*
    client is transparent from the following
    set of subsystem related operations.
    */
    boolean validData = true;
    String errorMessage = "";
    objAccount = new Account(fname, lname);
    if (objAccount.isValid() == false) {
        validData = false;
        errorMessage = "Invalid FirstName/LastName";
    }
    objAddress = new Address(address, city, state);
    if (objAddress.isValid() == false) {
        validData = false;
        errorMessage = "Invalid Address/City/State";
    }
    objCreditCard = new CreditCard(cardType, cardNumber, cardExpDate);
    if (objCreditCard.isValid() == false) {
        validData = false;
        errorMessage = "Invalid CreditCard Info";
    }
    if (!validData) {
        System.out.println(errorMessage);
        return false;
    }
    if (objAddress.save() && objAccount.save() && objCreditCard.save()) {
        return true;
    } else {

```



```
return false;
}
}
}
```

大B：“CustomerFacade类以saveCustomData方法的形式提供了业务层次上的服务。客户AccountManager不是直接和子系统的每一个组件交互，而是使用了由CustomFacade对象提供的验证和保存客户数据的更高层次、更简单的接口(如图4-7所示)。”

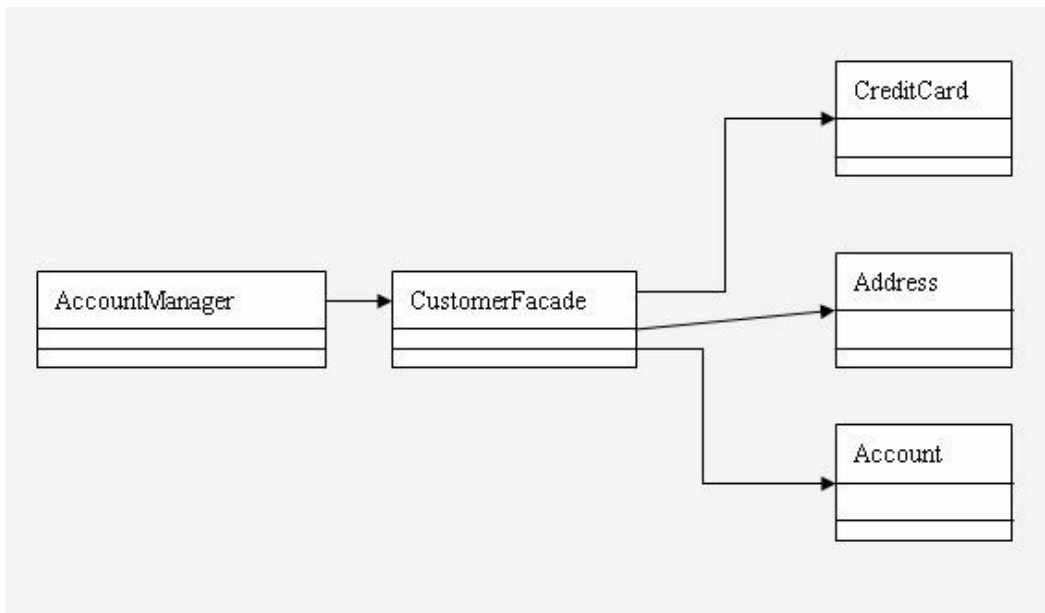


图4-7 验证和保存客户数据

大B：“在新的设计中，为了验证和保存客户数据，客户需要：1、建立或获得外观对象CustomFacade的一个实例。2、传递数据给CustomFacade实例进行验证和保存。3、调用CustomFacade实例上的saveCustomData方法。CustomFacade处理创建子系统中必要的对象并且调用这些对象上相应的验证、保存客户数据的方法这些细节问题。客户不再需要直接访问任何的子系统对象。”

图4-8展示了新的设计的消息流图：

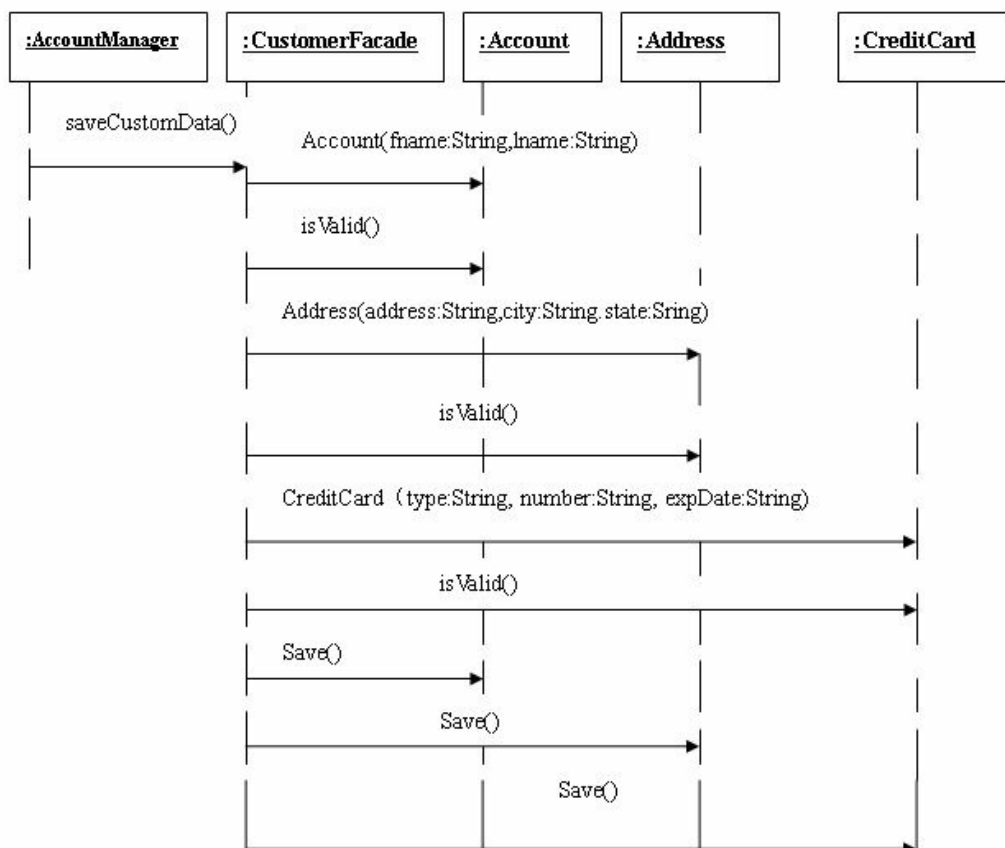


图4-8 新的设计的消息流图

4.5 应用外观模式的注意事项

小A：“师兄，应用外观模式有哪些注意事项？”

大B：“应用外观模式要注意以下事项：1、在设计外观时，不需要增加额外的功能。2、不要从外观方法中返回子系统组件给客户。例如：有一个下面的方法：`CreditCard getCreditCard()` 会报漏子系统的细节给客户。应用就不能从应用外观模式中取得最大的好处。3、应用外观的目的是提供一个高层次的接口。因此，外观方法最适合提供特定的高层次的业务服务，而不是进行底层次的单独的业务执行。”

小A：“明白。我记住了。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质
电子书下载！！！！

第五章 生日礼物——组合模式

5.1 生日礼物

时间：12月19日 地点：商场 人物：大B和他女朋友

大B和他的女朋友是大学同学，他们在一起渡过了美好的大学时光。今天又到了女朋友过生日，大B自然要送上礼物以表心意。

MM：“今天我过生日，你可要送我一件礼物喔！”

大B：“嗯，好吧。今年就好好地补偿你。”

MM：“好，那就要把过去的都补回来喔！”

大B：“好啊，没问题。”

MM：“那你打算怎么补偿我啊？”

大B：“去商店，你自己挑。怎么样？”

MM：“这还差不多。”

大B：“那当然，谁叫我是你男朋友哩。”

MM高兴地拉着大B去逛商店。刚好现在是新货上市的时候，各式各样的服装，饰品，琳琅满目，而且特别有个性。大B的女朋友看到这个也喜欢，那个也想要，结果什么都想买。

MM：“这件T恤挺漂亮，买，这条裙子好看，买，这个包也不错，买。”

大B：“喂，买了三件了呀！我只答应送一件礼物的哦。”

MM：“什么呀！T恤加裙子加包包，正好配成一套呀，小姐，麻烦你包起来。”

大B：“.....”（口吐白沫，四肢抽搐，不省人事ing...）

5.2 组合模式

时间：12月19日 地点：大B房间 人物：大B，小A

大B的MM十分之聪明地运用了组合模式，将3件礼物“合成”至1件（套），从而让大B以买一件礼物的方式（反正大B只要付一次钱~~）买了3件礼物。由此不难看出，组合模式将对象以树形结构组织起来，以达到“部分 - 整体”的层次结构，使得客户（钱包瘪瘪的大B）对单个对象和组合对象（1件vs1套）的使用具有一致性（付一次钱）。

大B：“你知道什么是组合模式吗？”

小A：“嗯，将对象组合成树形结构以表示‘部分-整体’的层次结构。组合模

式使得用户对单个对象和组合对象的使用具有一致性。（来自GOF定义）”

大B：“组合模式（又为‘部分-整体’模式）屏蔽了容器对象与单个对象在使用时的差异，为客户端提供统一的操作接口，从而降低客户代码与被调用对象的耦合关系，方便系统的维护与扩展。”

5.3 结构图

大B：“下面是组合模式的结构图如图5-1所示。”

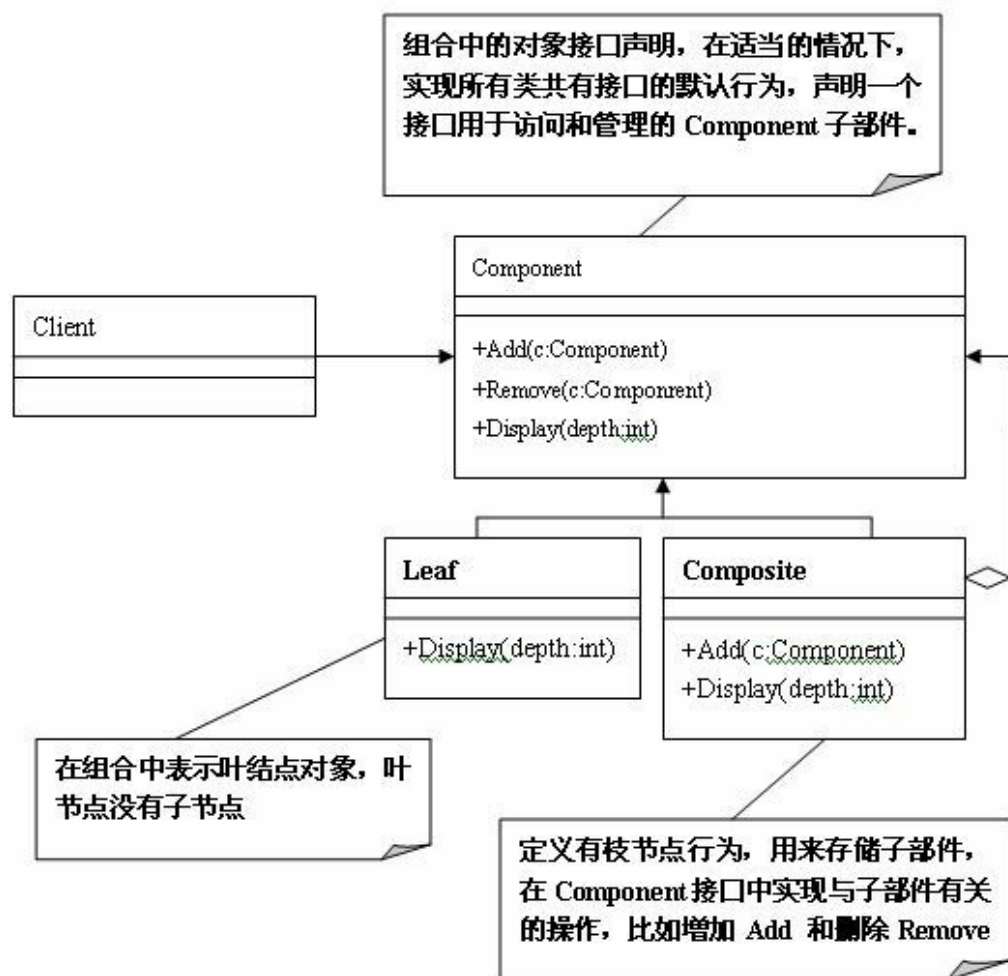


图5-1 结构图

大B：“组合模式为组合中的对象声明接口，在适当的情况下，实现所有类共有接口的默认行为。声明一个接口用于访问和管理组合模式的子部件。”

```
abstract class Component
{
    protected string name;
    public Component(string name)
    {
        this.name=name;
    }
    public abstract void Add(component c);//通常都用Add和Remove方法来提供增加或移出树叶或树枝的功能
    public abstract void Remove(Component c);
    public abstract void Display(in depth);
}
```

Leaf在组合中表示叶节点对象，叶节点没有子节点

```
class Leaf:Component
{
    public Leaf(string name):base(name)
    {}
    public override void Add(Component c)
    //由于叶节点没有再增加分枝和树叶，所以Add和Remove方法实现
    {
        Console.WriteLine("Cannot add to a leaf");
        //它没有意义，但这样可以消除叶节点和枝节点对象在抽象层次的区别
    }
    // 它们具备完全一致的接口
    public override void Remove(Component c)
```

```

{
    Console.WriteLine("Cannot remove to a leaf");
}
public override void Display(int depth)
{
    //叶节点的具体方法，此处是显示其名称和级别
    Console.WriteLine();
}
}

```

Composite定义有枝节点行为，用来存储子部件，在Component接口中实现与子部件有关的操作，比如增加Add和删除Remove

```

class Composite:Component
{
    private List children=new List();
    public Composite(string name):base(name)
    {}
    public override void Add(Component c)
    {
        children.add(c);
    }
    public override void Remove(Component c)
    {
        children.Remove(c);
    }
    public override void Display(int depth)
    { //显示枝节点名称，并对其下级进行遍历
        Console.WriteLine(new string('-',depth)+name);
        foreach(Component component in children)

```



```

{
component.Display(depth+2);
}
}
}

```

客户端代码，能通过Component接口操作组合部件的对象

```

static void Main(string[] args)
{
Component root=new Component("root");
root.Add(new Leaf("Leaf A"));//生成树根root,根上长出两叶
root.Add(new Leaf("Leaf B"));//LeafA与LeafB
Composite comp=new Composite("Componsite X");
comp.Add(new Leaf("Leaf XA"));
comp.Add(new Leaf("Leaf XB"));
root.Add(comp);
Composite comp2=new Composite("Composite XY");
comp2.Add(new Leaf("Leaf XYA"));
comp2.Add(new Leaf("Leaf XYB"));
comp.Add(comp2);
//根部又长出两页LeafC和LeafD,可惜LeafD没有长牢，被风吹走了
root.Add(new Leaf("Leaf c"));
Leaf leaf=new Leaf("Leaf D");
root.Add(leaf);
root.Remove(leaf);
root,Display(1);//显示大树的样子
}

```

显示结果：

-root

---leaf A

---leaf B

---Composite X

-----Leaf XA

-----Leaf XB

-----Composite XY

-----Composite XYA

-----Composite XYB

---Leaf c

大B：“现在你能用代码以组合模式，试写一下我给我女朋友买生日礼物。”

小A：“OK”

代码：

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Composite
{
```

```
interface IGift
{
void Pay();
void Add(IGift gift);
}

class GiftSingle : IGift
{
private string m_name;
public GiftSingle(string name)
{
m_name = name;
}
public void Add(IGift gift)
{
}
public void Pay()
{
Console.WriteLine("我买了" + m_name + "! hoho~~");
}
};

class GiftComposite : IGift
{
private string m_name;
List m_gifts;
public GiftComposite()
{
m_name = string.Empty;
m_gifts = new List();
}
public void Add(IGift gift)
{
m_gifts.Add(gift);
}
```

```

}
public void Pay()
{
foreach(IGift gift in m_gifts)
{
gift.Pay();
}
}
};
class Program
{
static void Main(string[] args)
{
//20岁生日，那时的MM还很单纯~~
Console.WriteLine("lalala~20岁生日来咯-----");
IGift singleGift20 = new GiftSingle("手表");
singleGift20.Pay();
//22岁生日，MM变得狡诈"了~~
Console.WriteLine("heiheihei~22岁生日来咯-----");
IGift compositeGift22 = new GiftComposite();
//打包，打包！我要把所有喜欢的礼物打包成“一套”~
compositeGift22.Add(new GiftSingle("手机"));
compositeGift22.Add(new GiftSingle("DC"));
compositeGift22.Add(new GiftSingle("DV"));
compositeGift22.Pay();
//24岁生日.....天哪！
Console.WriteLine("hiahiahia~24岁生日来咯-----");
//先逛商场一层~买化妆品！
IGift compositeGift24 = new GiftComposite();
//打包，打包！
compositeGift24.Add(new GiftSingle("香水"));
compositeGift24.Add(new GiftSingle("指甲油"));

```

```
compositeGift24.Add(new GiftSingle("眼影"));
//然后来到二层，看中了一套衣服~
IGift singleGift24 = new GiftSingle("衣服");
//因为只能买“一件”，所以“狡诈”的MM再次打包。。。
IGift totalGifts = new GiftComposite();
//我包，我包，我包包包！
totalGifts.Add(compositeGift24);
totalGifts.Add(singleGift24);
totalGifts.Pay();
}
}
}
```

大B：“嘿嘿！不错喔！”

5.4 组合模式的使用

小A：“组合模式比较简单，也很容易学习，当你面对一个树形结构的时候，脑筋就该多转一圈：是否可以在该结构中使用组合模式模式？”

大B：“我跟你讲一种简单的方法：定义一个公用的接口，让组合对象和单个对象都去实现该接口。因此，如果面对单个对象，则调用单个对象的方法；如果面对组合对象，递归遍历之，依次调用每个对象的方法；单个对象：相当于树形结构中的叶节点，它不包含任何子对象。”

小A：“如何去实现组合模式模式呢？”

大B：“组合对象相当于树形结构中的枝节点，它可以包含更小的枝对象，也可以包含叶对象。下面的代码是以抽象类定义，一般尽量用接口interface。”

```
public abstract class Equipment
{
    private String name;
    //实价
    public abstract double netPrice();
    //折扣价格
    public abstract double discountPrice();
    //增加部件方法
    public boolean add(Equipment equipment) { return false; }
    //删除部件方法
    public boolean remove(Equipment equipment) { return false; }
    //注重这里，这里就提供一种用于访问组合体类的部件方法。
    public Iterator iter() { return null; }
    public Equipment(final String name) { this.name=name; }
}
```

大B：“抽象类Equipment就是Component定义，代表着组合体类的对象们,Equipment中定义几个共同的方法。”

```
public class Disk extends Equipment
{
    public Disk(String name) { super(name); }
    //定义Disk实价为1
    public double netPrice() { return 1.; }
    //定义了disk折扣价格是0.5 对折。
    public double discountPrice() { return .5; }
```

```
}
```

小A：“什么是Disk？”

大B：“Disk是组合体内的一个对象，或称一个部件，这个部件是个单独元素(Primitive)。还有一种可能是，一个部件也是一个组合体，就是说这个部件下面还有‘儿子’，这是树形结构中通常的情况，应该比较轻易理解。”

现在我们先要定义这个组合体：

```
abstract class CompositeEquipment extends Equipment
{
    private int i=0;
    //定义一个Vector 用来存放'儿子'
    private List equipment=new ArrayList();
    public CompositeEquipment(String name) { super(name); }
    public boolean add(Equipment equipment) {
        this.equipment.add(equipment);
        return true;
    }
    public double netPrice()
    {
        double netPrice=0.;
        Iterator iter=equipment.iterator();
        for(iter.hasNext())
            netPrice+=((Equipment)iter.next()).netPrice();
        return netPrice;
    }
    public double discountPrice()
    {
```

```

double discountPrice=0.;
Iterator iter=equipment.iterator();
for(iter.hasNext())
discountPrice+=((Equipment)iter.next()).discountPrice();
return discountPrice;
}
//注重这里，这里就提供用于访问自己组合体内的部件方法。
//上面disk 之所以没有，是因为Disk是个单独(Primitive)的元素.
public Iterator iter()
{
return equipment.iterator()
{
//重载Iterator方法
public boolean hasNext() { return i

```

大B：“上面CompositeEquipment继续了Equipment,同时为自己里面的对象们提供了外部访问的方法,重载Iterator,Iterator是Java的Collection的一个接口,是Iterator模式的实现。”

5.5 适用场景

大B：“当需求中是体现部分与整体层次的结构时,以及希望用户可以忽略组合对象与单个对象的不同时,统一地使用组合结构中的所有对象时,就应该考虑用组合模式了。明白吧?”

小A：“嗯！是的，明白。”

5.6 树

大B：“经常使用Control，你会发现Control有Controls的属性，而Controls集合包含的还是一个Control，类似的还有XmlNode。他们都有一个共有的特性，数据结构都是树行结构。”

小A：“什么是树形模式呢？”

大B：“树(Tree)是 $n(n \geq 0)$ 个结点的有限集 T ， T 为空时称为空树，否则它满足如下两个条件：1、有且仅有一个特定的称为根(Root)的结点；2、其余的结点可分为 $m(m \geq 0)$ 个互不相交的子集 T_1, T_2, \dots, T_m ，其中每个子集本身又是一棵树，并称其为根的子树(SubTree)。”

大B：“上面给出的递归定义刻画了树的固有特性：一棵非空树是由若干棵子树构成的，而子树又可由若干棵更小的子树构成。而这里的子树可以是叶子也可以是分支。先看下一幅图，如图5-2所示，里面的套娃就是一个套着一个的。”



图5-2 套娃

这样一堆娃娃，一个大的套一个小的，小的里面还可以套更小的，所以其组织结构为：

Top toy

-Toy

--toy

---toy

----toy

```
abstract class CompositeEquipment extends Equipment
{
private int i=0;
//定义一个Vector 用来存放'儿子'
private List equipment=new ArrayList();
public CompositeEquipment(String name) { super(name); }
public boolean add(Equipment equipment) {
this.equipment.add(equipment);
return true;
}
public double netPrice()
{
double netPrice=0.;
Iterator iter=equipment.iterator();
for(iter.hasNext())
netPrice+=((Equipment)iter.next()).netPrice();
}
```

```

return netPrice;
}
public double discountPrice()
{
double discountPrice=0.;
Iterator iter=equipment.iterator();
for(iter.hasNext())
discountPrice+=((Equipment)iter.next()).discountPrice();
return discountPrice;
}
//注重这里，这里就提供用于访问自己组合体内的部件方法。
//上面disk 之所以没有，是因为Disk是个单独(Primitive)的元素.
public Iterator iter()
{
return equipment.iterator()
{
//重载Iterator方法
public boolean hasNext() { return i

```

```

abstract class CompositeEquipment extends Equipment
{
private int i=0;
//定义一个Vector 用来存放'儿子'
private List equipment=new ArrayList();
public CompositeEquipment(String name) { super(name); }
public boolean add(Equipment equipment) {
this.equipment.add(equipment);
return true;
}
public double netPrice()
{

```

```

double netPrice=0.;
Iterator iter=equipment.iterator();
for(iter.hasNext())
netPrice+=((Equipment)iter.next()).netPrice();
return netPrice;
}

public double discountPrice()
{
double discountPrice=0.;
Iterator iter=equipment.iterator();
for(iter.hasNext())
discountPrice+=((Equipment)iter.next()).discountPrice();
return discountPrice;
}

//注重这里，这里就提供用于访问自己组合体内的部件方法。
//上面disk 之所以没有，是因为Disk是个单独(Primitive)的元素.
public Iterator iter()
{
return equipment.iterator()
}

//重载Iterator方法
public boolean hasNext() { return i

```

大B：“如果用程序来描述图5-2套娃，用设计模式的组合模式(Composite)是一个不错的主意。组合模式在GOF中定义为：组合(Composite)模式将对象以树形结构组织起来，以达成‘部分 - 整体’的层次结构，使得客户端对单个对象和组合对象的使用具有一致性。”

类图如图5-3所示：

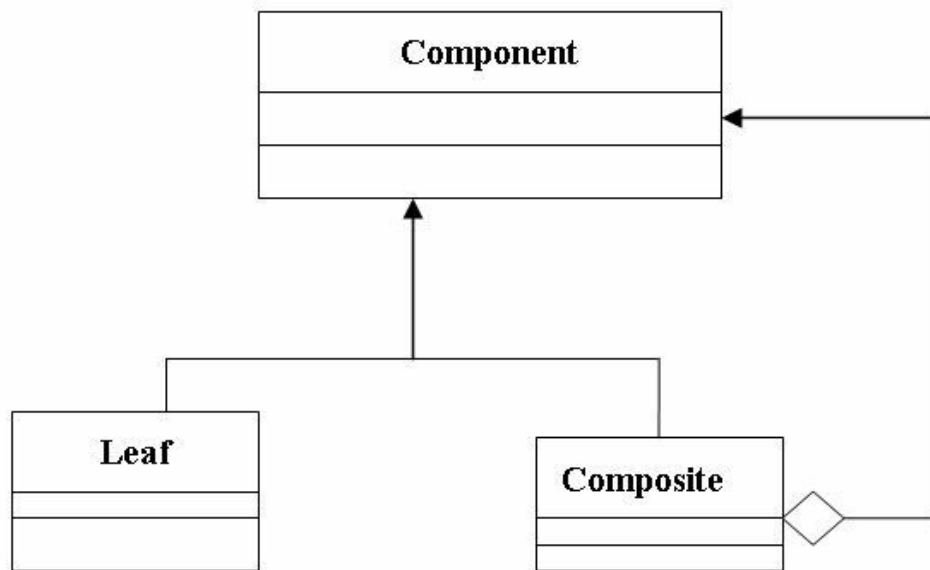


图5-3 类图

大B：“可以说，组合模式是比较简单易学的设计模式，我按照其定义和规则，实现一个论坛主题，帖子的组合关系。论坛中，一个主题可以包括很多帖子，一个帖子还可以包括很多回复。”

关系是：

Thread

-- Thread || Message

---- Thread || Message

下面是实现文件：

```
using System;
```

```
using System.Collections.Generic;
using System.Text;
namespace CompositeStudy
{
    public interface IThread
    {
        void Add(IThread thread);
        void Remove(IThread thread);
        void RenderContent();
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
namespace CompositeStudy
{
    public abstract class AbstractThread : IThread
    {
        bool _isTop;
        public bool IsTop
        {
            get
            {
                return _isTop;
            }
            set
            {
                _isTop = value;
            }
        }
    }
}
```

```
List list = new List();  
public List Children  
{  
    get  
    {  
        return list;  
    }  
    set  
    {  
        list = value;  
    }  
}  
string content = "";  
public string Content  
{  
    get  
    {  
        return content;  
    }  
    set  
    {  
        content = value;  
    }  
}  
public void Add(IThread thread)  
{  
    list.Add(thread);  
}  
public void Remove(IThread thread)  
{  
    list.Remove(thread);  
}
```

```

public abstract void RenderContent();
}
}

```

```

using System;
using System.Collections.Generic;
using System.Text;
namespace CompositeStudy
{
    public class Thread : AbstractThread
    {
        public override void RenderContent()
        {
            //输出自己的Content
            Console.WriteLine("Thread:"+this.Content);
            foreach (IThread t in Children)
            {
                t.RenderContent();
            }
        }
    }
}

```

```

using System;
using System.Collections.Generic;
using System.Text;
namespace CompositeStudy
{
    public class Message:AbstractThread

```



```

{
public override void RenderContent()
{
Console.WriteLine("Message:" + this.Content);
foreach (IThread t in Children)
{
t.RenderContent();
}
}
}
}
}

```

工厂类为：

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Data;
namespace CompositeStudy
{
/**////
/// 工厂类
///
/// 工厂类
public class ThreadFactory
{
DataTable table = new DataTable();
public ThreadFactory()
{
table.Columns.Add("content");
table.Columns.Add("IsTop");
}
}
}

```

```
table.Columns.Add("IsMessage");
table.Columns.Add("ID");
table.Columns.Add("ParentID");
DataRow row = table.NewRow();
row["content"] = "test";
row["IsTop"] = false;
row["IsMessage"] = false;
row["ID"] = 1;
row["ParentID"] = 0;
table.Rows.Add(row);
row = table.NewRow();
row["content"] = "test1";
row["IsTop"] = true;
row["IsMessage"] = false;
row["ID"] = 0;
row["ParentID"] = -1;
table.Rows.Add(row);
row = table.NewRow();
row["content"] = "test2";
row["IsTop"] = false;
row["IsMessage"] = true;
row["ID"] = 2;
row["ParentID"] = 0;
table.Rows.Add(row);
row = table.NewRow();
row["content"] = "test3";
row["IsTop"] = false;
row["IsMessage"] = true;
row["ID"] = 3;
row["ParentID"] = 0;
table.Rows.Add(row);
}
```

```
public List GetTopThreads()
{
    List list = new List();
    DataRow[] rows = table.Select("IsTop = true");
    foreach (DataRow row in rows)
    {
        Thread t = new Thread();
        t.Content = row["content"].ToString();
        t.IsTop = true;
        DataRow[] cs = table.Select("ParentID="+Convert.ToInt32(row["ID"]));
        foreach (DataRow r in cs)
        {
            if (Convert.ToBoolean(r["IsMessage"]))
            {
                Message m = new Message();
                m.Content = r["content"].ToString();
                m.IsTop = false;
                t.Add(m);
            }
            else
            {
                Thread tt = new Thread();
                tt.Content = r["content"].ToString();
                tt.IsTop = false;
                t.Add(tt);
            }
        }
        list.Add(t);
    }
    return list;
}
```

```
}
```

客户端调用方法为：

```
using System;
using System.Collections.Generic;
using System.Text;
namespace CompositeStudy
{
    class Program
    {
        static void Main(string[] args)
        {
            ThreadFactory factory = new ThreadFactory();
            List threads = factory.GetTopThreads();
            foreach(IThread t in threads)
            {
                t.RenderContent();
            }
            Console.Read();
        }
    }
}
```

类关系图如图5-4所示：

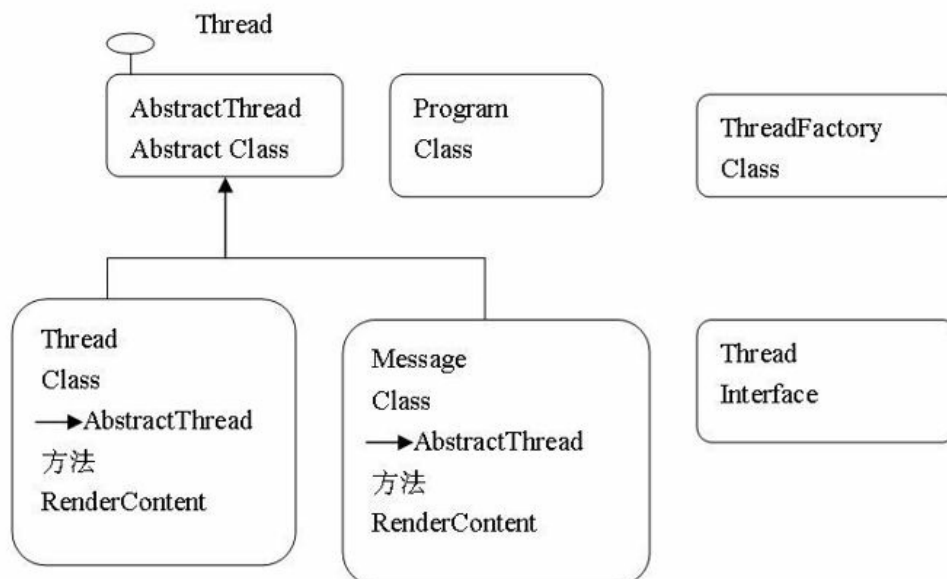


图5-4 类关系图

输结果为：

5.6 组合模式的优点

大B：“说说组合模式的优点吧。”

小A：“组合模式定义了包含基本对象和组合对象的类层次结构。基本对象可以被组合成更复杂的组合对象，而这个组合对象又可以被组合，这样不断地递归下去，客户代码中，任何用到基本对象的地方都可以使用组合对象了。用户不用关心到底是处理一个叶节点还是处理一个组合组件，也就是用不着为定义组合而写一些选择判断语句了，简单地说就是组合模式让客户可以一致地使用组合结构和单个对象。”

大B：“嗯，你知道什么是透明方式，什么是安全方式，及他们的好处。”

小A：“透明方式也就是说在Component中声明所有用来管理子对象的方法中，其中包括Add、Remove等。这样实现Component接口的所有子类都具备了Add和Remove。这样做的好处就是叶节点和枝节点对于外界没有区别，它们具备完全一致的行为接口。但问题也很明显，因为Leaf类本身不具备Add()、Remove()方法的功能，所以实现它是没有意义的。

安全方式也就是在Component接口中不去声明Add和Remove方法，那么子类的Leaf也不需要去实现它，而是在Composite声明所有用来管理子类对象的方法。不过由于不透明，所以树叶和树枝将不具有相同的接口，客户端的调用需要做相应的判断，带来了不便。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第六章 蜡笔与毛笔——桥接模式

6.1 蜡笔与毛笔

时间：12月21日 地点：大B房间 人物：大B，小A

大B：“师弟，你小的时候玩过蜡笔画画吗？”

小A：“有啊。小时候经常都有玩哩。怎么啦？”

大B：“记得那红红绿绿的蜡笔一大盒特别漂亮。”

小A：“嗯。特漂亮！”

大B：“我们那时经常用蜡笔根据想象描绘出格式图样。”

小A：“对啊！特有成就感。还可以用毛笔画国画哩！”

大B：“就是！毛笔下的国画更是工笔写意，各展风采。”

小A：“是啊！小时候觉得特好玩。”

大B：“嘿嘿！对啊！那今天我们就从蜡笔与毛笔说起吧。”

小A：“好啊。”

大B：“设想要绘制一幅图画，蓝天、白云、绿树、小鸟，如果画面尺寸很大，那么用蜡笔绘制就会遇到点麻烦。毕竟细细的蜡笔要涂出一片蓝天，是有些麻烦。如果有可能，最好有套大号蜡笔，粗粗的蜡笔很快能涂抹完成。至于色彩吗，最好每种颜色来支粗的，除了蓝天还有绿地呢。”

小A：“那得要好多蜡笔哩！”

大B：“是啊！这样，如果一套12种颜色的蜡笔，我们需要两套24支，同种颜色的一粗一细。”

小A：“呵呵，画还没画，开始做梦了：要是再有一套中号蜡笔就更好了，这样，不多不少总共36支蜡笔。”

如图6-1蜡笔所示

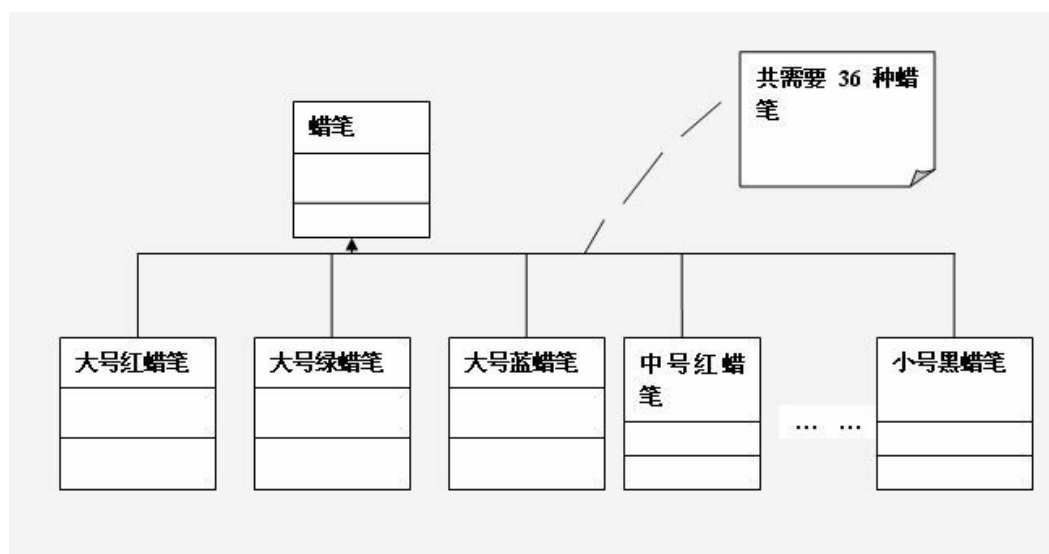


图6-1 蜡笔

大B：“那当然好。再看看毛笔这一边，居然如此简陋：一套水彩12色，外加大中小三支毛笔。你可别小瞧这‘简陋’的组合，画蓝天用大毛笔，画小鸟用小毛

笔，各具特色。如图6-2毛笔所示”

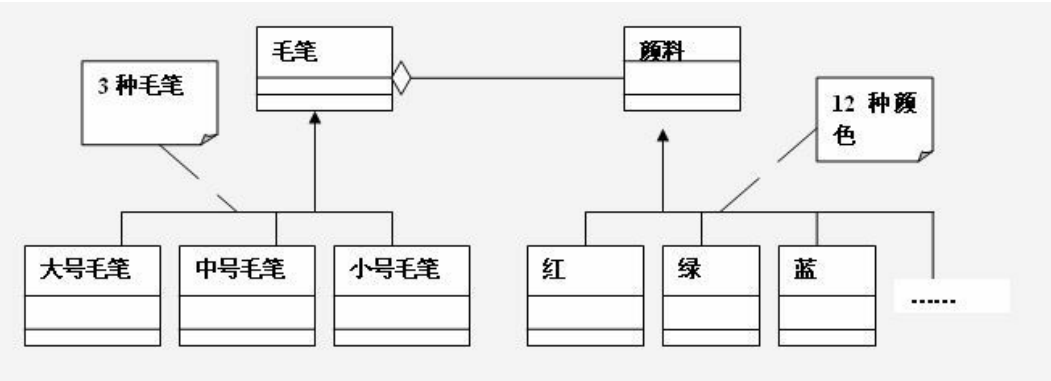


图6-2 毛笔

小A：“呵呵！我好像已经看出你今天想要说的模式了。”

大B：“不错！我今天要说的就是Bridge模式。”

小A：“还真被我看出来了哩！”

大B：“为了一幅画，我们需要准备36支型号不同的蜡笔，而改用毛笔三支就够了，当然还要搭配上12种颜料。通过Bridge模式，我们把乘法运算 $3 \times 12 = 36$ 改为了加法运算 $3 + 12 = 15$ ，这一改进可不小。”

小A：“那么我们这里蜡笔和毛笔到底有什么区别呢？”

大B：“实际上，蜡笔和毛笔的关键一个区别就在于笔和颜色是否能够分离。桥梁模式的用意是‘将抽象化(Abstraction)与实现化(Implementation)脱耦，使得二者可以独立地变化’。关键就在于能否脱耦。蜡笔的颜色和蜡笔本身是分不开的，所以就造成必须使用36支色彩、大小各异的蜡笔来绘制图画。而毛笔与颜料能够很好的脱耦，各自独立变化，便简化了操作。在这里，抽象层面的概念是：‘毛笔用颜料作画’，而在实现时，毛笔有大中小三号，颜料有红绿蓝等12种，于是便可出现 3×12 种组合。每个参与者（毛笔与颜料）都可以在自己的自由度上随意转

换。

蜡笔由于无法将笔与颜色分离，造成笔与颜色两个自由度无法单独变化，使得只有创建36种对象才能完成任务。Bridge模式将继承关系转换为组合关系，从而降低了系统间的耦合，减少了代码编写量。但这仅仅是Bridge模式带来的众多好处的一部分，更多层面的内容。”

小A：“那用代码怎么去表示啊？”

大B：“我写给你看一下，你应该就可以明白了。”

本文代码附于此处：

```
using System;
abstract class Brush
{ protected Color c;
  public abstract void Paint();
  public void SetColor(Color c)
  { this.c = c; }
}
class BigBrush : Brush
{ public override void Paint()
{ Console.WriteLine("Using big brush and color {0} painting ", c.color); }
}
class SmallBrush : Brush
{ public override void Paint()
{ Console.WriteLine("Using small brush and color {0} painting ", c.color); }
}
class Color
{ public string color;
```

```
}  
class Red : Color  
{ public Red()  
{ this.color = "red"; }  
}  
class Blue : Color  
{ public Blue()  
{ this.color = "blue"; }  
}  
class Green : Color  
{ public Green()  
{ this.color = "green"; }  
}  
class Client  
{ public static void Main()  
{   Brush b = new BigBrush();  
b.SetColor(new Red());  
b.Paint();  
b.SetColor(new Blue());  
b.Paint();  
b.SetColor(new Green());  
b.Paint();  
b = new SmallBrush();  
b.SetColor(new Red());  
b.Paint();  
b.SetColor(new Blue());  
b.Paint();  
b.SetColor(new Green());  
b.Paint();  
}  
}
```

6.2 桥接模式

大B：“聊了那么多，现在你来说说什么是桥接模式吧！”

小A：“桥接器模式（BridgePattern）又称为桥梁模式，它主要用意是为了实现抽象部分与实现部分脱耦，使它们各自可以独立地变化。”

大B：“在开发过程中大家通常会遇到一个对象有两个变化的维度，而且这两个维度变化地非常巨烈，这种变化导致了纵横交错的结果，使对象的设计变得困难，并且在对象数量上和可扩展性上都带来了很大的麻烦。此时我们应当把这两个变化比较巨烈的维度拆离，然后用组合的方式把它们结合在一起。这就是桥接器模式的思想。”

下面是它的结构图，如图6-3所示

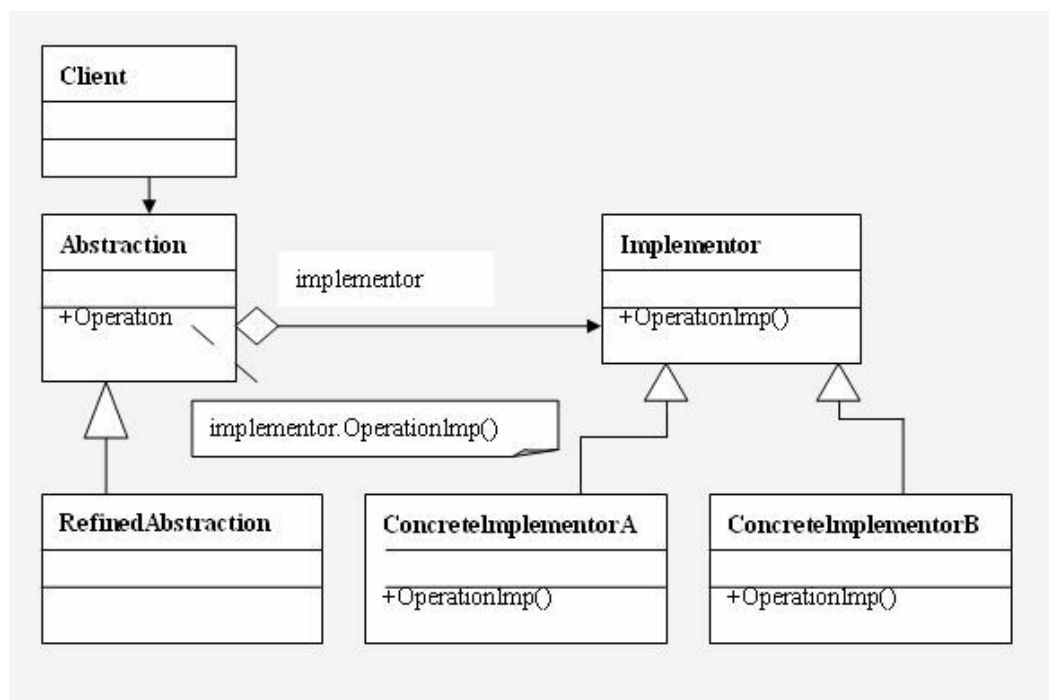


图6-3 Bridge模式结构图

小A：“桥接模式的主要特点有哪些啊？”

大B：“1、分离接口及其实现部分，这里实现了Abstraction和Implementor的分离，有助于降低对实现部分的依赖性，从而产生更好的结构化系统。2、提高了可扩充性，可以独立的对Abstraction和Implementor层次结构进行扩充。”

6.3 与适配器有什么不同

小A：“很多时候经常容易把桥接模式和适配器模式弄混。那什么时候用桥接，什么时候用适配器呢？有哪些共同点，又有哪些不同点哩？”

大B：“共同点：桥接和适配器都是让两个东西配合工作不同点：出发点不同。
适配器：改变已有的两个接口，让他们相容。 桥接模式：分离抽象化和实现，使两者的接口可以不同，目的是分离。所以说，如果你拿到两个已有模块，想让他们同时工作，那么你使用的适配器。如果你还什么都没有，但是想分开实现，那么桥接是一个选择。桥接是先有桥，才有两端的東西，适配是先有两边的东西，才有适配器，桥接是在桥好了之后，两边的东西还可以变化。例如游戏手柄，就象个桥，它把你的任何操作转化成指令。虽然，你可以任何操作组合，但是你的操作脱不开上下左右，a，b，选择，确定。”

小A：“为什么啊？”

大B：“JRE本身就是一个就是一个很好的桥，先写好在linux上执行的JRE，再写好可以在windows下执行的JRE，这样无论什么样的Java程序，只要配和相应的Jre就能在Linux或者Windows上运行。两个Jre并没有限定你写什么样的程序，但

要求你必须用Java来写。”

6.4 适用情况

小A：“师兄，桥梁模式适应在什么时候使用？”

大B：“在以下的情况下应当使用桥梁模式：1、如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的联系。2、设计要求实现化角色的任何改变不应当影响客户端，或者说实现化角色的改变对客户端是完全透明的。3、一个构件有多于一个的抽象化角色和实现化角色，系统需要它们之间进行动态耦合。虽然在系统中使用继承是没有问题的，但是由于抽象化角色和具体化角色需要独立变化，设计要求需要独立管理这两者。桥梁模式是一个非常有用的模式，也非常复杂，它很好的符合了开放-封闭原则和优先使用对象，而不是继承这两个面向对象原则。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第三部分

责任型模式

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第七章 击鼓传花——责任型模式

7.1 击鼓传花

时间：12月23日 地点：大B房间 人物：大B，小A

大B：“你听过什么是击鼓传花吗？”

小A：“听过，是一种很热闹的游戏，具体是什么游戏我就知道了。”

大B：“是的。击鼓传花是一种热闹而又紧张的饮酒游戏。在酒宴上宾客依次坐定位置，由一人击鼓，击鼓的地方与传花的地方是分开的，以示公正。开始击鼓时，花束就开始依次传递，鼓声一落，假如花束在某人手中，则该人就得饮酒。假比说，贾母、贾赦、贾政、贾宝玉和贾环是五个参加击鼓传花游戏的传花者，他们组成一个环链。击鼓者将花传给贾母，开始传花游戏。花由贾母传给贾赦，由贾赦传给贾政，由贾政传给贾宝玉，又由贾宝玉传给贾环，由贾环传回给贾母，如此往复（见图7-1击鼓传花）。当鼓声停止时，手中有花的人就得执行酒令。”小A：“这样的啊？听起来很好玩喔。”

大B：“呵呵.....是啊.....”

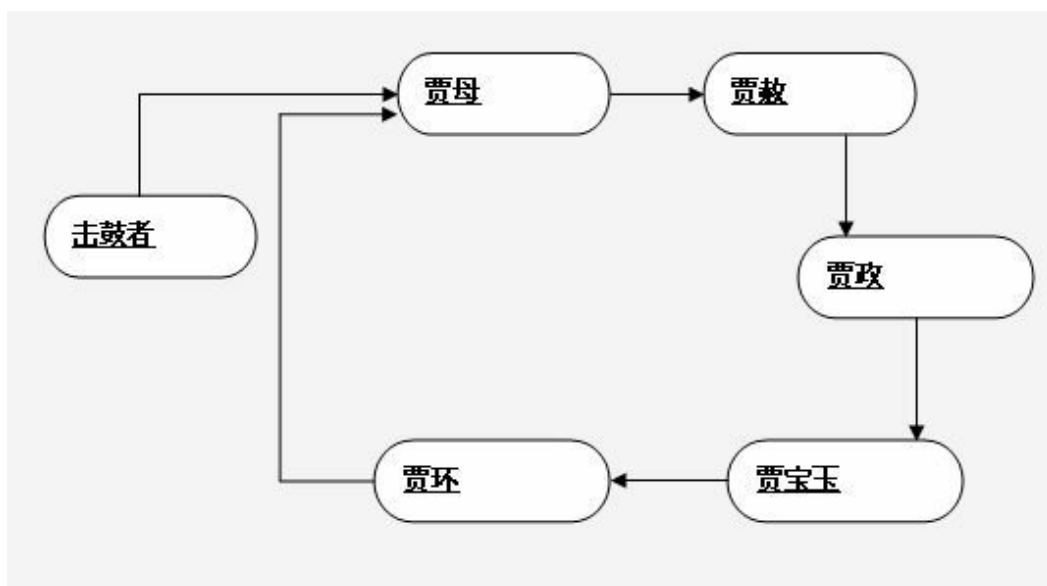


图7-1 击鼓传花

7.2 责任型模式

大B：“击鼓传花便是责任链模式的应用。在责任链模式里，很多的对象由每一个对象对其下家的引用而联接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织链和分配责任。”

小A：“哇，这就是责任链模式啊？听起来好像有点复杂喔。”

大B：“其实很简单的。责任链可能是一条直线、一个环链甚至一个树结构的一部分。”

小A：“这样一说好像很简单，但听起来好像很复杂似的。”

小A：“那你刚才说的红楼梦中击鼓传花的故事不是就是符合责任链模式吗？”

大B：“显然，击鼓传花符合责任链模式的定义。参加游戏的人是一个个的具体处理者对象，击鼓的人便是客户端对象。花代表酒令，是传向处理者的请求，每一个参加游戏的人在接到传来的花时，可选择的行为只有两个：一是将花向下传；一是执行酒令——喝酒。一个人不能既执行酒令，又向下家传花；当某一个人执行了酒令之后，游戏重新开始。击鼓的人并不知道最终是由哪一个做游戏的人执行酒令，当然执行酒令的人必然是做游戏的人们中的一个。”

击鼓传花的类图结构如图7-2击鼓传花系统的类图定义：

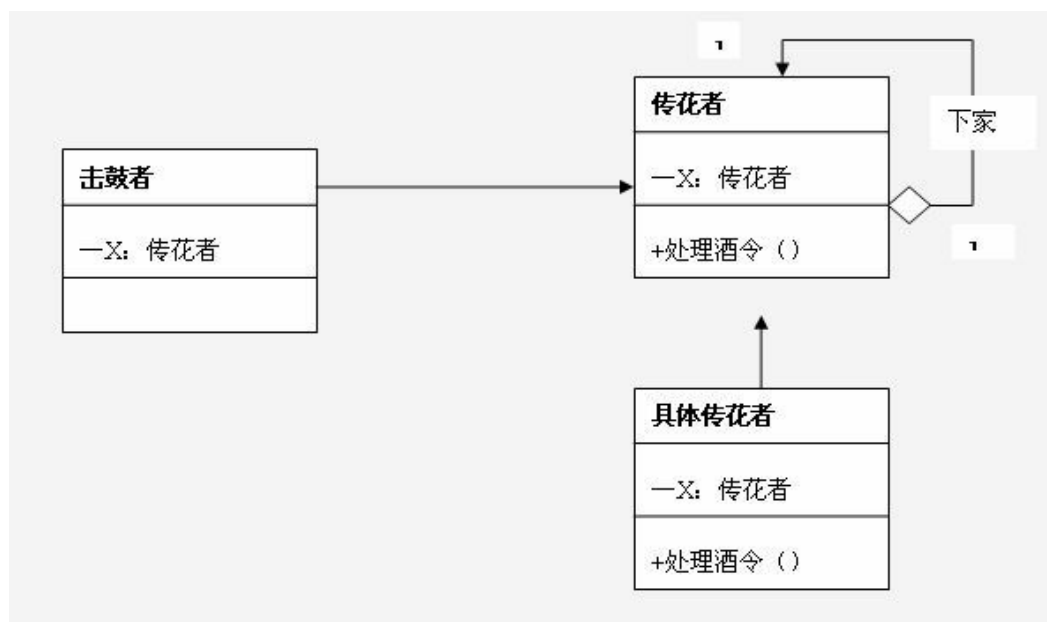


图7-2 击鼓传花系统的类图定义

大B：“单独考虑击鼓传花系统，那么像贾母、贾赦、贾政、贾宝玉和贾环等传花者均应当是‘具体传花者’的对象，而不应当是单独的类；但是责任链模式往往是建立在现有系统的基础之上的，因此链的结构和组成不由责任链模式本身决定。”

小A：“喔。是吗？”

大B：“系统的分析在《红楼梦》第七十五回里生动地描述了贾府里的一场击鼓传花游戏：‘贾母坐下，左垂首贾赦，贾珍，贾琏，贾蓉，右垂首贾政，宝玉，贾环，贾兰，团团围坐。贾母便命折一枝桂花来，命一媳妇在屏后击鼓传花。若花到谁手中，饮酒一杯。于是先从贾母起，次贾赦，一一接过。鼓声两转，恰恰在贾政手中住了，只得饮了酒。’这场游戏接着又把花传到了宝玉和贾赦手里，接着又传到了在贾环手里...如果用一个对象系统描述贾府，那么贾母、贾赦、贾政、贾宝玉和贾环等等就应当分别由一个个具体类代表，而这场击鼓传花游戏的类图，按照责任链模式。”

应当如图7-3红楼梦中的击鼓传花游戏的示意性类图所示：

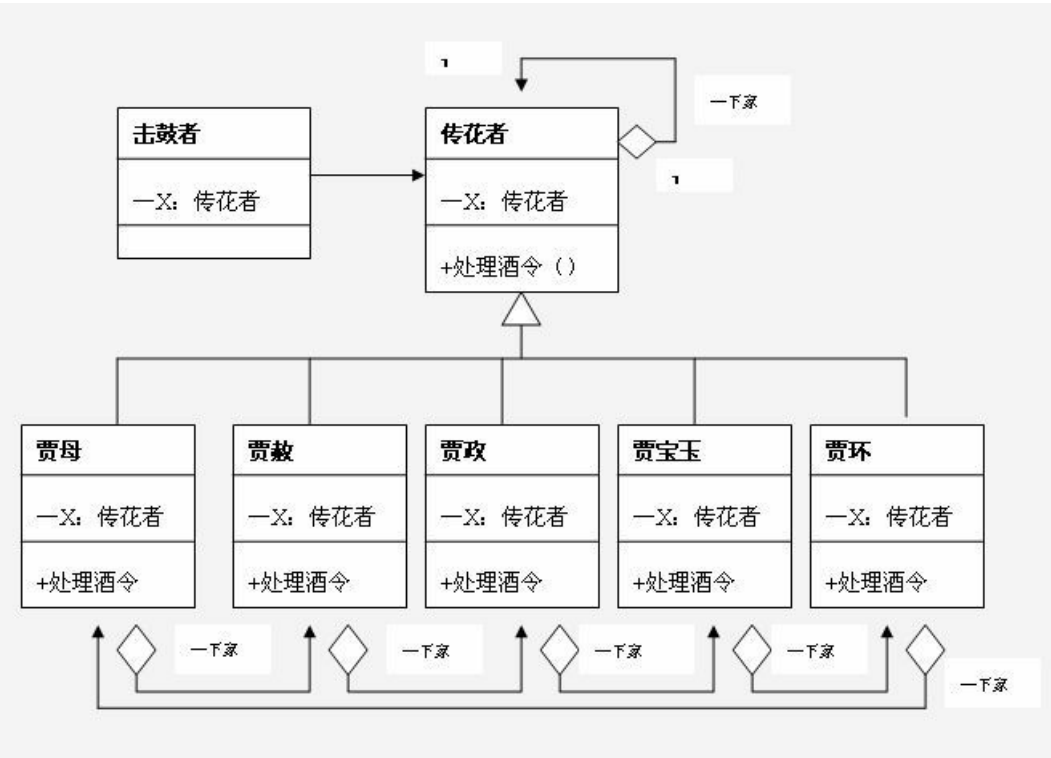


图7-3 红楼梦中的击鼓传花游戏的示意性类图

大B：“换言之，在击鼓传花游戏里面，有下面的几种角色：抽象传花者，或Handler角色、定义出参加游戏的传花人要遵守的规则，也就是一个处理请求的接口和对下家的引用；具体传花者，或ConcreteHandler角色、每一个传花者都知道

下家是谁，要么执行酒令，要么把花向下传。这个角色由贾母、贾赦、贾珍、贾琏、贾蓉、贾政、宝玉、贾环、贾兰等扮演。 击鼓人，或Client角色、即行酒令的击鼓之人。《红楼梦》没有给出此人的具体姓名，只是说由‘一媳妇’扮演。”

如图7-4贾府这次击鼓传花的示意性对象图：

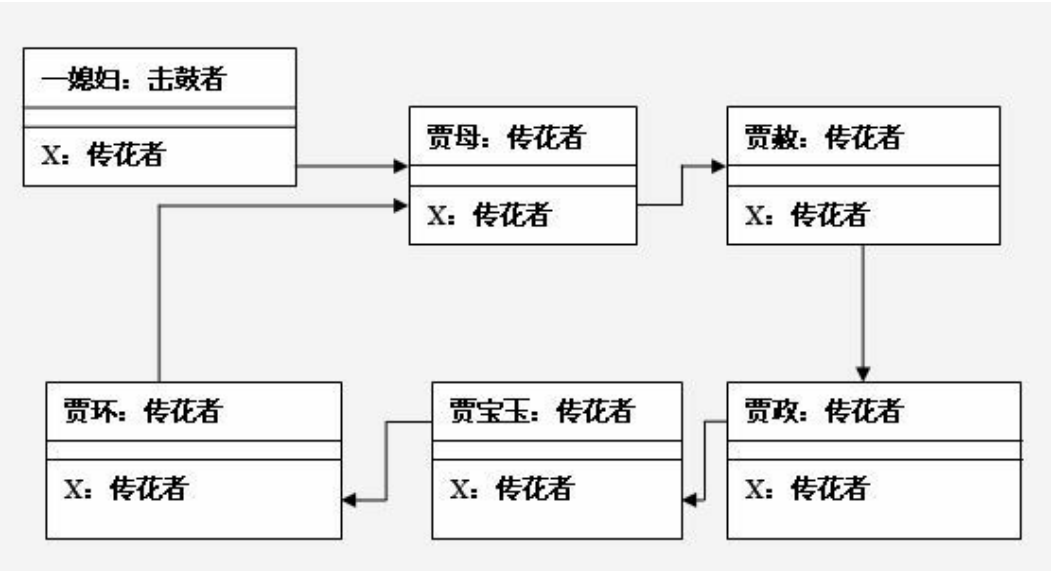


图7-4 贾府这次击鼓传花的示意性对象图

大B：“可以看出，击鼓传花游戏满足责任链模式的定义，是纯的责任链模式的例子。图7-5击鼓传花的类图完全符合责任链模式的定义的定义的类图给出了这些类的具体接口设计。不难看出，DrumBeater（击鼓者）、Player（传花者）、JiaMu（贾母）、JiaShe（贾赦）、JiaZheng（贾政）、JiaBaoYu（宝玉）、JiaHuan（贾环）等组成这个系统。”

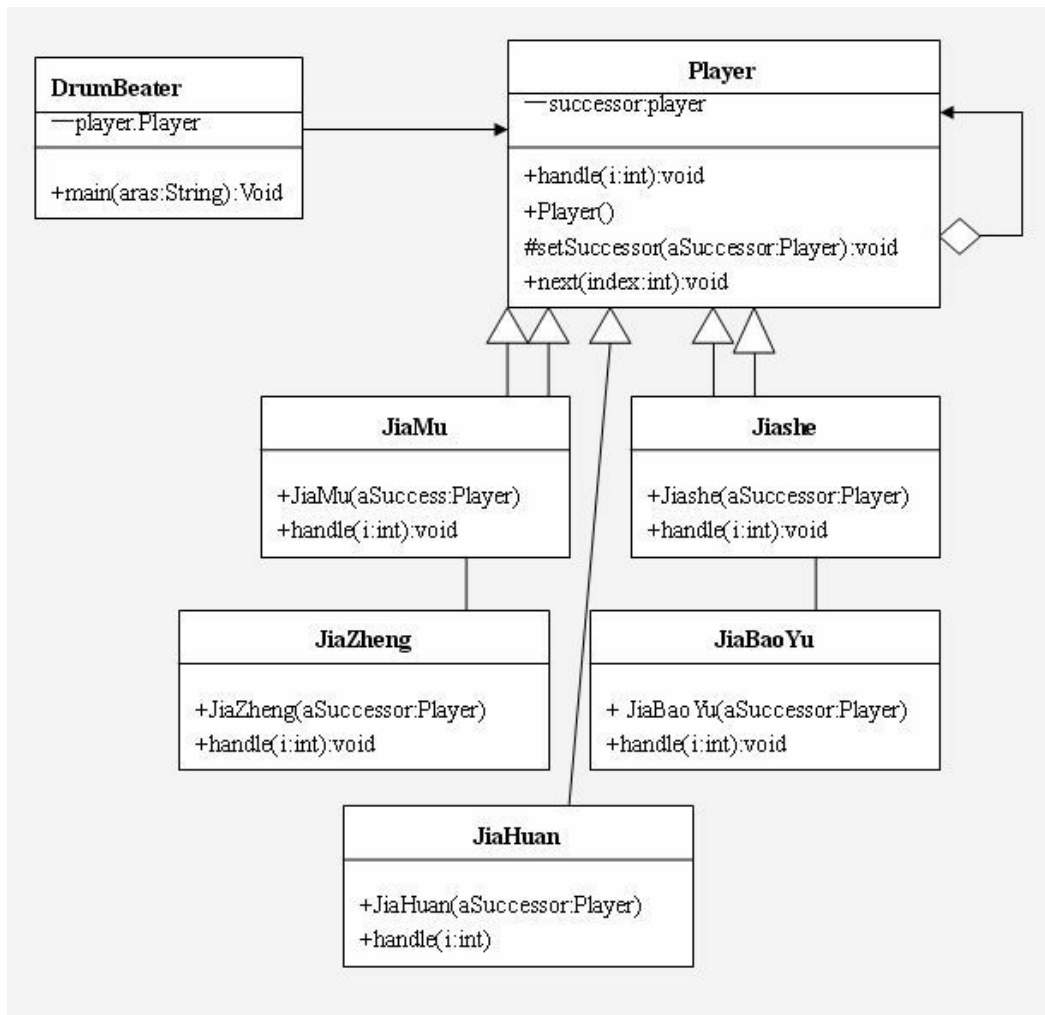


图7-5 击鼓传花的类图完全符合责任链模式的定义

下面是客户端类DrumBeater的源代码：

```

public class DrumBeater
{
    private static Player player;
    static public void main(String[] args)
    {
        player = new JiaMu( new JiaShe( new JiaZheng( new JiaBaoYu(new JiaHuan(null))));
        player.handle(4);
    }
}

```

代码清单1、DrumBeater的源代码。

```
abstract class Player
{
    abstract public void handle(int i);
    private Player successor;
    public Player() { successor = null;
    }
    protected void setSuccessor(Player aSuccessor)
    {
        successor = aSuccessor;
    }
    public void next(int index)
    {
        if( successor != null )
        {
            successor.handle(index);
        }
        else
        {
            System.out.println("rogram terminated.");
        }
    }
}
```

代码清单2、抽象传花者Play类的源代码。

大B：“抽象类Player给出了两个方法的实现，以格式setSuccessor()，另一个是next()。前者用来设置一个传花者对象的下家，后者用来将酒令传给下家。Player类给出了一个抽象方法handle()，代表执行酒令。”

下面的这些具体传花者类将给出handle()方法的实现。

```
class JiaMu extends Player
{
public JiaMu(Player aSuccessor)
{
this.setSuccessor(aSuccessor);
}
public void handle(int i)
{
if( i == 1 )
{
System.out.println("Jia Mu gotta drink!");
}
else
{
System.out.println("Jia Mu passed!"); next(i);
}
}
}
```

代码清单3、代表贾母的JiaMu类的源代码。

```
class JiaShe extends Player
{
public JiaShe(Player aSuccessor)
{
this.setSuccessor(aSuccessor);
}
public void handle(int i)
```

```
{
if( i == 2 )
{
System.out.println("Jia She gotta drink!");
}
else
{
System.out.println("Jia She passed!");
next(i);
}
}
}
```

代码清单4、代表贾赦的JiaShe类的源代码。

```
class JiaZheng extends Player
{
public JiaZheng(Player aSuccessor)
{
this.setSuccessor(aSuccessor);
}
public void handle(int i)
{
if( i == 3 )
{
System.out.println("Jia Zheng gotta drink!");
}
else
{
System.out.println("Jia Zheng passed!");
next(i);
}
```



```
}  
}  
}
```

代码清单5、代表贾政的JiaZheng类的源代码。

```
class JiaBaoYu extends Player  
{  
    public JiaBaoYu(Player aSuccessor)  
    {  
        this.setSuccessor(aSuccessor);  
    }  
    public void handle(int i)  
    {  
        if( i == 4 )  
        {  
            System.out.println("Jia Bao Yu gotta drink!");  
        }  
        else  
        {  
            System.out.println("Jia Bao Yu passed!");  
            next(i);  
        }  
    }  
}
```

代码清单6、代表贾宝玉的JiaBaoYu类的源代码。

```
class JiaHuan extends Player
```

```

{
public JiaHuan(Player aSuccessor)
{
this.setSuccessor(aSuccessor);
}
public void handle(int i)
{
if( i == 5 )
{
System.out.println("Jia Huan gotta drink!");
}
else
{
System.out.println("Jia Huan passed!");
next(i);
}
}
}

```

代码清单7、代表贾环的JiaHuan类的源代码。

大B：“可以看出，DrumBeater设定了责任链的成员和他们的顺序：责任链由贾母开始到贾环，周而复始。JiaMu类、JiaShe类、JiaZheng类、JiaBaoYu类与JiaHuan类均是抽象传花者Player类的子类。实现的DrumBeater类在把请求传给贾母时，实际上指定了由4号传花者处理酒令。虽然DrumBeater并不知道哪一个传花者类持有号码4，但是这个号码在本系统一开始就写死的。这当然并不符合击鼓传花游戏的精神，因为这个游戏实际上要求有两个同时进行的过程：击鼓过程和传花过程。击鼓应当是定时停止的，当击鼓停止时，执行酒令者就确定了。”

7.3 责任型模式涉及到哪些角色

大B：“责任链模式是一种对象的行为模式。你知道他都涉及到哪些角色吗？”

小A：“嘿嘿！不知道喔！”

大B：“没关系，我告诉你。所涉及到的角色如下：第一、抽象处理者角色、定义出一个处理请求的接口；假如需要，接口可以定义出一个方法，以返回对下家的引用。”

如图7-6抽象处理者角色给出了一个示意性的类图：

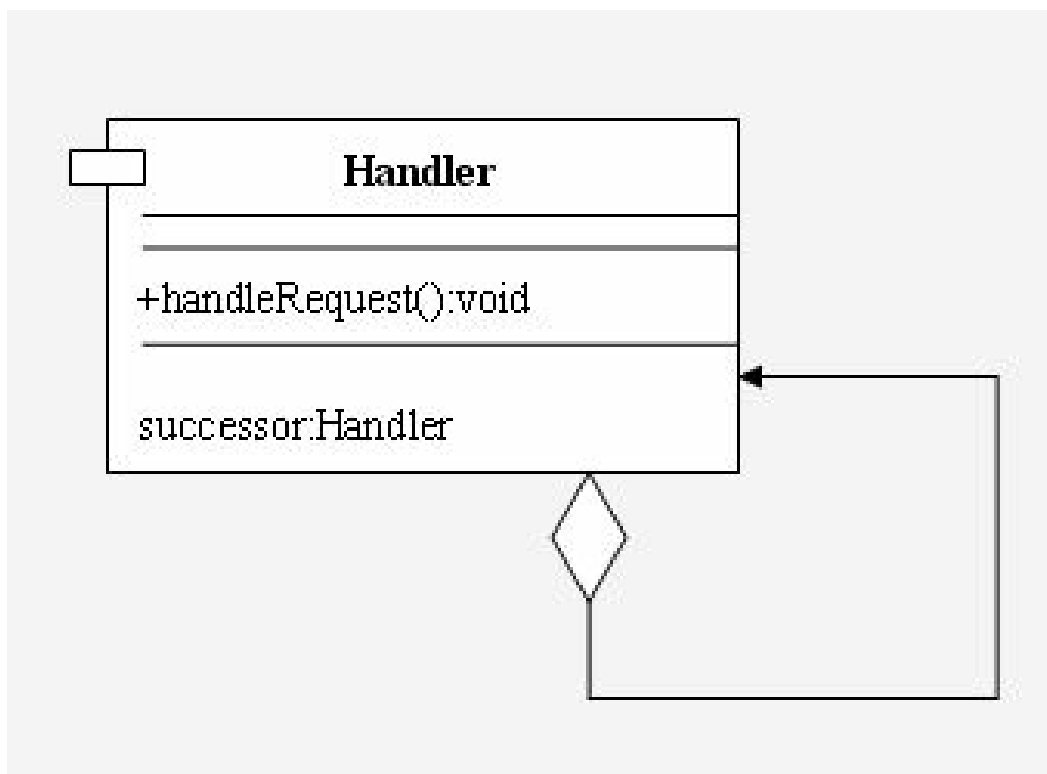


图7-6 抽象处理者角色

大B：“在图中的积累关系给出了具体子类对下家的引用，抽象方法 `handleRequest()` 规范了子类处理请求的操作。第二、具体处理者

(ConcreteHandler) 角色、处理接到请求后，可以选择将请求处理掉，或者将请求传给下家。”

如图7-7具体处理者角色给出了一个示意性的类图。

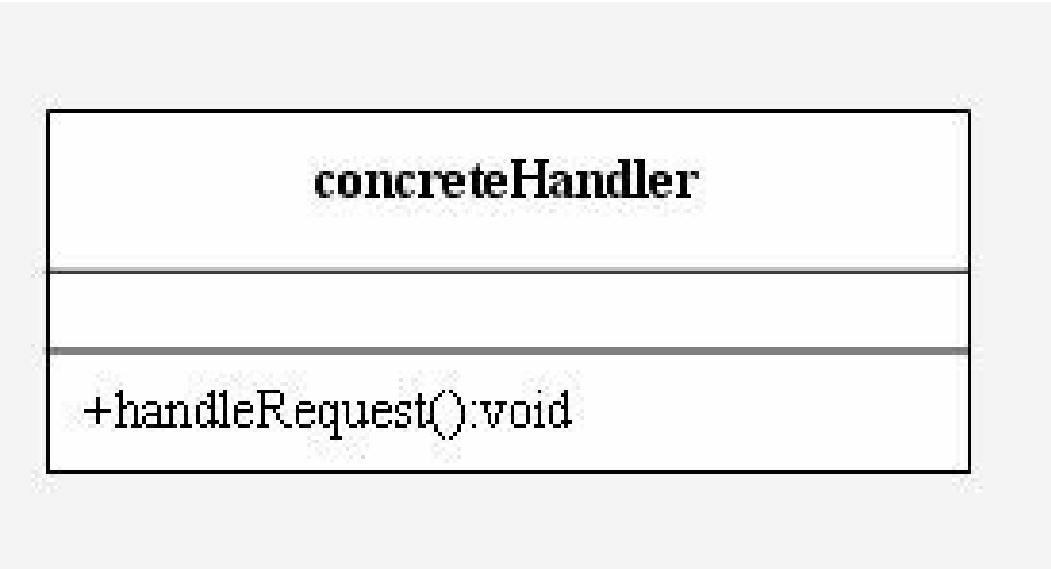
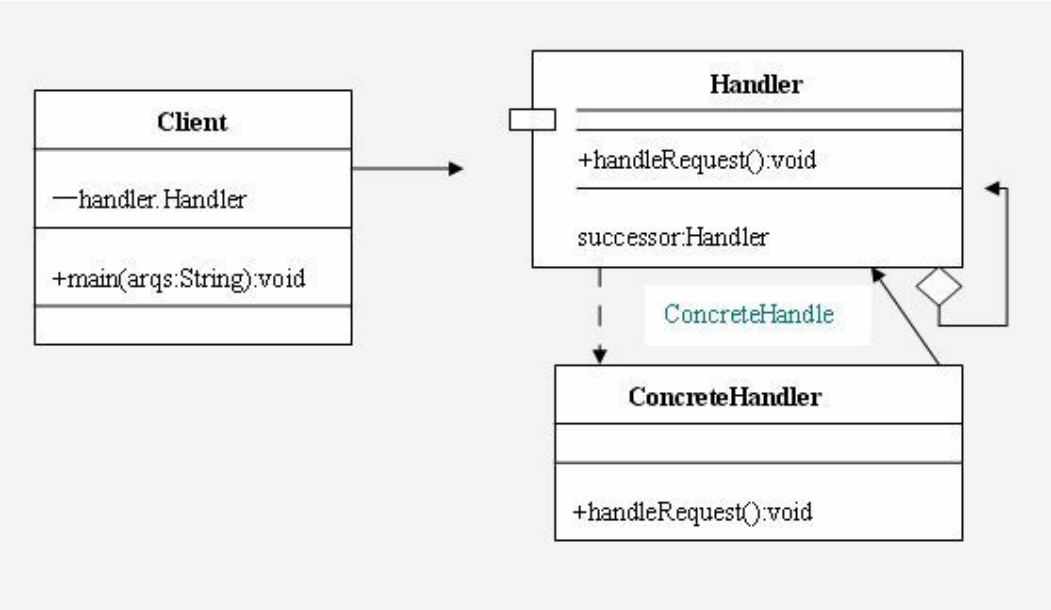


图7-7 具体处理者角色

大B：“图7-7中的示意性的具体处理者ConcreteHandler类只有 `handleRequest()` 一个方法。”

责任链模式的静态类结构可见图7-8责任链模式的类图定义。



大B：“在图中还给出了一个客户端，以便可以更清楚地看到责任链模式是怎样应用的。你能写出抽象处理者的示意性源代码吗？”

小A：“写不出来喔！还请师兄教我。”

大B：“好。”

抽象处理者的示意性源代码：

```
public class Handler
{
    public void handleRequest()
    {
        if (successor != null)
        {
            successor.handleRequest();
        }
        // Write your code here
    }
    public void setSuccessor(Handler successor)
    {
        this.successor = successor;
    }
    public Handler getSuccessor()
    {
        return successor;
    }
    private Handler successor;
}
```

代码清单8、抽象处理者的源代码。

具体处理者的示意性源代码：

```
public class ConcreteHandler extends Handler
{
    public void handleRequest()
    {
        if (getSuccessor() != null)
        {
            getSuccessor().handleRequest();
        }
        if (successor != null)
        {
            successor.handleRequest();
        }
        // Write your code here
    }
}
```

代码清单9、具体处理者的源代码。

客户端的源代码如下：

```
public class Client
{
    private Handler handler;
    public static void main(String[] args)
    {
        handler = new ConcreteHandler();
    }
}
```

```
//write your code here  
}  
}
```

代码清单10、客户端的源代码。

7.4 纯的与不纯的责任链模式

大B：“师弟，在责任链模式中还应该注意纯的与不纯的责任链模式。”

小A：“什么是纯的与不纯的责任链模式？”

大B：“一个纯的责任链模式要求一个具体的处理者对象只能在两个行为中选择一个：一是承担责任，二是把责任推给下家。不答应出现某一个具体处理者对象在承担了一部分责任后又把责任向下传的情况。在一个纯的责任链模式里面，一个请求必须被某一个处理者对象所接受；在一个不纯的责任链模式里面，一个请求可以最终不被任何接受端对象所接受。纯的责任链模式的实际例子很难找到，一般看到的例子均是不纯的责任链模式的实现。有些人认为不纯的责任链根本不是责任链模式，这也许是有道理的；但是在实际的系统里，纯的责任链很难找到；假如坚持责任链不纯便不是责任链模式，那么责任链模式便不会有太大的意义了。”

小A：“这回我可明白了。”

7.6 什么情况下使用责任链模式

小A：“在什么情况下使用责任链模式？”

大B：“第一、系统已经有一个由处理者对象组成的链。这个链可能由复合模式给出，第二、当有多于一个的处理者对象会处理一个请求，而且在事先并不知道到底由哪一个处理者对象处理一个请求。这个处理者对象是动态确定的。第三、当系统想发出一个请求给多个处理者对象中的某一个，但是不明显指定是哪一个处理者对象会处理此请求。第四、当处理一个请求的处理者对象集合需要动态地指定时。”

7.7 使用责任链模式的长处和短处

小A：“那使用责任链模式有哪些长处和短处？”

大B：“责任链模式减低了发出命令的对象和处理命令的对象之间的耦合，它允许与一个的处理者对象根据自己的逻辑来决定哪一个处理者最终处理这个命令。换言之，发出命令的对象只是把命令传给链结构的起始者，而不需要知道到底是链上的哪一个节点处理了这个命令。显然，这意味着在处理命令上，允许系统有更多的灵活性。哪一个对象最终处理一个命令可以因为由那些对象参加责任链、以及这些对象在责任链上的位置不同而有所不同。”

7.8 责任链模式的实现

大B：“首先是链结构的由来值得指出的是，责任链模式并不创建出责任链。责

任链的创建必须有系统的其它部分完成。责任链模式减低了请求的发送端和接收端之间的耦合，使多个对象都有机会处理这个请求。一个链可以是一条线，一个树，也可以是一个环。链的拓扑结构可以是单连通的或多连通的，责任链模式并不指定责任链的拓扑结构。但是责任链模式要求在同一个时间里，命令只可以被传给一个下家（或被处理掉）；而不可以传给多于一个下家。”

在下面的责任链是系统已有的树结构的一部分，责任链是一个树结构的一部分。图中有阴影的对象给出了一个可能的命令传播路径。

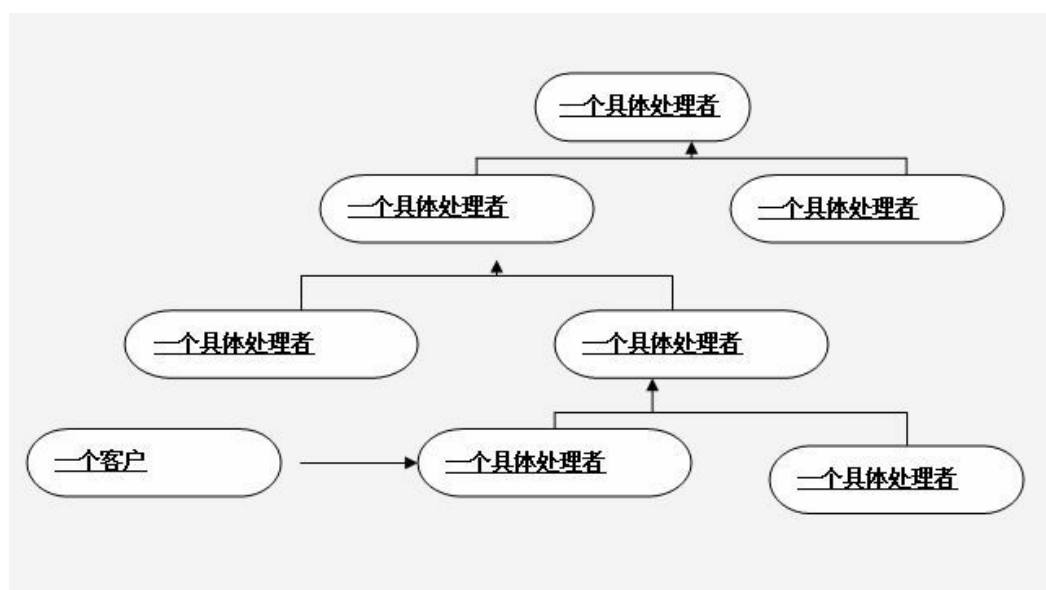


图7-9 责任链是系统已有的树结构的一部分

大B：“责任链的成员往往是一个更大的结构的一部分。比如我们刚才说的《红楼梦》中击鼓传花的游戏，所有的成员都是贾府的成员。如果责任链的成员不存在，那么为了使用责任链模式，就必须创建它们；责任链的具体处理者对象可以是同一个具体处理者类的实例。在Java的1.0版的AWT事件处理模型里，责任链便是视窗上的部件的容器等级结构。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质

电子书下载！！！！

第八章 购物车——单体模式

8.1 购物车

时间：12月24日 地点：XX超市 人物：大B和他的女朋友

今天是一年一度的平安夜，对于年轻男女而言，平安夜是一个狂欢的夜晚，对于大B和他的女朋友来说却是一个难得的享受幸福的时光。他们约好今天去超市买些食物，计划今天两个人一起动手做一顿丰盛的晚餐。

到了超市，大B去服务区推来购物车，我们都知道一个用户只能有一个购物车的。

大B：“今天平安夜，咱俩难得有时间一起做顿丰盛的晚餐。今晚就让我露一手好好地犒劳犒劳你吧！”

大B的女朋友：“呵呵！好啊！好久没见你亲自下厨了，今天你打算做什么好吃的呀？”

大B：“你想吃什么啊？”

大B的女朋友：“清蒸卢鱼，清蒸毛蟹，还有……”

大B：“呵呵.....好好好！今天都依你。”

大B的女朋友：“今天过节，有好多东西都在打特价，水果、蔬菜、肉类都好新鲜哩，我们买些回去好不好？”

大B：“好。”

转眼的功夫，大B和他的女朋友买了一大堆的东西，把整个购物车装得满满的。

大B：“我们这么多东西一会用购物车把东西推到超市停车场吧，这样就不用大包小包地往外提了。”

大B的女朋友：“嗯。好！”

8.2 单体模式

时间：12月24日 地点：大B房间 人物：大B，小A

小A：“什么是单体模式？”

大B：“对象只要利用自己的属性完成了自己的任务，那该对象就是承担了责任。除了维持了自身的一致性，该对象无需承担其他任何责任。如果该对象还承担着其他责任，而其他对象又依赖于该特定对象所承担的责任，我们就需要得到该特定对象。就像我和我的女朋友去超市购物使用的购物车一样。”

小A：“什么是单体模式的目的？”

大B：“将类的责任集中到唯一的单体对象中，确保该类只有一个实例，并且为该类提供一个全局访问点。这就是单体模式的目的。”

小A：“师兄，单体模式的难点是什么啊？”

大B：“单体模式的难点不在于单体模式的实现，而在于在系统中任何识别单体和保证单体的唯一性。”

8.3 单体模式的实现

小A：“师兄，单体模式的实现要怎么去实现？”

大B：“1、提供唯一的私有构造器，避免多个单体(Singleton)对象被创建，这也意味着该单体类不能有子类，那声明你的单例类为final是一个好主意，这样意图明确，并且让编译器去使用一些性能优化选项。如果有子类的话使用protected，protected的构造方法可以被其子类以及在同一个包中的其它类调用。私有构造器可以防止客户程序员通过除由我们提供的方法之外的任意方式来创建一个实例，如果不把构造器声明为private或protected，编译器会自动的创建一个public的构造函数。2、使用静态域(static field)来维护实例。将单体对象作为单体类的一个静态域实例化。使用保存唯一实例的static变量，其类型就是单例类型本身。需要的话使用final，使其不能够被重载。”

例如：`private static Runtime currentRuntime = new Runtime();`”

3、使用静态方法(Static Method)来监视实例的创建。

a、加载时实例化

例如：

```
public class Singleton {  
    private static final Singleton Singleton _instance = new Singleton();  
    private Singleton() {  
    }  
    public static synchronized Singleton getInstance() {  
        return Singleton _instance;  
    }  
}  
  
public class Singleton {  
    private static final Singleton Singleton _instance = new Singleton();  
    private Singleton() {  
    }  
    public static synchronized Singleton getInstance() {  
        return Singleton _instance;  
    }  
}
```

b、使用时实例化(惰性初始化)：这样做可以在运行时收集需要的信息来实例化单体对象，确保实例只有在需要时才被建立出来。

例如：

```
public class Singleton {  
    private static final Singleton Singleton _instance = null;  
    private Singleton() {
```

```

//使用运行时收集到的需要的信息,进行属性的初始化等操作。
}
public static synchronized Singleton getInstance() {
    if (Singleton _instance == null){
        Singleton _instance = new Singleton();
    }
    return Singleton _instance;
}
}

public class Singleton {
    private static final Singleton Singleton _instance = null;
    private Singleton() {
        //使用运行时收集到的需要的信息,进行属性的初始化等操作。
    }
    public static synchronized Singleton getInstance() {
        if (Singleton _instance == null){
            Singleton _instance = new Singleton();
        }
        return Singleton _instance;
    }
}
}

```

4、单体对象的成员变量(属性)：即单体对象的状态通过单例对象的初始化来实现成员变量的初始化。通过方法对单体对象的成员变量进行更新操作。

例如：

```

public class Singleton {
    private static final Singleton Singleton _instance = null;
    private Vector properties = null;

```

```

protected Singleton() {
//使用运行时收集到的需要的信息,进行属性的初始化等操作。
}
private static synchronized void syncInit() {
if (Singleton _instance == null) {
Singleton _instance = new Singleton();
}
}
public static Singleton getInstance() {
if (Singleton _instance == null){
syncInit();
}
return Singleton _instance;
}
public synchronized void updateProperties() {
// 更新属性的操作。
}
public Vector getProperties() {
return properties;
}
}
public class Singleton {
private static final Singleton Singleton _instance = null;
private Vector properties = null;
protected Singleton() {
//使用运行时收集到的需要的信息,进行属性的初始化等操作。
}
private static synchronized void syncInit() {
if (Singleton _instance == null) {
Singleton _instance = new Singleton();
}
}
}

```



```
public static Singleton getInstance() {  
    if (Singleton _instance == null){  
        syncInit();  
    }  
    return Singleton _instance;  
}  
  
public synchronized void updateProperties() {  
    // 更新属性的操作。  
}  
  
public Vector getProperties() {  
    return properties;  
}  
}
```

8.4 单体模式的一般方法

小A：“单体模式一般有哪些方法？”

大B：“单体模式主要作用是保证在Java应用程序中，一个Class只有一个实例存在。一般有三种方法，下面我就具体来说说这三种方法吧。”

1、定义一个类，它的构造函数为private的，所有方法为static的。如java.lang.Math其他类对它的引用全部是通过类名直接引用。

例如：

```
public final class Math {
```

```

/**
 * Don't let anyone instantiate this class.
 */
private Math() {}
public static int round(float a) {
    return (int)floor(a + 0.5f);
}
...
}

```

2、定义一个类，它的构造函数为private的，它有一个static的private的该类变量，在类初始化时实例化，通过一个public的getInstance方法获取对它的引用,继而调用其中的方法。

例如：

```

public class Runtime {
    private static Runtime currentRuntime = new Runtime();
    public static Runtime getRuntime() {
        return currentRuntime;
    }
    ...
}

```

3、定义一个类，它的构造函数为private的，它有一个static的private的boolean变量,用于表示是否有实例存在。

例如：

```

class PrintSpooler {
//this is a prototype for a printer-spooler class
//such that only one instance can ever exist
static boolean
instance_flag=false; //true if 1 instance
public PrintSpooler() throws SingletonException
{
if (instance_flag)
throw new SingletonException("Only one spooler allowed");
else
instance_flag = true; //set flag for 1 instance
System.out.println("spooler opened");
}
//-----
public void finalize()
{
instance_flag = false; //clear if destroyed
}
}

```

8.5 单体模式的不同表现形式

大B：“你知不知道单体模式有哪些不同表现形式？”

小A：“我知道。不同表现形式：1、饿汉式单体类：类被加载的时候将自己初始化。更加安全些。2、懒汉式单体类：在第一次被引用的时候将自己初始化。提高了效率。3、多例类(多例模式)：除了可以提供多个实例，其他和单体类没有区

别。”

大B：“不错，我再详细说给你听吧！”

单体模式的不同表现形式之：多例类(多例模式)

所谓多例 (Multiton Pattern) 实际上就是单例模式的自然推广。作为对象的创建模式，多例模式或多例类有以下的特点：

1、多例类可以有多个实例

2、多例类必须自己创建，管理自己的实例，并向外界提供自己的实例。这种允许有限个或无限个实例，并向整个JVM提供自己实例的类叫做多例类，这种模式叫做多例模式。

(1) 有上限多例模式 。有上限的多例类已经把实例的上限当作逻辑的一部分，并建造到了多例类的内部，这种多例模式叫做有上限多例模式。

```
import java.util.Random;
import java.util.Date;
public class Die
{
    private static Die die1 = new Die();
    private static Die die2 = new Die();
    /**
    *私有的构造函数保证外界无法直接将此类实例化
    *
    */
    private Die()
    {
```

```

}
/**
*工厂方法
*/
public static Die getInstance(int whichOne)
{
    if(whichOne==1)
    {
        return die1;
    }else{
        return die2;
    }
}
/**
*投骰子，返回一个在1 - 6之间的随机数
*/
public synchronized int dice()
{
    Date d = new Date();
    Random r = new Random(d.getTime());
    int value = r.nextInt();//获取随机数
    value = Math.abs(value);//获取随机数的绝对值
    value = value % 6;//对6取模
    value +=1;//由于 value的范围是0 - 5，所以value+1成为1-6
    return value;
}
}
/**
*测试的客户端，投掷骰子
*/
public class DieClient
{

```

```

private static Die die1,die2;

public static void main(String[] args)
{
    die1 = Die.getInstance(1);
    die2 = Die.getInstance(2);
    System.out.println(die1.dice());
    System.out.println(die2.dice());
}
}

import java.util.Random;
import java.util.Date;

public class Die
{
    private static Die die1 = new Die();
    private static Die die2 = new Die();
    /**
    *私有的构造函数保证外界无法直接将此类实例化
    *
    */
    private Die()
    {
    }
    /**
    *工厂方法
    */
    public static Die getInstance(int whichOne)
    {
        if(whichOne==1)
        {
            return die1;
        }else{
            return die2;
        }
    }
}

```

```

}
}
/**
 *投骰子，返回一个在1 - 6之间的随机数
 */
public synchronized int dice()
{
    Date d = new Date();
    Random r = new Random(d.getTime());
    int value = r.nextInt();//获取随机数
    value = Math.abs(value);//获取随机数的绝对值
    value = value % 6;//对6取模
    value +=1;//由于 value的范围是0 - 5，所以value+1成为1-6
    return value;
}
}
/**
 *测试的客户端，投掷骰子
 */
public class DieClient
{
    private static Die die1,die2;
    public static void main(String[] args)
    {
        die1 = Die.getInstance(1);
        die2 = Die.getInstance(2);
        System.out.println(die1.dice());
        System.out.println(die2.dice());
    }
}

```

(2) 无上限多例模式 。 例数目没有上限的多例模式叫无上限多例模式。

注意：由于没有上限的多例类对实例的数目是没有限制的，因此，虽然这种多例模式是单例模式的推广，但是这种多例类并不一定能够回到单例类。于事先不知道要创建多少个实例，因此，必然使用聚集管理所有的实例。

例子：购物车

```
import java.util.ArrayList;
import java.util.HashMap;
public class ShoppingCart {
    private ShoppingCart shoppingCart = null;
    //使用HashMap聚集管理所有的实例；
    private static HashMap instanse = new HashMap();
    //订单列表
    private ArrayList orderedItems = null;
    //更新器
    private int readCount = 0;
    /**
     *同单例类一样，私有的构造函数保证外界无法直接将此类实例化
     *
     */
    private ShoppingCart() {
        orderedItems = new ArrayList();
    }
    /**
     * 获取购物车，一个用户只能有一个购物车。有多少用户就有多少购物车。
     */
    public synchronized static ShoppingCart getInstance(String user) {
        ShoppingCart shoppingCart = instanse.get(user);
```



```

if (shoppingCart == null){
shoppingCart = new ShoppingCart();
instance.put(user, shoppingCart);
}
return shoppingCart;
}
/*
* 用户退出登陆的时候，通过外部调用将购物车移除。
* */

public synchronized void removeShoppingCart(String key){
instance.remove(key);
}
/*
* 获取购物车中订单列表 ( orderedItems )
* */

public ArrayList getOrderedItems() {
readIn();
readOut();
return orderedItems;
}
/*
* 管理订单。
* 如果是旧订单则更新其数量。
* 如果是新订单则添加到订单列表中。
* */

public void addItem(String itemId){
updateIn();
ItemOrder order;
for(int i=0; i= 0) {
orderedItems.remove(i);
} else {
order.setNumItems(numOrdered);

```

```

}
return;
}
}
ItemOrder newOrder = new ItemOrder(Catalog.getItem(itemId), 1);
orderedItems.add(newOrder);
}
private synchronized void updateIn() {
while (readCount > 0) {
try {
wait();
} catch (Exception e) {
}
}
}
private synchronized void readIn() {
readCount++;
}
private synchronized void readOut() {
readCount--;
notifyAll();
}
}

```

例子：购物车

```

import java.util.ArrayList;
import java.util.HashMap;
public class ShoppingCart {
private ShoppingCart shoppingCart = null;
//使用HashMap聚集管理所有的实例；

```

```

private static HashMap instance = new HashMap();
//订单列表
private ArrayList orderedItems = null;
//更新器
private int readCount = 0;
/**
 *同单例类一样，私有的构造函数保证外界无法直接将此类实例化
 *
 */
private ShoppingCart() {
orderedItems = new ArrayList();
}
/*
 * 获取购物车，一个用户只能有一个购物车。有多少用户就有多少购物车。
 * */
public synchronized static ShoppingCart getInstance(String user) {
ShoppingCart shoppingCart = instance.get(user);
if (shoppingCart == null){
shoppingCart = new ShoppingCart();
instance.put(user, shoppingCart);
}
return shoppingCart;
}
/*
 * 用户退出登陆的时候，通过外部调用将购物车移除。
 * */
public synchronized void removeShoppingCart(String key){
instance.remove(key);
}
/*
 * 获取购物车中订单列表 ( orderedItems )
 * */

```

```

public ArrayList getOrderedItems() {
    readIn();
    readOut();
    return orderedItems;
}
/*
 * 管理订单。
 * 如果是旧订单则更新其数量。
 * 如果是新订单则添加到订单列表中。
 * */
public void addItem(String itemId){
    updateIn();
    ItemOrder order;
    for(int i=0; i= 0) {
        orderedItems.remove(i);
    } else {
        order.setNumItems(numOrdered);
    }
    return;
}
}
ItemOrder newOrder = new ItemOrder(Catalog.getItem(itemId), 1);
orderedItems.add(newOrder);
}
private synchronized void updateIn() {
    while (readCount > 0) {
        try {
            wait();
        } catch (Exception e) {
        }
    }
}
}
}

```

```
private synchronized void readIn() {  
    readCount++;  
}  
private synchronized void readOut() {  
    readCount--;  
    notifyAll();  
}  
}
```

8.6 单体对象的同步(单体模式与多线程)

大B：“在多线程模式下，惰性初始化会使多个线程同时初始化该单体，造成一个JVM中多个单例类型的实例，如果这个单例类型的成员变量在运行过程中发生变化，会造成多个单例类型实例的不一致。”

小A：“那应该怎么办？”

大B：“加个同步修饰符：`public static synchronized Singleton getInstance()`。这样就保证了线程的安全性。这种处理方式虽然引入了同步代码，但是因为这段同步代码只会在最开始的时候执行一次或多次，所以对整个系统的性能不会有影响。”

小A：“在更新属性的时候，会造成属性的读写不一致。那应该怎么处理？”

大B：“1、读者/写者的处理方式。设置一个读计数器，每次读取信息前，将计数器加1，读完后将计数器减1。使用`notifyAll()`解除在该对象上调用`wait`的线

程阻塞状态。只有在读计数器为0时，才能更新数据，同时调用wait()方法要阻塞所有读属性的调用。

2、采用‘影子实例’的办法。具体说，就是在更新属性时，直接生成另一个单例对象实例，这个新生成的单例对象实例将从数据库，文件或程序中读取最新的信息；然后将这些信息直接赋值给旧单例对象的属性。”

小A：“嘿嘿！师兄，能不能举例来看一下啊？”

大B：“好的。”

例子：

```
public class GlobalConfig {
    private static GlobalConfig instance;
    private Vector properties = null;
    private boolean isUpdating = false;
    private int readCount = 0;
    private GlobalConfig() {
        //Load configuration information from DB or file
        //Set values for properties
    }
    private static synchronized void syncInit() {
        if (instance == null) {
            instance = new GlobalConfig();
        }
    }
    public static GlobalConfig getInstance() {
        if (instance==null) {
            syncInit();
        }
        return instance;
    }
}
```

```

}
public synchronized void update(String p_data) {
    syncUpdateIn();
    //Update properties
}
private synchronized void syncUpdateIn() {
    while (readCount > 0) {
        try {
            wait();
        } catch (Exception e) {
        }
    }
}
private synchronized void syncReadIn() {
    readCount++;
}
private synchronized void syncReadOut() {
    readCount--;
    notifyAll();
}
public Vector getProperties() {
    syncReadIn();
    //Process data
    syncReadOut();
    return properties;
}
}

public class GlobalConfig {
    private static GlobalConfig instance;
    private Vector properties = null;
    private boolean isUpdating = false;
    private int readCount = 0;

```

```

private GlobalConfig() {
//Load configuration information from DB or file
//Set values for properties
}

private static synchronized void syncInit() {
if (instance == null) {
instance = new GlobalConfig();
}
}

public static GlobalConfig getInstance() {
if (instance==null) {
syncInit();
}
return instance;
}

public synchronized void update(String p_data) {
syncUpdateIn();
//Update properties
}

private synchronized void syncUpdateIn() {
while (readCount > 0) {
try {
wait();
} catch (Exception e) {
}
}
}

private synchronized void syncReadIn() {
readCount++;
}

private synchronized void syncReadOut() {
readCount--;
}

```



```
notifyAll();  
}  
public Vector getProperties() {  
    syncReadIn();  
    //Process data  
    syncReadOut();  
    return properties;  
}  
}
```

8.7 识别单体模式

小A：“师兄，要如何去识别单体模式呢？”

大B：“1、区别工具类和单体类在于该类是否是有状态的。无状态化，提供工具性质的功能，那就是工具类。如果愿意的话，你可以将单体类分为有他状态和无状态。有状态单体类又称为可变单体类，常用作状态库维护系统的状态。无状态单体类又称为不变单体类，常用作提供工具性质方法的对象。2、是否承担了唯一的责任，并且是否提供了唯一的实例。”

8.8 单体模式和一个所有方法都是静态的工具类之间有什么区别

小A：“单体模式和一个所有方法都是静态的工具类之间有什么区别？”

大B：“1、当一个Class被Load的时候，静态工具类的所有状态都已经被初始化了，而单体模式则可以控制自己的初始化过程2、单体可以继承别的类或被别的类继承，而静态工具类则不能(其实也能，但一旦继承了一个有非静态方法或静态值的类以后，它就无法保证自己只拥有一个实例，或达到只有一个实例的效果)3、单体可以被扩展到‘双体’，‘三体’，等等。但静态工具类则丧失了这种可扩展性。般的无状态工具集合适合实现成静态工具类，而拥有丰富状态，但整个系统只允许有一个实例的类，适合实现成单体。”

8.9 与单体模式相关的模式

小A：“那哪些是与单体模式相关的模式？”

大B：“在抽象工厂模式中可以使用单体模式，将具体工厂类设计为单体类。建造模式可以使用单例模式，将具体类设计成单体类。”

8.10 单体模式的应用

小A：“应该怎么去应用单体模式哩？”

大B：“首先是要建立目录，数据库连接或 Socket 连接要受到一定的限制，必须保持同一时间只能有一个连接的存在等这样的单线程操作。使用Singleton的好

处还在于可以节省内存，因为它限制了实例的个数，有利于Java垃圾回收（garbage collection）。"

小A：“关于数据库连接应用单体模式的问题应该怎么去解决？”

大B：“关于把单例模式应用到数据库connection的问题较为复杂，如果是简单地把一个connection对象封存在单例对象中，那么在J2EE环境中这是错误的。如果数据库连接做成单例模式也就是说系统只会存在一个数据库连接实例，大家公用。实例不可以并发使用。因此存在排队，单例模式管理是需要排队的。资源有两种，一种需要排队，一种不需要排队，或者需要一种复杂的排队，譬如数据库就是复杂的排队问题，系统的排队是不可避免的，应该由数据库引擎自行解决。解决方案就是纪录的locking，而locking不应该由Java程序解决。尽量将排队的工作交给更低一层来做，这样可以获得更高的效率。但是在单用户系统中这并不是什么严重的问题，因为在某一个时刻只有一个用户在使用，唯一的问题就是系统可能需要几个connection，譬如两个、三个等，而不是一个。2EE服务器系统中单例模式可以用来管理一个数据库连接池（connection pool）。单例模式可以用来保存这样一个connection pool，在初始化的时候创建譬如100个connection对象，然后再需要的时候提供一个，用过之后返回到pool中。如果不是用单例模式的话，这个pool存在哪里，就是一个问题。最后可能只好存到Application对象中。”

小A：“那关于‘全局’变量的问题又应该如何去解决？”

大B：“使用单例模式来存放‘全局’变量是违背单例模式的用意的，单例模式只在有真正的‘单一实例’的需求时才可以使用。其次，一个设计得当的系统不应该有所谓的‘全局’变量的。这些变量应该放到他们所描述的实体所对应的类中去。将这些变量从他们所描述的实体类中抽出来，放到一个不相干的单体类中去，

使得这些变量产生错误的依赖关系和耦合关系。所以，如果需要的话，我们可以将一个承担了责任的类作为一个单体类来实现，而不仅仅是为了一个‘全局’变量。”

大B：“有时使用Singleton并不能达到Singleton的目的，如有多个Singleton对象同时被不同的类装入器装载；在EJB这样的分布式系统中使用也要注意这种情况，因为EJB是跨服务器，跨JVM的。总之：Singleton模式看起来简单，使用方法也很方便，但是真正用好，是非常不容易，需要对Java的类 线程内存等概念有相当的了解。如果你的应用基于容器，那么Singleton模式少用或者不用，可以使用相关替代技术。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第九章 放风者与偷窃者——观察者模式

9.1 放风者与偷窃者

时间：12月25日 地点：大B房间 人物：大B，小A

这天，小A又到大B家玩，闲聊时他们说起了警匪片。

大B：“你喜欢看警匪片吗？”

小A：“嘿嘿！很喜欢。”

大B：“那你还记得警匪片上，匪徒们是怎么配合实施犯罪的吗？”

小A：“我知道他们总是会有放风者，还有就是所谓的做案者。”

大B：“嗯。对！一个团伙在进行盗窃的时候，总有一两个人在门口把风——如果有什么风吹草动，则会立即通知里面的同伙紧急撤退。也许放风的人并不一定认识里面的每一个同伙；而在里面也许有新来的小弟不认识这个放风的。但是这没什么，这个影响不了他们之间的通讯，因为他们之间有早已商定好的暗号。”

小A：“是啊！那个情节是最精彩的时候，我最喜欢看了。”

大B：“看警匪片的时候最好看的就是那个片段了。”

9.2 观察者模式

大B：“你知道什么样的属于观察者模式吗？”

小A：“是不是有观察者和被观察者的就是属于观察者模式？”

大B：“上面提到的放风者、偷窃者之间的关系就是观察者模式在现实中的活生生的例子。你现在知道什么是观察者模式了吧？”

小A：“嘿嘿！还不能完全理解。”

大B：“观察者模式又名发布订阅模式。定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。”

大B：“当两个对象之间松耦合，它们依然可以交互，但是不太清楚彼此的细节，观察者模式提供了一种对象设计，让主题和观察者之间松耦合。”

小A：“为什么呢？”

大B：“1、关于观察者的一切，主题只知道观察者实现了某个接口，不需要知道观察者的具体类是谁2、任何时候我们都可以增加新的观察者，因为主题唯一依赖的东西是一个实现Observer接口的对象列表3、有新类型的观察者出现时，主题的代码不需要修改。只要在新类里实现观察者接口，然后注册为观察者即可4、我们可以独立的复用主题或观察者，因为2者并非紧耦合5、改变主题或观察者任何一方，

并不会影响另一方，只要他们之间的接口人被遵守。”

9.3 组成部分

小A：“那它的组成部分哩？是不是就是有观察者和被观察者的就是属于观察者模式？”

大B：“不是。1、抽象目标角色：目标角色知道它的观察者，可以有任意多个观察者观察同一个目标。并且提供注册和删除观察者对象的接口。目标角色往往由抽象类或者接口来实现。2、抽象观察者角色：为那些在目标发生改变时需要获得通知的对象定义一个更新接口。抽象观察者角色主要由抽象类或者接口来实现。3、具体目标角色：将有关状态存入各个具体观察者角色对象。当它的状态发生改变时，向它的各个观察者发出通知。4、具体观察者角色：存储有关状态，这些状态应与目标的状态保持一致。实现观察者角色的更新接口以使自身状态与目标的状态保持一致。在这个角色内也可以维护一个指向具体目标角色对象的引用。”

放上观察者模式的类图，如图9-1所示。这样能将关系清晰的表达出来。

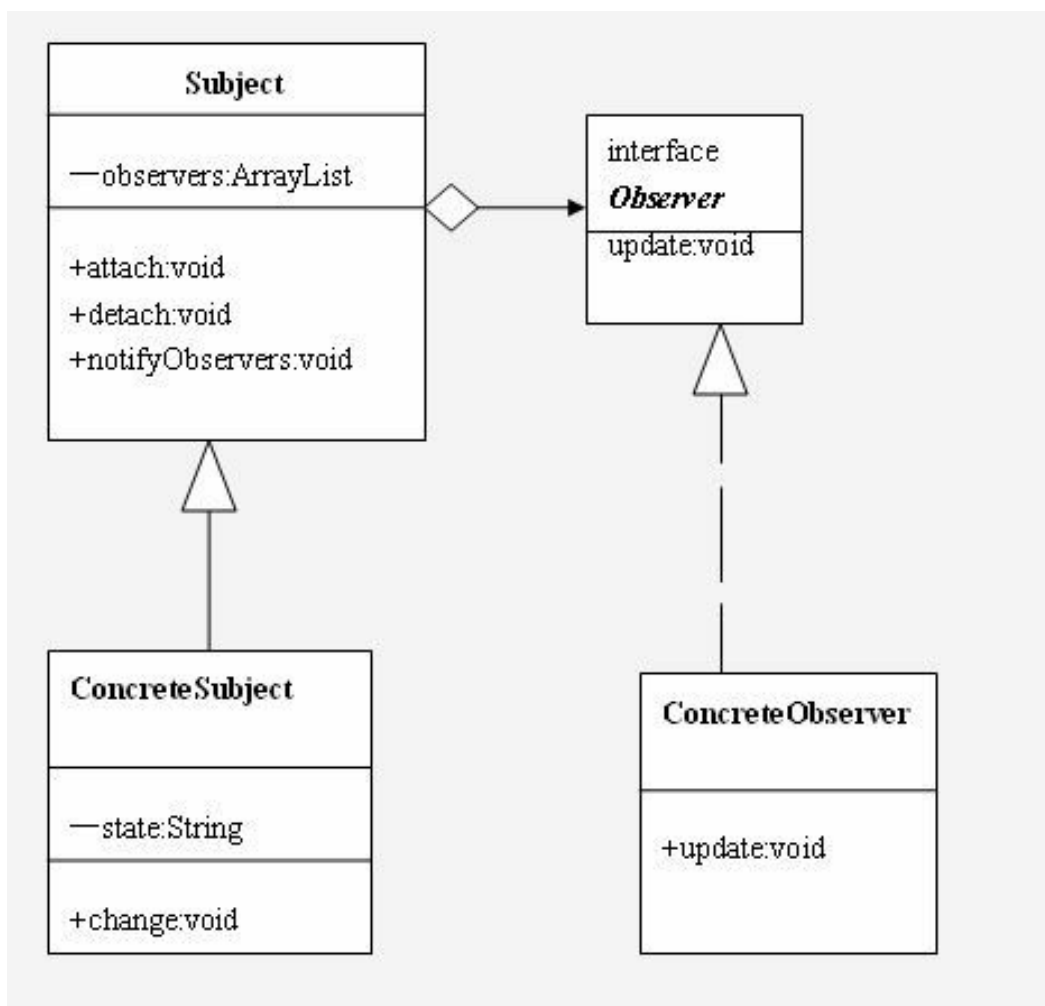


图9-1 观察者模式的类图

9.4 天气预报

大B：“用一个实际的例子来说明：日常生活中说起观察者，最常见的例子可能就是天气预报，在这里我们的观察对象是地球，而我们是通过发射气象卫星这个观察者来检测地球气象变化的。”

所以这个例子中涉及三个对象：

地球(Earth)：被观察对象

气象卫星(Satellite)：观察者

气象局(WeatherService)：客户端调用

被观察对象：地球 (Earth)

```
import java.util.Observable;

/**
 * 被观察对象：地球
 *
 * @version 1.0 create on 2006-5-18 9:42:45
 */
public class Earth extends Observable {
    private String weather = "晴朗";

    /**
     * @return Returns the weather.
     */
    public String getWeather(){
        return weather;
    }

    /**  */

    * @param weather
    * The weather to set.
    */
    public void setWeather(String weather){
        this.weather=weather;
        //设置变化点
        setChanged();
        notifyObservers(weather);
    }
}
```

[注意] 在需检测的对象前需要设置变化点setChanged()和通知观察者notifyObservers()，这两个函数是由Observable类实现的，封装了观察者模式实现的细节。

观察者：气象卫星(Satellite)

```
import java.util.Observable;
import java.util.Observer;

/**  */
* 观察对象：气象卫星
*
* @version 1.0 create on 2006-5-18 9:46:30
*/
public class Satellite implements Observer{
    private String weather;
    public void update(Observable obj,Object arg){
        weather = (String)arg;
        //捕获天气变化情况，反馈给检测者
        System.out.println("近期天气变化：" + weather);
    }
}
```

客户端调用：气象局(WeatherService)

```
/**  */
* 客户端调用：天气预报
*
* @version 1.0 create on 2006-5-18 9:57:19
*/
```

```
public class WeatherService{
/**    */
/**
 * @param args
 */

public static void main(String[] args){
Earth earth = new Earth();
Satellite satellite = new Satellite();
//发射气象卫星
earth.addObserver(satellite);
System.out.println("天气预报：");
System.out.println("-----");
earth.setWeather("台风 '珍珠' 逼近");
earth.setWeather("大到暴雨");
earth.setWeather("天气炎热");
}
}
```

[运行结果]

天气预报：

近期天气变化：台风‘珍珠’逼近

近期天气变化：大到暴雨

近期天气变化：天气炎热

9.5 观察者模式原理

小A：“观察者模式原理是什么？”

大B：“主题对象并不知道引用了哪些具体观察者对象类型，而只知道抽象观察者类型，这样具体主题对象可以动态地维护一系列的观察者对象的引用，并在需要的时候调用每一个观察者共有的更新方法。这是‘针对接口编程’的体现。”

9.6 现实生活中的观察者

在现实世界中，观察者模式对于那种由许多JavaScript程序员合作开发的大型程序特别有用。它可以提高API的灵活性，使并行开发的多个实现能够彼此独立地进行修改。

大B：“作为开发人员，你可以对自己的应用程序中什么是‘令人感兴趣的时刻’做出决定。你能监听的不再只是click、load、blur和mouseover等浏览器事件。在用户界面（rich UI）应用程序中，drag（拖动）、drop（拖放）、moved（移动）、complete（完成）和tabSwitch（标签切换）都可能是令人感兴趣的事件。它们都是在普通浏览器事件的基础上抽象出来的可观察事件，可由发布者对象向其监听者广播。”

9.7 事件监听器也是观察者

大B：“在DOM脚本编程环境中的高级事件模式中，事件监听器说到底就是一种内置的观察者。事件处理器（handler）与事件监听器（listener）并不是一回事。事件处理器说穿了就是一种把事件传给与其关联的函数的手段。而且在这种模型中一种事件只能指定一个回调方法。而在监听器模式中，一个事件可以与几个监听器关联。每个监听器都能独立于其他监听器而改变。”

小A：“师兄，能不能再细说一下？”

大B：“可以，我举个例子，你应该就可以明白了。打个比方，对San Francisco Chronicle这家报社来说，其订阅者Joe订没订New York Times都无所谓。同样，Joe也不在乎Lindsay是否也订了San Francisco Chronicle。每一方都只管处理自己的数据和相关的行为。”

例如，使用事件监听器，可以让多个函数响应同一个事件：

```
//example using listeners
Var element=$( 'example' );
Var fn1=function(e){
//handle click
};
Var fn2=function(e){
//do other stuff with click
};
addEvent(element,' click' ,fn1);
addEvent(element,' click' ,fn2);
```

但用事件处理器就办不到：

```
//example using handlers  
Var elemet=document.getElementById( 'b' );  
Var fn1=function(e){  
  //handle click  
};  
Var fn2=function(e){  
  //do other stuff with click  
};  
element.onclick=fn1;  
element.onclick=fn2;
```

大B：“在第一个例子中，由于使用的是事件监听器，所以click事件发生时fn1和fn2都会被调用。而第二个例子使用的是事件处理器，其中第二次对onclick赋值的结果是fn1被fn2取代，因此click事件发生时只会调用fn2，不会调用fn1。言归正传。监听器和观察者之间的共同之处显而易见。实际上它们互为同义语。它们都订阅特定的事件，然后等待事件的发生。事件发生时，订阅方的回调函数会得到通知。传给它们的参数是一个事件对象，其中包含着事件发生时间、事件类型和事件发源地等有用的信息。”

小A：“这样我就明白了。”

9.8 观察者模式的典型应用

大B：“我再和你说说观察者模式的典型应用。”

小A：“好。”

大B：“1、侦听事件驱动程序设计中的外部事件2、侦听/监视某个对象的状态变化3、发布者/订阅者(publisher/subscriber)模型中，当一个外部事件（新的产品，消息的出现等等）被触发时，通知邮件列表中的订阅者。”

9.9 观察者模式的优点

小A：“那观察者模式有什么优点呢？”

大B：“对象之间可以进行同步通信，可以同时通知一到多个关联对象，对象之间的关系以松耦合的形式组合，互不依赖。”

9.10 使用情况

大B：“讲了这么多，你现在能说说观察者模式的使用情况吗？”

小A：“1、当一个抽象模型有两个方面， 其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。2、当对一个对象的改变需要同时改变其它对象， 而不知道具体有多少对象有待改变。3、当一个对象必须通知其它对象，而它又不能假定其它对象是谁。”

大B：“换言之， 你不希望这些对象是紧密耦合的。其实观察者模式同前面讲过的桥梁、策略有着共同的使用环境：将变化独立封装起来，以达到最大的重用和解耦。观察者与后两者不同的地方在于，观察者模式中的目标和观察者的变化不是独

立的，而是有着某些联系。”

9.11 我推你拉

大B：“观察者模式在关于目标角色、观察者角色通信的具体实现中，有两个版本。”

小A：“哪两种版本呐？”

大B：“一种情况是目标角色在发生变化后，仅仅告诉观察者角色‘我变化了’；观察者角色如果想要知道具体的变化细节，则就要自己从目标角色的接口中得到。这种模式被很形象的称为：拉模式——就是说变化的信息是观察者角色主动从目标角色中‘拉’出来的。还有一种方法，那就是我目标角色‘服务一条龙’，通知你发生变化的同时，通过一个参数将变化的细节传递到观察者角色中去。这就是‘推模式’——管你要不要，先给你啦。这两种模式的使用，取决于系统设计时的需要。如果目标角色比较复杂，并且观察者角色进行更新时必须得到一些具体变化的信息，则‘推模式’比较合适。如果目标角色比较简单，则‘拉模式’就很合适啦。”

9.12 圣斗士星矢的状态模式和观察者模式

小A：“讲了那么多，师兄如果能给我举个例子那就更好了。”

大B：“那我给你举个圣斗士星矢的状态模式和观察者模式的例子吧！”

星矢：动画片《圣斗士星矢》的男猪蹄，超级小强，怎么打也打不死。

雅典娜：动画片《圣斗士星矢》的女猪蹄，自称女神，手下有88个男人为他卖命。

状态模式：为了方便的控制状态的变化，避免一堆IF/ELSE，以及状态规则改变的时避免代码改动的混乱。

观察者模式：一个被观察者一动，多个观察者跟着动，经常用于界面UI。

话说星矢和很强的某斗士甲对打，雅典娜在一边看，星矢总是挨揍，每次挨揍完之后星矢的状态总是会发生一些变化：

正常--挨打--濒死--挨打--小宇宙爆发--挨打--濒死--挨打--女神护体--挨打(星矢无敌了，打也没用，战斗结束)--正常

以上状态转变用状态模式来表现，一个Saiya类代表星矢，一个SaiyaState代表他的状态，SaiyaState下面有多个子类，分别代表星矢的多种状态，如正常NORMAL、濒死DYING、小宇宙爆发UNIVERSE、女神护体GODDESS，即把状态抽象成对象，在每种状态里面实现被打的时候所需要更改的状态，这样就避免了每次被打都要进行一次IF/ELSE的判断。

Java代码

```
public abstract class SaiyaState {  
    protected Saiya saiya;
```

```

public SaiyaState(Saiya saiya) {
    this.saiya = saiya;
}
public String status(){
    String name=getClass().getName();
    return name.substring(name.lastIndexOf(".")+1);
}
//星矢被打了
public abstract void hit();
}
public abstract class SaiyaState {
    protected Saiya saiya;
    public SaiyaState(Saiya saiya) {
        this.saiya = saiya;
    }
    public String status(){
        String name=getClass().getName();
        return name.substring(name.lastIndexOf(".")+1);
    }
    //星矢被打了
    public abstract void hit();
}

```

在每种状态里面实现被打的时候所需要更改的状态，例如小宇宙爆发状态下被打

Java代码

```

public class UniverseState extends SaiyaState {
    /**

```

```

* @param saiya
*/
public UniverseState(Saiya saiya) {
    super(saiya);
}
/* 小宇宙爆发状态被打进入濒死状态
*
*/
public void hit() {
    saiya.setState( saiya.DYING);
}
}

public class UniverseState extends SaiyaState {
/**
* @param saiya
*/
public UniverseState(Saiya saiya) {
    super(saiya);
}
/* 小宇宙爆发状态被打进入濒死状态
*
*/
public void hit() {
    saiya.setState( saiya.DYING);
}
}

```

雅典娜在一边看，星矢每次被打她都要给星矢加油，她是个观察者，星矢是被观察者，这里星矢实现java.util.Observable，每次被打hit就notifyObservers，雅典娜就加油。

Java代码

```
public class Athena implements Observer {  
    /* 我是雅典娜 我是观察者  
    *  
    */  
    public void update(Observable arg0, Object arg1) {  
        System.out.println("雅典娜说：星矢加油啊!!!");  
    }  
}  
  
public class Athena implements Observer {  
    /* 我是雅典娜 我是观察者  
    *  
    */  
    public void update(Observable arg0, Object arg1) {  
        System.out.println("雅典娜说：星矢加油啊!!!");  
    }  
}
```

总的来看这个过程就是这样子：

Java代码

```
public class StateMain {  
    public static void main(String[] args) {  
        Saiya saiya = new Saiya();  
        Observer athena = new Athena();  
        saiya.addObserver(athena);  
        System.out.println("星矢最初的状态是：" + saiya.status());  
        for (int i = 0; i < 5; i++) {
```

```

System.out.println("星矢被揍了" + (i + 1) + "次");
saiya.hit();
System.out.println("星矢现在的状态是：" + saiya.status());
}
}
}

public class StateMain {
    public static void main(String[] args) {
        Saiya saiya = new Saiya();
        Observer athena = new Athena();
        saiya.addObserver(athena);
        System.out.println("星矢最初的状态是：" + saiya.status());
        for (int i = 0; i < 5; i++) {
            System.out.println("星矢被揍了" + (i + 1) + "次");
            saiya.hit();
            System.out.println("星矢现在的状态是：" + saiya.status());
        }
    }
}

```

结果星矢在雅典娜的帮助下，有惊无险的战胜了很强的某斗士甲：

Java代码

星矢最初的状态是：NormalState

星矢被揍了1次

雅典娜说：星矢加油啊!!!

星矢现在的状态是：DyingState

星矢被揍了2次

雅典娜说：星矢加油啊!!!

星矢现在的状态是：UniverseState

星矢被揍了3次

雅典娜说：星矢加油啊!!!

星矢现在的状态是：DyingState

星矢被揍了4次

雅典娜说：星矢加油啊!!!

星矢现在的状态是：GoddessState

星矢被揍了5次

雅典娜说：星矢加油啊!!!

星矢现在的状态是：NormalState

星矢最初的状态是：NormalState

星矢被揍了1次

雅典娜说：星矢加油啊!!!

星矢现在的状态是：DyingState

星矢被揍了2次

雅典娜说：星矢加油啊!!!

星矢现在的状态是：UniverseState

星矢被揍了3次

雅典娜说：星矢加油啊!!!

星矢现在的状态是：DyingState

星矢被揍了4次

雅典娜说：星矢加油啊!!!

星矢现在的状态是：GoddessState

星矢被揍了5次

雅典娜说：星矢加油啊!!!

星矢现在的状态是：NormalState

总结：状态模式的缺点就是会弄出很多子类，如果状态没那么复杂，状态规则改变的可能性比较小的话就不要用了。

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第十章 中介公司——中介者模式

10.1 中介公司

时间：12月26日 地点：中介公司 人物：大B，中介公司业务员

大B和女朋友准备结婚了，他们想在结婚之前买房子，新房地段好的价格太贵，而他们又想用少的钱买到好房子。为此他们看过不少的房子，最后打算买二手房。于是，他们找到了中介公司。

中介公司业务员：“你想在哪里买房子？”

大B：“福田区。”

中介公司业务员：“打算买多大的？”

大B：“100平米”

中介公司业务员：“价位大概在多少的？”

大B：“60万”

中介公司业务员：“有没有什么要求？”

大B：“光线好，交通方便为主。”

中介公司业务员：“可以，有类似的房子我通知你过来看行吗？”

大B：“好的。”

10.2 中介者模式

中介在现实生活中并不陌生，满大街的房屋中介、良莠不齐的出国中介.....它们的存在是因为它们能给我们的生活带来一些便利，租房、买房用不着各个小区里瞎转；出国留学也不用不知所措。

小A：“什么叫中介者模式？”

大B：“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。简单点来说，将原来两个直接引用或者依赖的对象拆开，在中间加入一个‘中介’对象，使得两头的对象分别和‘中介’对象引用或者依赖。”

小A：“喔。所有的对象都需要加入‘中介’对象吗？”

大B：“当然并不是所有的对象都需要加入‘中介’对象。如果对象之间的关系原本一目了然，中介对象的加入便是‘画蛇添足’。”

10.3 中介者模式的组成部分

小A：“中介者模式它由哪些角色组成呢？”

大B：“1、抽象中介者角色：抽象中介者角色定义统一的接口用于各同事角色之间的通信。2、具体中介者角色：具体中介者角色通过协调各同事角色实现协作行为。为此它要知道并引用各个同事角色。3、同事角色：每一个同事角色都知道对应的具体中介者角色，而且与其他的同事角色通信的时候，一定要通过中介者角色协作。”

10.4 中介者模式的特点

大B：“你是否还记得应用广泛的MVC分为哪三层？”

小A：“记得，模型层、表现层还有控制层。”

大B：“对，控制层便是位于表现层与模型层之间的中介者。笼统地说MVC也算是中介者模式在框架设计中的一个应用。由于中介者模式在定义上比较松散，在结构上和观察者模式、命令模式十分相像；而应用目的又与结构模式“门面模式”有些相似。”

小A：“他们有哪些地方相似？”

大B：“在结构上，中介者模式与观察者模式、命令模式都添加了中间对象——只是中介者去掉了后两者在行为上的方向。因此中介者的应用可以仿照后两者的例子去写。但是观察者模式、命令模式中的观察者、命令都是被客户所知的，具体哪个观察者、命令的应用都是由客户来指定的；而大多中介者角色对于客户程序却是

透明的。当然造成这种区别的原因是由于它们要达到的目的不同。从目的上看，中介者模式与观察者模式、命令模式便没有了任何关系，倒是与前面讲过的门面模式有些相似。但是门面模式是介于客户程序与子系统之间的，而中介者模式是介于子系统与子系统之间的。这也注定了它们有很大的区别。”

小A：“他们都有什么区别呢？”

大B：“门面模式是将原有的复杂逻辑提取到一个统一的接口，简化客户对逻辑的使用。它是被客户所感知的，而原有的复杂逻辑则被隐藏了起来。而中介者模式的加入并没有改变客户原有的使用习惯，它是隐藏在原有逻辑后面的，使得代码逻辑更加清晰可用。”

小A：“使用中介者模式最大的好处是什么？”

大B：“使用中介者模式最大的好处就是将同事角色解耦。这带来了一系列的系统结构改善：提高了原有系统的可读性、简化原有系统的通信协议——将原有的多对多变为一对多、提高了代码的可复用性……但是中介者角色集中了太多的责任，所有有关的同事对象都要由它来控制。这不由得让我想起了简单工厂模式，但是由于中介者模式的特殊性——与业务逻辑密切相关，不能采用类似工厂方法模式的解决方法。因此建议在使用中介者模式的时候注意控制中介者角色的大小。”

小A：“那什么时候才是中介者模式的使用时机？”

大B：“一组对象以定义良好但是复杂的方式进行通信，产生了混乱的依赖关系，也导致对象难以复用。中介者模式很容易在系统中应用，也很容易在系统中误用。当系统出现了‘多对多’交互复杂的对象群，不要急于使用中介者模式，而要先反思你的系统在设计上是不是合理。”

10.5 男人和女人通过媒人约会

小A：“师兄，如果能举个例子的话，那就更好了。”

大B：“好。没问题的。那就举个通俗的模式吧！就拿一个男人和女人通过媒人约会的例子来实现这种设计模式吧。”

1、明确，男人和女人如果约会，假设男方提出约会(女方提出过程类似)。如果没有媒人，该过程如下：

A、男方如果想和女方约会

B、首先请示自己的父母(^_^,假设是封建家庭哦...),

C、然后通知女方

D、女方需要请示女方父母.是否同意约会。

弊端：需要男方交换的对象太多，且关系复杂，当随着男女之间的交往的复杂，会涉及到更多复杂的交换，双方父母的交换也很更复杂，使得四个对象(男、女、男方家长、女方家长)关系复杂，难以控制。

模式改进：采取媒人做‘中介者’-模式，将变得简单，其中无论那一个人有什么要求，请求，只需要告诉‘媒人’，至于该请求需要告诉谁，和谁交换，只有媒人知道。

//模拟代码类如下：

```

package mediator;

/**
 *
 */
public class Man {
    private Matchmaker mat; // 媒人
    public Man(Matchmaker mat) {
        this.mat=mat;
        mat.registeMan(this); //把自己在媒人那里注册(声明)
    }
    public static void main(String[ ] args) {
    }
    /**
    *考虑是否同意
    * @return
    */
    public boolean thinking(String says) {
        System.out.println("男人考虑:我该不该同意呢???");
        if (says.length() > 5){
            System.out.println("我同意了");
            return true;
        }
        else{
            System.out.println("我不同意.");
            return false;
        }
    }
    /**
    * 提出约会
    * 男人提出约会,只需要告诉媒人,由媒人负责跟其他人交互.
    * @param says
    */
    public void tryst(String says){

```

```

System.out.println("男人提出约会请求,说:"+says);
mat.doManTryst(says);
}
}
package mediator;
/**
*

```

Title: 男方家长类

*

Description:

```

* @version 1.0 */ public class ManParent { private Matchmaker mat;
//媒人类 public ManParent(Matchmaker mat) { this.mat = mat;
mat.registeManParent(this); //把自己在媒人那里注册(声明) } } /** *考
虑是否同意 * @return */ public boolean thinking(String says) {
System.out.println("男方父母考虑:我们做父母的该不该同意呢???"); if
(says.length() > 5){ System.out.println("我们做父母的同意了");
return true; } else{ System.out.println("我们做父母的不同意.");
return false; } }
////////////////////////////////////
package mediator; /** *

```

Title:女人类

*

Description:

```
*/ public class Woman { private Matchmaker mat; // 媒人 public
Woman(Matchmaker mat) { this.mat=mat; mat.registeWoman(this); //把
自己在媒人那里注册(声明) } /** *考虑是否同意 * @return */ public
boolean thinking(String says) { System.out.println("女人考虑:我该不
该同意呢???"); if (says.length() > 5){ System.out.println("我同意
了"); return true; } else{ System.out.println("我不同意."); return
false; } } /** * 提出约会 * 女人提出约会,只需要告诉媒人,由媒人负责跟其
他人交互. * @param says */ public void tryst(String says){
System.out.println("女人提出约会请求,说:"+says);
mat.doWomanTryst(says);
}
//////////////////////////////////// package
meditator; /** *
```

Title: 女方家长类

*

Description:

```
*/ public class WomanParent { private Matchmaker mat; //媒人类
public WomanParent(Matchmaker mat) { this.mat=mat;
mat.registerWomanParent(this); //把自己在媒人那里注册(声明) } /** *考
虑是否同意 * @return */ public boolean thinking(String says) {
System.out.println("女方父母考虑:我们做父母的该不该同意呢???"); if
```

```
(says.length() > 5){ System.out.println("我们做父母的同意了");
return true; } else{ System.out.println("我们做父母的不同意.");
return false; } }
```

////////////////////////////////////

```
package mediator; /** *
```

Title:媒人类-----中介者

*

Description:

```
* @version 1.0 */ public class Matchmaker { private Man man; //男人
private Woman woman; //女人 private ManParent manp; //男方父母
private WomanParent womanp; //女方父母 Matchmaker() { } public
static void main(String[] args) { Matchmaker matchmaker1 = new
Matchmaker(); } /** * 媒人处理男人提出的约会, * 男人提出约会,则只需要
把该问题告诉媒人, * 由媒人负责去跟其女方,女方父母,男方家长交互 * @param
says */ public void doManTryst(String says) {
System.out.println("媒人开始处理约会问题开始");
womanp.thinking(says); System.out.println("媒人处理约会问题结束,根据
同意的结果做其他处理"); } /** * 媒人处理女人提出的约会, * 由媒人负责去跟
男方,女方父母,男方家长交互 * @param says */ public void
doWomanTryst(String says) { System.out.println("媒人开始处理约会问题
开始"); man.thinking(says); manp.thinking(says);
womanp.thinking(says); System.out.println("媒人处理约会问题结束,根据
```

```

同意的结果做其他处理"); } /** * 处理彩礼等其他问题..... * @param man
*/ public void doOther(String says) { System.out.println("处理其他
问题");          man.thinking(says);          woman.thinking(says);
manp.thinking(says); womanp.thinking(says); } //以下四个注册方法,可
以理解为,男方和女方之间的通讯必须通过媒人 /** * 注册男人 * @param man
*/ public void registeMan(Man man){ this.man=man; } /** * 注册女人
* @param woman */ public void registeWoman(Woman woman){
this.woman=woman; } /** * 注册男方家长 * @param manp */ public void
registeManParent(ManParent manp) { this.manp = manp; } /** *注册女
方家长      *      @param      womanp      */      public      void
registerWomanParent(WomanParent womanp){ this.womanp = womanp; } }
package mediator; /** *

```

Title: 测试类---可以运行该类

*

Description:

*

Copyright: Copyright (c) 2004

```

* @version 1.0 */ public class MatchmakerDemo { public
MatchmakerDemo() { } public static void main(String[] args) {
MatchmakerDemo matchmakerDemo1 = new MatchmakerDemo(); Matchmaker
mat = new Matchmaker(); //媒人出现 Man man = new Man(mat); //男人,以

```



```
媒人为建立构造参数  Woman woman=new  Woman(mat);  ManParent  manp=new
ManParent(mat);      WomanParent  womanp=new      WomanParent(mat);
////////////////////
System.out.println("//////////////////////////////////////////
man.tryst("我想和你约会,可以吗?");          //////////////////
System.out.println("//////////////////////////////////////////
woman.tryst("想和你约会"); } }
```

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质
电子书下载！！！！

第十一章 高老庄的故事——代理模式

11.1 高老庄的故事

时间：12月27日 地点：大B房间 人物：大B，小A

这天大B和小A在一起聊起了《西游记》。

大B：“师弟，你小的时候喜欢看《西游记》吗？”

小A：“嘿嘿！很喜欢，记得以前天天看，重复看都不觉得腻。”

大B：“是啊！我都是喔。你记得高老庄悟空降八戒吗？”

小A：“记得。悟空代替了高家小姐去和八戒见面，制服了八戒。”

大B：“对！那春融时节，悟空牵着白马，与唐僧赶路西行。忽一日天色将晚，远远地望见一村人，这就是高老庄，猪八戒的丈人高太公家。”

小A：“对。”

大B：“为了将高家三小姐解救出八戒的魔掌，悟空决定扮做高小姐，会一会这个妖怪，行者却弄神通，摇身一变，变得就如那女子一般，独自个坐在房里等那妖

精。不多时，一阵风来，真个是走石飞砂……那阵狂风过处，只见半空里来了一个妖精，果然生得丑陋：黑脸短毛，长喙大耳，穿一领青不青、蓝不蓝的梭布直裰，系一条花布手巾……走进房，一把搂住，就要亲嘴……”

小A：“哈哈哈哈。这个情节我最爱看了。特搞笑。”

大B：“是啊。《西游记》里面太多的情节都让我百看不厌呐！”

11.2 代理模式

悟空的下手之处是将高家三小姐的神貌和她本人分割开来，这和“开一闭”原则有异曲同工之妙。这样一来，“高家三小姐本人”也就变成了“高家三小姐神貌”的具体实现，而“高家三小姐神貌”则变成了抽象角色。

小A：“这么说来，这就是所谓的代理模式吗？”

大B：“是啊！为其他对象提供一种代理以控制对这个对象的访问。说白了就是，在一些情况下客户不想或者不能直接引用一个对象，而代理对象可以在客户和目标对象之间起到中介作用，去掉客户不能看到的内容和服务或者增添客户需要的额外服务。”

小A：“那么什么时候要使用代理模式呢？”

大B：“在对已有的方法进行使用的时候出现需要对原有方法进行改进或者修改，这时候有两种改进选择：修改原有方法来适应现在的使用方式，或者使用一个‘第三者’方法来调用原有的方法并且对方法产生的结果进行一定的控制。第一

种方法是明显违背了‘对扩展开放、对修改关闭’（开闭原则），而且在原来方法中作修改可能使得原来类的功能变得模糊和多元化（就像现在企业多元化一样），而使用第二种方式可以将功能划分的更加清晰，有助于后面的维护。所以在一定程度上第二种方式是一个比较好的选择！当然，话又说回来了，如果是一个很小的系统，功能也不是很繁杂，那么使用代理模式可能就显得臃肿，不如第一种方式来的快捷。这就像一个三口之家，家务活全由家庭主妇或者一个保姆来完成是比较合理的，根本不需要雇上好几个保姆层层代理。”

11.3 代理模式的角色

小A：“代理模式一般涉及到哪些角色？”

大B：“抽象角色：声明真实对象和代理对象的共同接口；代理角色：代理对象角色内部含有对真实对象的引用，从而可以操作真实对象，同时代理对象提供与真实对象相同的接口以便在任何时刻都能代替真实对象。同时，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。真实角色：代理角色所代表的真实对象，是我们最终要引用的对象。”

使用类图来表示下三者间的关系如图11-1：

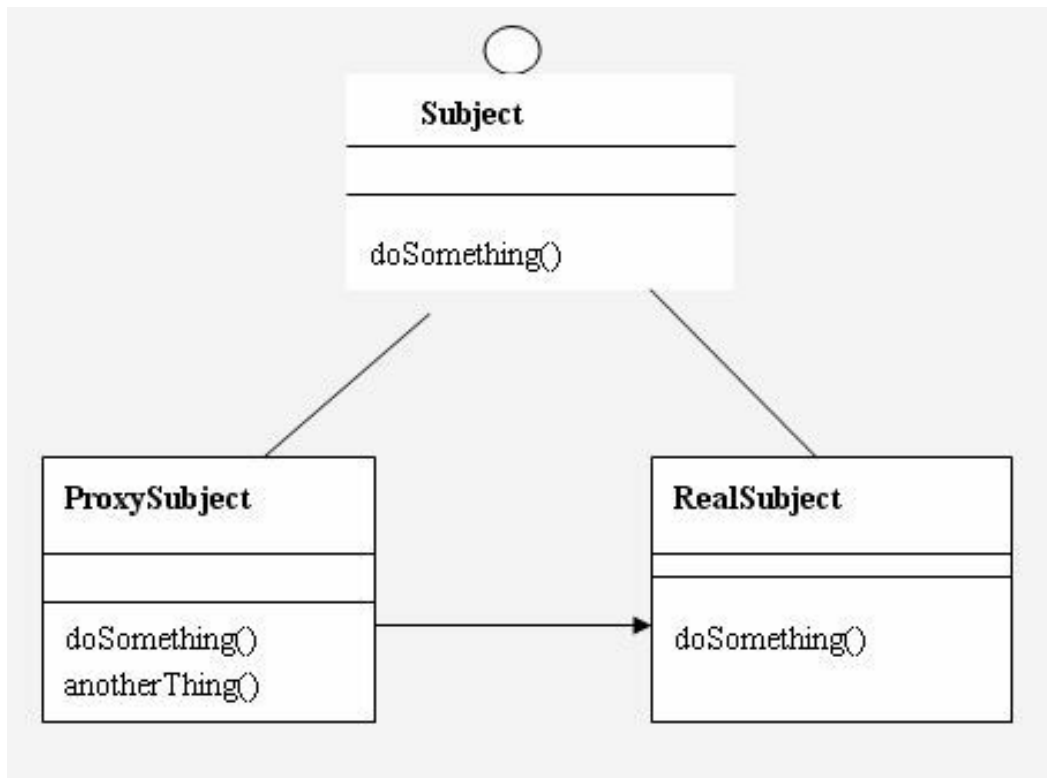


图11-1 代理模式类图

抽象角色：

```
abstract public class Subject
{
    abstract public void request();
}
```

真实角色：实现了Subject的request()方法。

```
public class RealSubject extends Subject
{
    public RealSubject()
    {
    }
    public void request()
    {
        System.out.println("From real subject.");
    }
}
```

```
}  
}
```

代理角色：

```
public class ProxySubject extends Subject  
{  
    private RealSubject realSubject; //以真实角色作为代理角色的属性  
    public ProxySubject()  
    {  
    }  
    public void request() //该方法封装了真实对象的request方法  
    {  
        preRequest();  
        if( realSubject == null )  
        {  
            realSubject = new RealSubject();  
        }  
        realSubject.request(); //此处执行真实对象的request方法  
        postRequest();  
    }  
    private void preRequest()  
    {  
        //something you want to do before requesting  
    }  
    private void postRequest()  
    {  
        //something you want to do after requesting  
    }  
}
```

客户端调用：

```
Subject sub=new ProxySubject();
```

```
Sub.request();
```

大B：“由以上代码可以看出，客户实际需要调用的是RealSubject类的request()方法，现在用ProxySubject来代理RealSubject类，同样达到目的，同时还封装了其他方法(preRequest(),postRequest())，可以处理一些其他问题。另外，如果要按照上述的方法使用代理模式，那么真实角色必须是事先已经存在的，并将其作为代理对象的内部属性。但是实际使用时，一个真实角色必须对应一个代理角色，如果大量使用会导致类的急剧膨胀。”

小A：“如果事先并不知道真实角色，该如何使用代理呢？”

大B：“这个问题可以通过Java的动态代理类来解决。”

11.4 已注册用户和游客的权限

大B：“下面以论坛中已注册用户和游客的权限不同来作为第一个例子，我再给你详细说一下。”

大B：“已注册的用户拥有发帖，修改自己的注册信息，修改自己的帖子等功能；而游客只能看到别人发的帖子，没有其他权限。为了简化代码，更好的显示出代理模式的骨架，我们这里只实现发帖权限的控制。首先我们先实现一个抽象主题角色MyForum，里面定义了真实主题和代理主题的共同接口——发帖功能。”

代码如下：

```
public interface MyForum
{
    public void AddFile();
}
```

大B：“这样，真实主题角色和代理主题角色都要实现这个接口。其中真实的主题角色基本就是将这个方法内容填充进来。所以在这里就不再赘述它的实现。我们把主要的精力放到关键的代理主题角色上。”

代理主题角色代码大体如下：

```
public class MyForumProxy implements MyForum
{
    private RealMyForum forum ;
    private int permission ; //权限值
    public MyForumProxy(int permission)
    {
        forum = new RealMyForum()
        this.permission = permission ;
    }
    //实现的接口
    public void AddFile()
    {
        //满足权限设置的时候才能够执行操作
        //Constants是一个常量类
        if(Constants.ASSOCIATOR == permission)
        {
            forum.AddFile();
        }
        else
```



```
System.out.println("You are not a associator of MyForum ,please registe!");  
}  
}
```

大B：“这样就实现了代理模式的功能。当然你也可以在这个代理类上添加自己的方法来实现额外的服务，比如统计帖子的浏览次数，记录用户的登录情况等等。还有一个很常见的代理模式的使用例子就是对大幅图片浏览的控制。”

小A：“当我们在网站上面浏览图文的信息时，图片位置放置的是经过缩小的，当有人要仔细的查看这个图片时，可以通过点击图片来激活一个链接，在一个新的网页打开要看的图片。”

大B：“嗯。对。这样对于提高浏览速度是很有好处的，因为不是每个人都要去看仔细图上的信息。”

小A：“是吗？”

大B：“这种情况就可以使用代理模式来全面实现。这里我将思路表述出来，至于实现由于工作原因，就不表述了，至于这种方式在B/S模式下的真实可行性，我没有确认过，只是凭空的想象。如果不是可行的方式，那这个例子可以放到一个C/S下来实现，这个是绝对没有问题的，而且在很多介绍设计模式的书和文章中使用。两种方式的实现有兴趣的可以尝试一下。我们在浏览器中访问网页时是调用的不是真实的装载图片的方法，而是在代理对象中的方法，在这个对象中，先使用一个线程向浏览器装载了一个缩小版的图片，而在后台使用另一个线程来调用真实的装载大图片的方法将图片加载到本地，当你要浏览这个图片的时候，将其在新的网页中显示出来。当然如果在你想浏览的时候图片尚未加载成功，可以再启动一个线程来显示提示信息，直到加载成功。这样代理模式的功能就在上面体现的淋漓尽致——

通过代理来将真实图片的加载放到后台来操作，使其不影响前台的浏览。代理模式能够协调调用者和被调用者，能够在一定程度上降低系统的耦合度。不过一定要记住前面讲的使用代理模式的条件，不然的话使用了代理模式不但不会有好的效果，说不定还会出问题的。”

11.5 代理分类

小A：“代理分为哪些类？”

大B：“代理分为静态代理与动态代理。”

小A：“按功能怎么分类哩？”

大B：“按功能将代理的组成类分为：标的类、标的接口、拦截类、耦合类。”

下面以具本代码举例说明。

1、静态与动态代理的公共部分

```
package proxy.common;

/**
 *
 * *   AngelSoftware, Inc.
 *   Copyright (C): 2008
 *   Description:
 *   TODO 标的类
 *
 *   Revision History:
```

```

*
*/
public class TargetImpl implements Target1, Target2
{
    public void doSomething()
    {
        System.out.println("doSomething!");
    }
    public String do1(String msg)
    {
        // TODO Auto-generated method stub
        System.out.println("do1: " + msg);
        return "this is String Mehtod!";
    }
    public int do2()
    {
        System.out.println("do2!");
        return 1000;
    }
}

package proxy.common;

/**
 *
 *
 * AngelSoftware, Inc.
 * Copyright (C): 2008
 *
 * Description:
 *   TODO 标的接口
 *
 * Revision History:
 *   Jun 5, 2008 fity.wang initial version.

```

```
*
*
*/
public interface Target1
{
void doSomething();
}
package proxy.common;
/**
*
*
*   AngelSoftware, Inc.
*   Copyright (C): 2008
*
*   Description:
*   TODO 标的接口
*
*   Revision History:
*   Jun 5, 2008 fity.wang initial version.
*
*
*/
public interface Target2
{
String do1(String msg);
int do2();
}
package proxy.common;
/**
*
*
*
```

```

*   Description:
*   TODO 拦截类
*
*   Revision History:
*   Jun 5, 2008 fity.wang initial version.
*
*
*/
public class Intercept
{
    public void before()
    {
        System.out.println("Before.....");
    }
    public void after()
    {
        System.out.println("After.");
    }
}

```

2、静态代理特征部分

```

package proxy.jingtai;
import proxy.common.Intercept;
import proxy.common.TargetImpl;
/**
 *
 * *   AngelSoftware, Inc.
 *   Copyright (C): 2008
 *
 *   Description:
 *   TODO 耦合类(耦合是为了解耦)

```

```

*
*
*
*/
public class Invocation
{
    public Object invokeDoSomething()
    {
        TargetImpl t = new TargetImpl();
        Intercept p = new Intercept();
        //调用真实的标的类的方法之前置入拦截类的方法
        p.before();
        //调用真实的标的类的方法
        t.doSomething();
        //调用真实的标的类的方法之后置入拦截类的方法
        p.after();
        return null;
    }
}

package proxy.jingtai;

/**
 *
 *
 *
 * Description:
 *   TODO 静态代理(这理只简单地讲一下,着重讲动态代理)
 *
 * Revision History:
 *   Jun 5, 2008 fity.wang initial version.
 *
 *
 */

```

```
public class Test
{
    public static void main(String args[])
    {
        new Invocation().invokeDoSomething();
    }
}
```

3、动态代理特征部分

```
package proxy.dongtai;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import proxy.common.Intercept;
import proxy.common.TargetImpl;
/**
 *
 * *   AngelSoftware, Inc.
 *   Copyright (C): 2008
 *
 *   Description:
 *   TODO 耦合类(耦合是为了解耦)
 *
 *   Revision History:
 *   Jun 5, 2008 fity.wang initial version.
 *
 *
 */
public class Invocation implements InvocationHandler
{
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable
```

```

{
    TargetImpl t = new TargetImpl();
    Intercept p = new Intercept();
    if (args!=null&&args.length ==1)
    {
        //更改参数
        args[0] = "param value has changed";
    }
    //调用真实的标的类的方法之前置入拦截类的方法
    p.before();
    //调用真实的标的类的方法
    Object o = method.invoke(t, args);
    //调用真实的标的类的方法之后置入拦截类的方法
    p.after();
    return o;
}
}

package proxy.dongtai;
import proxy.common.Target1;
import proxy.common.Target2;
/**
 *
 *
 * AngelSoftware, Inc.
 * Copyright (C): 2008
 *
 * Description:
 * TODO 测试类
 *
 *
 */

```



```

public class Test
{
/**
 * logic1与logic的共同逻辑
 * @param proxy 代理
 */
private static void publicLogic(Object proxy)
{
//对目标接口Target1代理的调用
System.out.println("对目标接口Target1代理的调用");
Target1 t1 = (Target1)proxy;
t1.doSomething();
System.out.println();
//对目标接口Target2的调用
System.out.println("对目标接口Target2代理的调用");
Target2 t2 = (Target2)proxy;
System.out.println("Target Mehod do1 return: " + t2.do1("hello!"));
System.out.println("Target Mehod do2 return: " + t2.do2());
System.out.println();
System.out.println();
}
/**
 * new Class[] {Target2.class, Target1.class}
 * 正常
 * @return
 */
public static void logic1()
{
Invocation iv = new Invocation();
/*
 * Proxy.newProxyInstance的参数说明
 * 参数1:类加载器(个人感觉这个参数有点多余,这个参数完成可以去掉,不知当初他们为何要设这个参数

```

干么)

- * 参数2:代理的标的接口.就是说,你要代理的标的类可能会实现多个接口,你可以有选择性地代理这些接口

- * 参数3:InvocationHandler的实现类.InvocationHandler接口做用就是解耦,解开标的类与拦截类之间的耦合,使它们之间可以互不关心

*/

```
Object proxy = java.lang.reflect.Proxy.newProxyInstance(Thread.currentThread().getCor  
{Target2.class,Target1.class}, iv);
```

```
publicLogic(proxy);
```

```
}
```

```
/**
```

- * new Class[]{Target1.class}

- * 将会出异常,因为他没有在参数中声明自己要调用Target2接口,而后面却又去调用

- * @return

*/

```
public static void logic2()
```

```
{
```

```
Invocation iv = new Invocation();
```

```
/*
```

- * Proxy.newProxyInstance的参数说明

- * 参数1:类加载器(个人感觉这个参数有点多余,这个参数完成可以去掉,不知当初他们为何要设这个参数干么)

- * 参数2:代理的标的接口.就是说,你要代理的标的类可能会实现多个接口,你可以有选择性地代理这些接口

- * 参数3:InvocationHandler的实现类.InvocationHandler接口做用就是解耦,解开标的类与拦截类之间的耦合,使它们之间可以互不关心

*/

```
Object proxy = java.lang.reflect.Proxy.newProxyInstance(Thread.currentThread().getCor  
{Target1.class}, iv);
```

```
publicLogic(proxy);
```

```
}
```

```
public static void main(String args[])
```

```
{  
    logic1();  
    logic2();  
}  
}
```

11.6 代理模式分为8种，列举几种常见的、重要的。

小A：“师兄，我知道代理模式分为8种，能不能列举几种常见的、重要的？”

大B：“可以。我给你讲几种较常见也是比较重要的几种。1、远程（Remote）代理：为一个位于不同的地址空间的对象提供一个局域代表对象。比如：你可以将一个在世界某个角落一台机器通过代理假象成你局域网中的一部分。2、虚拟（Virtual）代理：根据需要将一个资源消耗很大或者比较复杂的对象延迟的真正需要时才创建。比如：如果一个很大的图片，需要花费很长时间才能显示出来，那么当这个图片包含在文档中时，使用编辑器或浏览器打开这个文档，这个大图片可能就影响了文档的阅读，这时需要做个图片Proxy来代替真正的图片。3、保护（Protect or Access）代理：控制对一个对象的访问权限。比如：在论坛中，不同的身份登陆，拥有的权限是不同的，使用代理模式可以控制权限（当然，使用别的方式也可以实现）。4、智能引用（Smart Reference）代理：提供比对目标对象额外的服务。比如：纪录访问的流量（这是个再简单不过的例子），提供一些友情提示等等。代理模式是一种比较有用的模式，从几个类的‘小结构’到庞大系统

的 ‘大结构’ 都可以看到它的影子。”

11.7 动态代理类

小A：“动态代理类位于哪里？”

大B：“Java动态代理类位于Java.lang.reflect包下。”

小A：“他一般会涉及到哪些类呢？”

大B：“一般主要涉及到以下两个类：1、Interface InvocationHandler：该接口中仅定义了一个方法Object：invoke(Object obj, Method method, J2EEjava语言JDK1.4APIjavalangObject.html">Object[] args)。在实际使用时，第一个参数obj一般是指代理类，method是被代理的方法，如上例中的request()，args为该方法的参数数组。这个抽象方法在代理类中动态实现。2、Proxy：该类即为动态代理类，作用类似于上例中的ProxySubject，其中主要包含以下内容：Protected Proxy(InvocationHandler h)：构造函数，估计用于给内部的h赋值。Static Class getProxyClass (ClassLoader loader, Class[] interfaces)：获得一个代理类，其中loader是类装载器，interfaces是真实类所拥有的全部接口的数组。Static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h)：返回代理类的一个实例，返回后的代理类可以当作被代理类使用(可使用被代理类的在Subject接口中声明过的方法)。”

小A：“那什么是Dynamic Proxy？”

大B：“所谓Dynamic Proxy是这样一种class：它是在运行时生成的class，在生成它时你必须提供一组interface给它，然后该class就宣称它实现了这些interface。你当然可以把该class的实例当作这些interface中的任何一个来用。当然啦，这个Dynamic Proxy其实就是一个Proxy，它不会替你作实质性的工作，在生成它的实例时你必须提供一个handler，由它接管实际的工作。在使用动态代理类时，我们必须实现InvocationHandler接口，以第一节中的示例为例。”

抽象角色(之前是抽象类，此处应改为接口)：

```
public interface Subject
{
    abstract public void request();
}
```

具体角色RealSubject：同上；

代理角色：

```
import java.lang.reflect.Method;
import java.lang.reflect.InvocationHandler;

public class DynamicSubject implements InvocationHandler {
    private Object sub;

    public DynamicSubject() {
    }

    public DynamicSubject(Object obj) {
        sub = obj;
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("before calling " + method);
        method.invoke(sub, args);
    }
}
```

```

System.out.println("after calling " + method);
return null;
}
}

```

大B：“该代理类的内部属性为Object类，实际使用时通过该类的构造函数DynamicSubject(Object obj)对其赋值；此外，在该类还实现了invoke方法，该方法中的method.invoke(sub,args)；其实就是调用被代理对象的将要被执行的方法，方法参数sub是实际的被代理对象，args为执行被代理对象相应操作所需的参数。通过动态代理类，我们可以在调用之前或之后执行一些相关操作。”

客户端：

```

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

public class Client
{
    static public void main(String[] args) throws Throwable
    {
        RealSubject rs = new RealSubject(); //在这里指定被代理类
        InvocationHandler ds = new DynamicSubject(rs); //初始化代理类
        Class cls = rs.getClass();
        //以下是分解步骤
        /*
        Class c = Proxy.getProxyClass(cls.getClassLoader(),cls.getInterfaces()) ;
        Constructor ct=c.getConstructor(new Class[]{InvocationHandler.class});
        Subject subject =(Subject) ct.newInstance(new Object[]{ds});
        */
    }
}

```

```
//以下是一次性生成
Subject subject = (Subject) Proxy.newProxyInstance(cls.getClassLoader(),
cls.getInterfaces(),ds );
subject.request();
}
```

大B：“通过这种方式，被代理的对象(RealSubject)可以在运行时动态改变，需要控制的接口(Subject接口)可以在运行时改变，控制的方式(DynamicSubject类)也可以动态改变，从而实现了非常灵活的动态代理关系。”

11.8 代理模式使用原因和应用方面

小A：“为什么使用代理模式？”

大B：“1、授权机制不同级别的用户对同一对象拥有不同的访问权利，如Jive论坛系统中，就使用Proxy进行授权机制控制，访问论坛有两种人：注册用户和游客（未注册用户），Jive中就通过类似ForumProxy这样的代理来控制这两种用户对论坛的访问权限。2、某个客户端不能直接操作到某个对象，但又必须和那个对象有所互动。”

小A：“能不能举个例子啊？”

大B：“可以，就举例两个具体情况：1、如果那个对象是一个很大的图片，需要花费很长时间才能显示出来，那么当这个图片包含在文档中时，使用编辑器或浏览器打开这个文档，打开文档必须很迅速，不能等待大图片处理完成，这时需要做个图片Proxy来代替真正的图片。2、如果那个对象在Internet的某个远端服务器

上，直接操作这个对象因为网络速度原因可能比较慢，那我们可以先用Proxy来代替那个对象。总之原则是，对于开销很大的对象，只有在使用它时才创建，这个原则可以为我们节省很多宝贵的Java内存。所以，有些人认为Java耗费资源内存，我以为这和程序编制思路也有一定的关系。”

小A：“那他一般用在哪些地方哩？”

大B：“现实中，Proxy应用范围很广，现在流行的分布计算方式RMI和Corba等都是Proxy模式的应用。”

11.9 代理模式的作用

小A：“代理模式有什么作用哩？”

大B：“为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个客户不想或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。”

11.10 秘书-局长

大B：“再举个通俗的例子，你想找某局长帮你做一件事情，但局长官位显赫，你又不能轻易见着，你就想到了找他的秘书，通过她传话给局长，这样你就等于请他的秘书帮你办成了那件事。秘书为什么就可以找到局长呢，因为秘书和局长之间

有一定的关系。这里产生了四个对象：你、秘书、局长、秘书-局长（关系）。JAVA中同样有代理关系，我们叫做代理模式。”

小A：“代理模式有什么作用呢？”

大B：“他能为其他对象（局长）提供一种代理（秘书）以控制对这个对象（局长）的访问。代理对象可以在客户端（你）和目标对象（局长）之间起到中介的作用。”

小A：“代理模式都有些什么角色？”

大B：“1、抽象角色（秘书-局长）：声明真实对象和代理对象的共同接口（秘书-局。2、代理角色（秘书）：代理对象角色（秘书）内部含有对真实对象（局长）的引用，从而可以操作真实对象（局长），同时代理对象（秘书）提供与真实对象（局长）相同的接口（秘书-局长）以便在任何时刻都能代替真实对象（局长）。同时，代理对象（秘书）可以在执行真实对象（局长）操作时，附加其他的操作，相当于对真实对象（局长）进行封装。3、真实角色（局长）：代理角色（秘书）所代表的真实对象（局长），是我们最终要引用的对象（局长）。”

下面用四个代码来是这个原理：

Client.java(你)、ProxySubject.java(秘书)、RealSubject.java(局长)、Subject.java(关系)

(1) Subject.java(关系)

```
package com.pjwqh.proxyTest;
// 抽象角色
abstract public class Subject
```

```
{  
abstract public void request();  
}
```

(2) RealSubject.java(局长)

```
package com.pjwqh.proxyTest;  
//真实角色：实现了Subject的request()方法  
public class RealSubject extends Subject  
{  
    public RealSubject()  
    {  
    }  
    public void request()  
    {  
        System.out.println("我是局长，哈哈");  
    }  
}
```

(3) ProxySubject.java(秘书)

```
package com.pjwqh.proxyTest;  
//代理角色  
public class ProxySubject extends Subject  
{  
    private RealSubject realSubject; // 以真实角色作为代理角色的属性  
    public ProxySubject()  
    {  
    }  
    public void request() // 该方法封装了真实对象的request方法  
    {  
    }  
}
```

```

preRequest();
if (realSubject == null)
{
    realSubject = new RealSubject();
}
realSubject.request(); // 此处执行真实对象的request方法
postRequest();
}

private void preRequest()
{
    // something you want to do before requesting
}

private void postRequest()
{
    // something you want to do after requesting
}
}

```

(4) Client.java(你)

```

package com.pjwqh.proxyTest;
//客户端调用
public class Client
{
    public static void main(String[] args)
    {
        // 你直接找 ( 秘书 )
        Subject sub = new ProxySubject();
        sub.request();
    }
}

```

运行输出了“我是局长，哈哈”

大B：“这说明我们通过代理对象（秘书）成功调用了被代理对象（局长）的方法。由代码可以看出，客户实际需要调用的是RealSubject类的request()方法，现在用ProxySubject来代理 RealSubject类，同样达到目的，同时还封装了其他方法(preRequest(),postRequest()),可以处理一些其他问题。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第十二章 包子——享元模式

12.1 包子

时间：12月29日 地点：XX早餐店 人物：大B，小A

今天是星期日，大B和小A约好了去喝早茶。

他们早早到早餐店里，由于这个早餐店包子是出了名的好吃。所以他们点了菜包和肉包。不一会，早餐店里坐满了人。

小A：“这里不知道有什么包子？这么受欢迎。”

大B：“这个店主要是做菜包和肉包的。”

小A：“就只有这两种口味吗？”

大B：“是啊！”

小A：“可是为什么这么受欢迎哩？这么多来这里吃包子的人都只是点菜包和肉包这两种包子吗？”

大B：“是啊！”

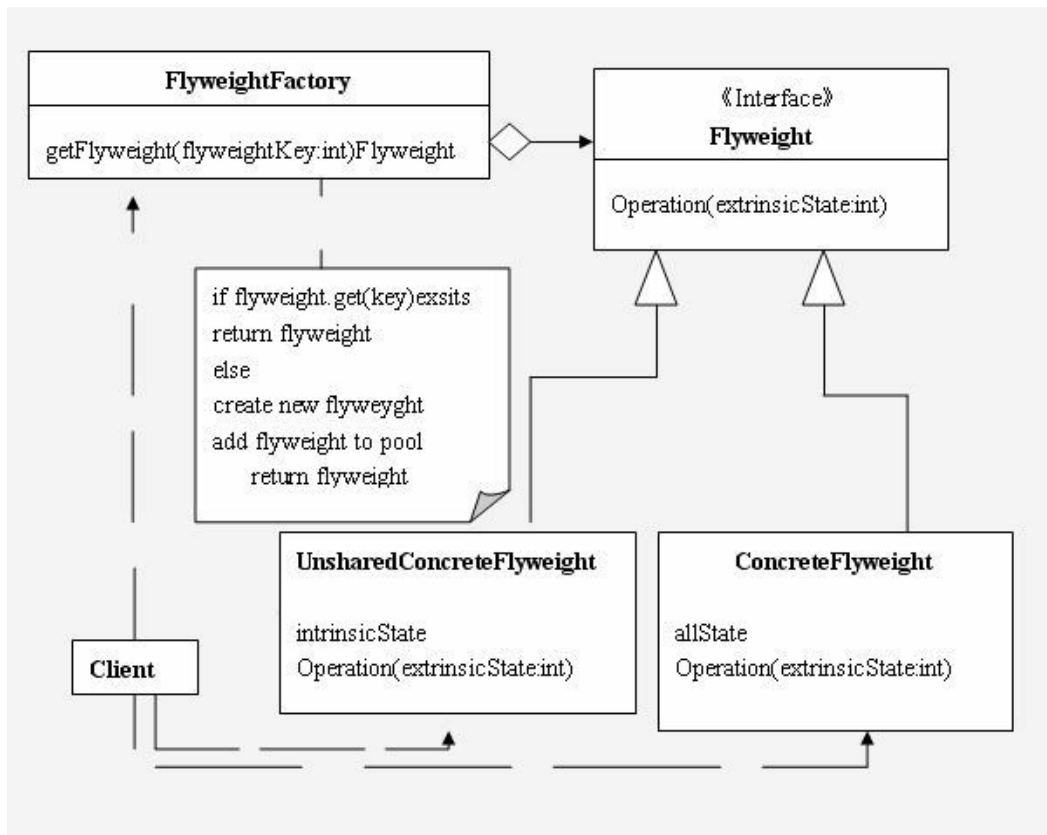
12.2 享元模式

大B：“你现在知道享元模式的意图了吗？”

小A：“享元模式的意图是运用共享技术有效地支持大量细粒度的对象。”

大B：“是的。也就是说在一个系统中如果有多个相同的对象，那么只共享一份就可以了，不必每个都去实例化一个对象。在Flyweight模式中，由于要产生各种各样的对象，所以在Flyweight(享元)模式中常出现Factory模式。Flyweight的内部状态是用来共享的，Flyweight factory负责维护一个对象存储池（Flyweight Pool）来存放内部状态的对象。Flyweight模式是一个提高程序效率和性能的模式，会大大加快程序的运行速度。”

如图12-1所示享元模式结构图



12.3 享元模式原理

小A：“享元对象能做到共享的关键是区分内蕴状态（Internal State）和外蕴状态（External State）。”

大B：“是的。一个内蕴状态是存储在享元对象内部的，并且不会随环境改变而有所不同的。因此，一个享元可以具有内蕴状态并可以共享。一个外蕴状态是随环境改变而改变的，不可以共享状态。享元对象的外蕴状态必须由客户端保存，并在享元对象被创建之后，在需要使用的时候再传到享元对象内部。外蕴状态不可以影响享元对象的内蕴状态，它们是相互独立的。所有的内蕴状态在对象创建完后就不可再改变。”

12.4 享元模式设计初衷

小A：“师兄，享元模式的设计初衷是什么？”

大B：“面向对象语言的原则就是一切都是对象，但是如果真正使用起来，有时对象数可能显得很庞大，比如，字处理软件，如果以每个文字都作为一个对象，几千个字，对象数就是几千，无疑耗费内存，那么我们还是要‘求同存异’，找出这些对象群的共同点，设计一个元类，封装可以被共享的类，另外，还有一些特性是取决于应用(context)，是不可共享的。”

12.5 咖啡外卖店

大B：“我就以咖啡外卖店写几个Java类来描述说明Flyweight设计模式的实现方式吧。”

客户买咖啡下订单，订单只区分咖啡口味，如果下了1W个订单，而咖啡店只卖20种口味的咖啡，那么我们就没有必要生成1W个订单对象，通过享元模式我们只需要生成20个订单对象。

这个例子举的不太好，但足以说明问题。下面是具体的代码。

- 1、 Order.java 订单抽象类
- 2、 FlavorOrder.java 订单实现类
- 3、 FlavorFactory.java 订单生成工厂
- 4、 Client.java 客户类、带有main方法的测试类

===== 1、 Order.java

```
package flyweight;

public abstract class Order {
    //执行卖出动作
    public abstract void sell();
    //获取咖啡口味
    public abstract String getFlavor();
}
```


===== 1 end

===== 2、 FlavorOrder.java

```
package flyweight;

public class FlavorOrder extends Order{
    private String flavor;

    public FlavorOrder(String flavor){
        this.flavor = flavor;
    }

    public String getFlavor(){
        return this.flavor;
    }

    public void sell(){
        System.out.println("卖出一杯 [" + flavor + "]. " );
    }
}
```

===== 2 end

===== 3、 FlavorFactory.java

```
package flyweight;

import java.util.HashMap;
import java.util.Map;

public class FlavorFactory {
    //订单池
    private Map flavorPool = new HashMap(20);
    //静态工厂,负责生成订单对象
    private static FlavorFactory flavorFactory = new FlavorFactory();
}
```

```

private FlavorFactory() {}

public static FlavorFactory getInstance() {
    return flavorFactory;
}

//获得订单
public Order getOrder(String flavor) {
    Order order = null;
    if(flavorPool.containsKey(flavor)){
        order = flavorPool.get(flavor);
    }else{
        //获得新口味订单
        order = new FlavorOrder(flavor);
        //放入对象池
        flavorPool.put(flavor, order);
    }
    return order;
}

//获得已经卖出的咖啡全部口味数量
public int getTotalFlavorsMade() {
    return flavorPool.size();
}
}

```

===== 3 end

===== 4、 Client.java

```

package flyweight;

import java.util.ArrayList;
import java.util.List;

public class Client {

```

```
//客户下的订单
private static List orders = new ArrayList(100);
//订单对象生成工厂
private static FlavorFactory flavorFactory;
//增加订单
private static void takeOrders(String flavor) {
    orders.add(flavorFactory.getOrder(flavor));
}
public static void main(String[] args) {
//订单生成工厂
flavorFactory = FlavorFactory.getInstance();
//增加订单
takeOrders("摩卡");
takeOrders("卡布奇诺");
takeOrders("香草星冰乐");
takeOrders("香草星冰乐");
takeOrders("拿铁");
takeOrders("卡布奇诺");
takeOrders("拿铁");
takeOrders("卡布奇诺");
takeOrders("摩卡");
takeOrders("香草星冰乐");
takeOrders("卡布奇诺");
takeOrders("摩卡");
takeOrders("香草星冰乐");
takeOrders("拿铁");
takeOrders("拿铁");
//卖咖啡
for(Order order : orders){
    order.sell();
}
//打印生成的订单Java对象数量
```

```
System.out.println("\n客户一共买了 " + orders.size() + " 杯咖啡! ");  
//打印生成的订单Java对象数量  
System.out.println("\n共生成  
了 " + flavorFactory.getTotalFlavorsMade() + " 个 FlavorOrder Java对象! ");  
}  
}
```

12.6 享元模式特征

大B：“你知道享元模式有哪些特征吗？”

小A：“享元模式包括有单纯享元模式和复合享元模式。他们都有不同的角色不同的特征。”

大B：“下面我来具体说说。

首先单纯享元模式，它有抽象享元角色、具体享元角色、享元工厂角、客户端角色。

抽象享元角色：所有的具体享元类的超类，规定出需要实现的公共接口。那些需要外蕴状态的操作可以通过方法的参数传入。

具体享元角色：实现抽象享元角色所规定的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。享元对象的内蕴状态必须与对象所处的周围环境无关，从而使得享元对象可以在系统内共享。

享元工厂角：负责创建和管理享元角色。本角色必须保证享元对象可以被系统

适当地共享。当一个客户端对象调用一个享元对象时，享元工厂角色会检查系统中是否已经有一个符合要求的享元对象。如果有，享元工厂就提供这个已经有的享元对象，如果没有，享元工厂创建一个适当的享元对象。

客户端角色：需要维护一个对所有享元对象的引用。本角色需要自行存储所有享元对象的外蕴状态。

还有就是复合享元模式，它有抽象享元角色、具体享元角色、复合享元角色、享元工厂角、客户端角色。

抽象享元角色：所有的具体享元类的超类，规定出需要实现的公共接口。那些需要外蕴状态的操作可以通过方法的参数传入。抽象享元的接口使得享元变得可能，但是并不强制子类实行共享，因此并非所有的享元对象都是可以共享的。

具体享元角色：实现抽象享元角色所规定的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。享元对象的内蕴状态必须与对象所处的周围环境无关，从而使得享元对象可以在系统内共享。

复合享元角色是由具体享元角色通过复合而成。复合享元角色：复合享元角色所代表的对象是不可以共享的，但是可以分解成多个可以共享的具体享元角色。

享元工厂角：负责创建和管理享元角色。本角色必须保证享元对象可以被系统适当地共享。当一个客户端对象调用一个享元对象时，享元工厂角色会检查系统中是否已经有一个符合要求的享元对象。如果有，享元工厂就提供这个已经有的享元对象，如果没有，享元工厂创建一个适当的享元对象。

客户端角色：需要维护一个对所有享元对象的引用。本角色需要自行存储所有

享元对象的外蕴状态。”

如图12-2单纯享元模式类图、图12-3复合享元模式类图

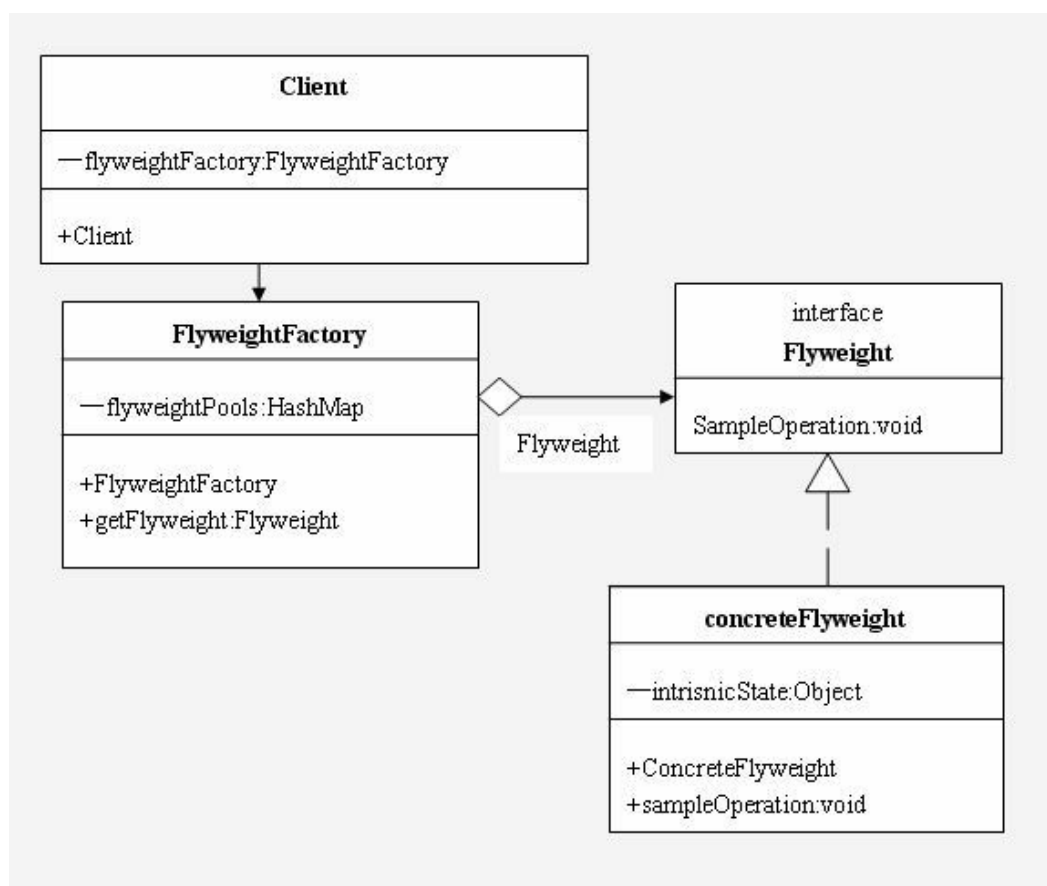


图12-2 单纯享元模式类图

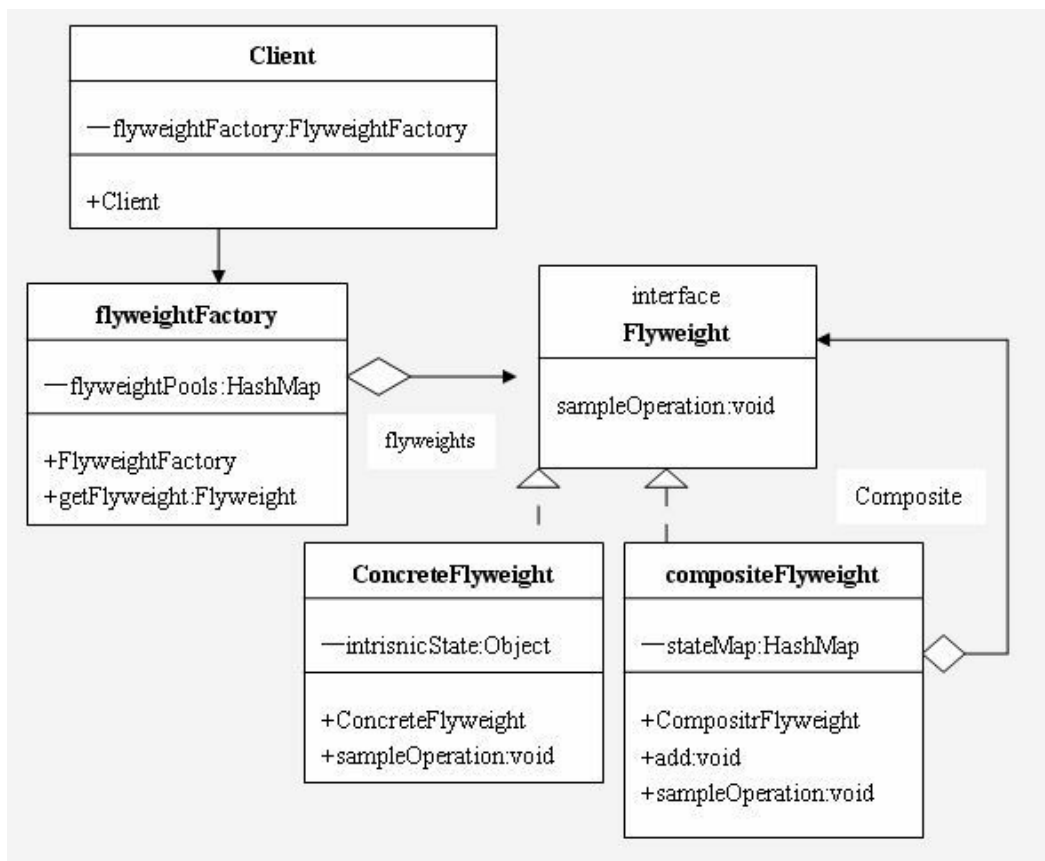


图12-3 复合享元模式类图

12.7 适用性

大B：“你说一下享元模式适用于哪些地方？”

小A：“Flyweight模式的有效性很大程度上取决于如何使用它以及在何处使用它。当以下情况都成立时使用Flyweight模式。1、一个应用程序使用了大量的对象。2、完全由于使用大量的对象，造成很大的存储开销。3、对象的大多数状态都可变为外部状态。4、如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。5、应用程序不依赖对象标识。”

12.8 为什么使用享元模式?

小A：“为什么要使用享元模式?”

大B：“面向对象语言的原则就是一切都是对象，但是如果真正使用起来，有时对象数可能显得很庞大，比如，字处理软件，如果以每个文字都作为一个对象，几千个字，对象数就是几千，无疑耗费内存，那么我们还是要‘求同存异’，找出这些对象群的共同点，设计一个元类，封装可以被共享的类，另外，还有一些特性是取决于应用(context)，是不可共享的，这也Flyweight中两个重要概念内部状态intrinsic和外部状态extrinsic之分。说白点，就是先捏一个的原始模型，然后随着不同场合和环境，再产生各具特征的具体模型，很显然，在这里需要产生不同的新对象，所以Flyweight模式中常出现Factory模式.Flyweight的内部状态是用来共享的，Flyweight factory负责维护一个Flyweight pool(模式池)来存放内部状态的对象。”

大B：“Flyweight模式是一个提高程序效率和性能的模式，会大大加快程序的运行速度.应用场合很多：比如你要从一个数据库中读取一系列字符串，这些字符串中有许多是重复的，那么我们可以将这些字符串储存在Flyweight池(pool)中。”

12.9 如何使用享元模式?

小A：“如何去使用享元模式?”

大B：“我们先从Flyweight抽象接口开始。”


```
public interface Flyweight
{
    public void operation( ExtrinsicState state );
}
//用于本模式的抽象数据类型(自行设计)
public interface ExtrinsicState { }
```

大B：“接下来我们讲的是接口的具体实现(ConcreteFlyweight)，并为内部状态增加内存空间， ConcreteFlyweight必须是可共享的,它保存的任何状态都必须是内部(intrinsic)，也就是说，ConcreteFlyweight必须和它的应用环境场合无关。”

```
public class ConcreteFlyweight implements Flyweight {
    private IntrinsicState state;
    public void operation( ExtrinsicState state )
    {
        //具体操作
    }
}
```

小A：“是不是所有的Flyweight具体实现子类都需要被共享？”

大B：“当然，并不是所有的Flyweight具体实现子类都需要被共享的，所以还有另外一种不共享的ConcreteFlyweight。”

```
public class UnsharedConcreteFlyweight implements Flyweight {
    public void operation( ExtrinsicState state ) { }
```

```
}
```

大B：“Flyweight factory负责维护一个Flyweight池(存放内部状态)，当客户端请求一个共享Flyweight时,这个factory首先搜索池中是否已经有可适用的，如果有，factory只是简单返回送出这个对象，否则，创建一个新的对象，加入到池中，再返回送出这个对象池。”

```
public class FlyweightFactory {  
    //Flyweight pool  
    private Hashtable flyweights = new Hashtable();  
    public Flyweight getFlyweight( Object key ) {  
        Flyweight flyweight = (Flyweight) flyweights.get(key);  
        if( flyweight == null ) {  
            //产生新的ConcreteFlyweight  
            flyweight = new ConcreteFlyweight();  
            flyweights.put( key, flyweight );  
        }  
        return flyweight;  
    }  
}
```

大B：“到现在为止， Flyweight模式的基本框架已经就绪,我们就来看看如何调用。”

```
FlyweightFactory factory = new FlyweightFactory();  
Flyweight fly1 = factory.getFlyweight( "Fred" );  
Flyweight fly2 = factory.getFlyweight( "Wilma" );  
.....
```

大B：“从调用上看,好象是个纯粹的Factory使用,但奥妙就在于Factory的内部设计上。”

12.10 享元模式的优缺点

小A：“享元模式有什么优点和缺点？”

大B：“1、享元模式使得系统更加复杂。为了使对象可以共享，需要将一些状态外部化，这使得程序的逻辑复杂化。2、享元模式将享元对象的状态外部化，而读取外部状态使得运行时间稍微变长。”

12.11 Flyweight模式在XML等数据源中应用

大B：“我们上面已经提到，当大量从数据源中读取字符串，其中肯定有重复的，那么我们使用Flyweight模式可以提高效率，以唱片CD为例，在一个XML文件中，存放了多个CD的资料。”

每个CD有三个字段：

- 1、出片日期(year)
- 2、歌唱者姓名等信息(artist)
- 3、唱片曲目 (title)

其中，歌唱者姓名有可能重复，也就是说，可能有同一个演唱者的多个不同时期不同曲目的CD。我们将‘歌唱者姓名’作为可共享的ConcreteFlyweight。其他两个字段作为UnsharedConcreteFlyweight。

首先看看数据源XML文件的内容：

1978

Eno, Brian

1950

Holiday, Billie

1977

Eno, Brian

.....

虽然上面举例CD只有3张,CD可看成是大量重复的小类,因为其中成分只有三个字段,而且有重复的(歌唱者姓名)。

CD就是类似上面接口 Flyweight:

```
public class CD {
    private String title;
    private int year;
    private Artist artist;
    public String getTitle() {    return title;  }
    public int getYear() {        return year;   }
    public Artist getArtist() {    return artist;  }
    public void setTitle(String t){    title = t;}
    public void setYear(int y){year = y;}
    public void setArtist(Artist a){artist = a;}
}
```

将"歌唱者姓名"作为可共享的ConcreteFlyweight:

```
public class Artist {
    //内部状态
    private String name;
    // note that Artist is immutable.
    String getName(){return name;}
    Artist(String n){
        name = n;
    }
}
```

再看看Flyweight

factory, 专门用来制造上面的可共享的

ConcreteFlyweight:Artist

```

public class ArtistFactory {
    Hashtable pool = new Hashtable();
    Artist getArtist(String key){
        Artist result;
        result = (Artist)pool.get(key);
        ////产生新的Artist
        if(result == null) {
            result = new Artist(key);
            pool.put(key,result);
        }
        return result;
    }
}

```

当你有几千张甚至更多CD时,Flyweight模式将节省更多空间,共享的flyweight越多,空间节省也就越大。

给个例子, coffee商店

```

package FlyWeight.coffeeshop;

public class Table {
    private int number;
    public int getNumber() {
        return number;
    }
    public void setNumber(int number) {
        this.number = number;
    }
    public Table(int number) {

```

```

super();
// TODO Auto-generated constructor stub
this.number = number;
}
}

package FlyWeight.coffeeshop;

public abstract class Order {
    public abstract void serve(Table table);
    public abstract String getFlavor();
}

package FlyWeight.coffeeshop;

public class Flavor extends Order {
    private String flavor;

    public Flavor(String flavor) {
        super();
        // TODO Auto-generated constructor stub
        this.flavor = flavor;
    }

    public String getFlavor() {
        return flavor;
    }

    public void setFlavor(String flavor) {
        this.flavor = flavor;
    }

    public void serve(Table table) {
        System.out.println("Serving table " + table.getNumber() + " with flavor " + flavor );
    }
}

package FlyWeight.coffeeshop;

public class FlavorFactory {
    private Order[] flavors = new Flavor[10];
    private int ordersMade = 0; //已经处理好的订单数

```

```
private int totalFlavors = 0; //已购买的coffee风味种类数
public Order getOrder(String flavorToGet){
    if(ordersMade > 0){
        for(int i=0; i
```

运行结果：

```
Serving table 0 with flavor Black Coffee
Serving table 1 with flavor Capucino
Serving table 2 with flavor Espresso
Serving table 3 with flavor Espresso
Serving table 4 with flavor Capucino
Serving table 5 with flavor Capucino
Serving table 6 with flavor Black Coffee
Serving table 7 with flavor Espresso
Serving table 8 with flavor Capucino
Serving table 9 with flavor Black Coffee
Serving table 10 with flavor Espresso
```

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质
电子书下载！！！！

第四部分

构造型模式

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第十三章 可恶的皇帝——构造型模式

13.1 可恶的皇帝

时间：12月29日 地点：大B房间 人物：大B，小A

大B：“在遥远的过去，有这么一个与世无争的小村子，村里有一个村长(A)、村子之外有一个可恶的皇帝(E)和很多的村民(Bs)。在这个小村子，发生了许许多多的故事。”

小A：“很多的故事？嘿嘿！我最喜欢听故事了，说来听听。”

大B：“皇帝要让所有的村民交租，他要经历下面的流程：他首先跑到村民b1那里收租，村民b1的家里只有门，他就从门进入。他又跑到村民b2那里收租，村民b2家里只有窗，没有门，他就从窗进入。村民b3家里没有门，也没有窗，皇帝只好采用直升飞机空降的方式进入。终于有一天，皇帝再也受不了了，他把村长叫过来，对他说：以后我收租，只找你一个，从门进入，我收谁的你就跑腿。从此之后，可怜的村长就成了跑腿的人。皇帝很高兴，他把自己的创意称之为FACADE。时光如梭，老村长不停的跑腿，终于累死了。于是村民b3当选为村长。这年，皇帝过来收租，发现怎么都进不去村长的家里：村长家里没有门！皇帝想，‘妈的，总不成每次都让我用直升飞机空降来收租吧？’邪恶的皇帝又想了一个主意，他对村民b4

说：‘你在村长家外面造一个房子，把他的家全部包围起来，盖一个又宽又大的门，以后我找村长就找你了’ 皇帝很高兴，他又不用跑腿了，他把自己的想法称之为Adapter，而村长就成了Adaptee时间就像小风一样过着，村长不停的换，而邪恶的皇帝却始终活着，但是他已经厌倦了每次为新的村长找一个Adapter。他又开始思考了。他发现村里面的村民bx和自己一样的长寿。于是他改变了自己的策略，他让bx做这样的事情：准备好村长，准备好门。每次收租的时候，他都只需要去bx那里问一下：‘现在的村长是谁？’ 皇帝又胜利了，他把自己的方法称之为Bridge。有一天，邻国的女皇想到这个小村子里面参观。皇帝一想，‘坏了，这些村民个个连衣服都买不起，光着屁股，我国的威严何在？’ 于是，皇帝把秘书d叫过来，对他说：‘你给每个村民一件漂亮的衣服，别让他们给我丢脸！’ 最后，邻国的女皇看到的全是穿着漂亮衣服的村民。皇帝很高兴，于是他把自己的方法叫做Decorator。村子越来越大，村民越来越多。终于有一天，村子分裂了，变成了两个村子。皇帝想，每次收租我都找两个村长，太麻烦了！于是邪恶的皇帝又有了点子：在村子之上设立乡政府，在乡政府之上设立县政府…… 于是不管将来有多少的村民，自己都很方便管理，他把自己的方法叫做 Composite。终于有一天，皇帝有了自己的王国，村门很多很多，管理起来太过于复杂。皇帝每天要处理每个村民的事情，忙的头昏脑涨的。于是，邪恶的皇帝又有点子了，他成立了一个特殊部门‘部长’，然后又制定了惩罚规定，叫做‘拘留，坐牢，流放，砍头’。每当有一个村民发生问题的时候，皇帝就问‘部长’：他的问题怎么办？部长说：坐牢。又有一个村民发生了问题，皇帝问部长：怎么半？部长说：他的问题以前的不行，我又发明了一种新的处理方法，叫做‘凌迟’。皇帝很高兴，自己终于又可以轻轻松松的管理国家了。他把自己的方法称之为Flyweight。皇帝继续做着自己的美梦。他越来越依赖于自己的宠臣太监t了，不管有什么问题，他都问t，然后t去处理。他问t:我们有多少国民阿？t说，1000万。他问t说，我们有多少收入阿，t说

1000万。其实t已经大权独揽，自己腰包里面赚了无数的10000万了。t自己偷偷的大笑：哈哈，我就是传说中的Proxy阿!!”

13.2 构造器

小A：“什么是构造器？”

大B：“首先要注意的是Java的构造器并不是函数，所以他并不能被继承，这在我们extends的时候写子类的构造器时比较的常见，即使子类构造器参数和父类的完全一样，我们也要写super就是因为这个原因。构造器的修饰符比较的有限，仅仅只有public private protected这三个，其他的例如任何修饰符都不能对其使用，也就是说构造器不允许被成名成抽象、同步、静态等等访问限制以外的形式。

因为构造器不是函数，所以它是没有返回值的，也不允许有返回值。但是这里要说明一下，构造器中允许存在return语句，但是return什么都不返回，如果你指定了返回值，虽然编译器不会报出任何错误，但是JVM会认为他是一个与构造器同名的函数罢了，这样就会出现一些莫名其妙的无法找到构造器的错误，这里是要加倍注意的。”

小A：“在我们extends一个子类的时候经常会出现一些意想不到的问题，你能和我说说一些和构造器有关的吗？”

大B：“首先说一下Java在构造实例时的顺序（不讨论装载类的过程），构造的粗略过程如下 1、分配对象空间，并将对象中成员初始化为0或者空，Java不允许用户操纵一个不定值的对象。2、执行属性值的显式初始化（这里有一点变化，一会

解释，但大体是这样的) 3、执行构造器 4、将变量关联到堆中的对象上。”

小A：“能介绍一下准备知识吗？以备一会来详细说明这个的流程。”

大B：“this() super()是你如果想用传入当前构造器中的参数或者构造器中的数据调用其他构造器或者控制父类构造器时使用的，在一个构造器中你只能使用 this()或者super()之中的一个，而且调用的位置只能在构造器的第一行，在子类中如果你希望调用父类的构造器来初始化父类的部分，那就用合适的参数来调用 super()，如果你用没有参数的super()来调用父类的构造器（同时也没有使用 this()来调用其他构造器），父类缺省的构造器会被调用，如果父类没有缺省的构造器，那编译器就会报一个错误，注意这里，我们经常在继承父类的时候构造器中并不写和父类有关的内容，此时如果父类没有缺省构造器，就会出现编译器添加的缺省构造器给你添麻烦的问题了哦。例如：Class b extends a{public b(){} } 就没有任何有关父类构造器的信息，这时父类的缺省构造器就会被调用。”

举个SL-275中的例子

```
public class Manager extends Employee {
    private String department;
    public Manager(String name, double salary, String dept)
    {
        super(name, salary);
        department = dept;
    }
    public Manager(String n, String dept) {
        super(name);
        department = dept;
    }
}
```

```
public Manager(String dept) { // 这里就没有super(), 编译器会自动地添加一个空参数的缺省
    super构造器, 此时如果Employee类中没有空参数的缺省构造器, 那就会导致一个编译错误
    department = d;
}
}
```

大B：“你必须在构造器的第一行放置super或者this构造器，否则编译器会自动地放一个空参数的super构造器的，其他的构造器也可以调用super或者this，调用成一个递归构造链，最后的结果是父类的构造器（可能有多级父类构造器）始终在子类的构造器之前执行，递归的调用父类构造器。在具体构造类实例的过程中，上边过程的第二步和第三步是有一些变化的，这里的顺序是这样的，分配了对象空间及对象成员初始化为默认值之后，构造器就递归的从继承树由根部向下调用，每个构造器的执行过程是这样的：1、Bind构造器的参数2、如果显式的调用了this，那就递归调用this构造器然后跳到步骤5。3、递归调用显式或者隐式的父类构造器，除了Object以外，因为它没有父类4、执行显式的实例变量初始化（也就是上边的流程中的第二步，调用返回以后执行，这个步骤相当于在父构造器执行后隐含执行的，看样子像一个特殊处理）5、执行构造器的其它部分。”

小A：“好像有点明白了。”

大B：“这里的步骤很重要哦！从这个步骤中可以很明显的发现这个实例初始化时的递归调用过程。”

13.3 使用构造器中的注意事项

小A：“在使用构造器中的要注意哪些事项？”

大B：“1、构造器中一定不要创建自身的实例，否则会造成调用栈溢出错误。这个规则也适用于对象的实例变量，如果对象中有自身的引用，这个引用一定不能在定义中或者构造器中初始化。”

```
class a
{
    a _a = new a();
    public a()
    {
        _a = new a();
        a _b = new a();
    }
}
```

大B：“以上三种情况都会造成栈溢出，呵呵，这样会造成一个无穷递归的调用栈。2、如果父类是一个抽象类，那通过调用父类的构造器，也可以将它初始化，并且初始化其中的数据。3、如果你要在构造器中调用一个方法时，将该方法声明为private。对于这个规则是需要一些说明的，假使你的父类构造器中要调用一个非静态方法，而这个方法不是private的又被子类所重载，这样在实际创建子类的过程中递归调用到了父类的构造器时，父类构造器对这个方法的调用就会由于多态而实际上调用了子类的方法，当这个子类方法需要用到子类中实例变量的时候，就会由于变量没有初始化而出现异常（至于为什么子类中的实例变量没有初始化可以参考上边的实例初始化过程），这是Java不想看到的情况。而当父类构造器中调用的方法是一个private方法时，多态就不会出现，也就不会出现父类构造器调用子类

方法的情况，这样可以保证父类始终调用自己的方法，即使这个方法中调用了父类中的实例变量也不会出现变量未初始化的情况（变量初始化总是在当前类构造器主体执行之前进行）。”

13.4 理解构造器--构造器和方法的区别

大B：“要学习Java，你必须理解构造器。因为构造器可以提供许多特殊的方法，这个对于初学者经常混淆。但是，构造器和方法又有很多重要的区别。”

小A：“那它的功能和作用的有什么不同？”

大B：“功能和作用的不同下面我就详细给你讲一下。构造器是为了创建一个类的实例。这个过程也可以在创建一个对象的时候用到：`Platypus p1 = new Platypus();` 相反，方法的作用是为了执行Java代码。 修饰符，返回值和命名的不同。”

小A：“构造器和方法在这里有什么区别？”

大B：“修饰符，返回值，命名。和方法一样，构造器可以有任何访问的修饰：`public`，`protected`，`private`或者没有修饰（通常被`package` 和 `friendly`调用）不同于方法的是，构造器不能有以下非访问性质的修饰：`abstract`，`final`，`native`，`static`，或者 `synchronized`。 返回类型也是非常重要的。方法能返回任何类型的值或者无返回值（`void`），构造器没有返回值，也不需要`void`。”

小A：“构造器和方法命名有什么不同？”

大B：“构造器使用和类相同的名字，而方法则不同。按照习惯，方法通常用小写字母开始，而构造器通常用大写字母开始。构造器通常是一个名词，因为它和类名相同；而方法通常更接近动词，因为它说明一个操作。‘this’的用法，构造器和方法使用关键字this有很大的区别。方法引用this指向正在执行方法的类的实例。静态方法不能使用this关键字，因为静态方法不属于类的实例，所以this也就没有什么东西去指向。构造器的this指向同一个类中，不同参数列表的另外一个构造器。”

我们看看下面的代码：

```
public class Platypus {
    String name;
    Platypus(String input) {
        name = input;
    }
    Platypus() {
        this("John/Mary Doe");
    }
    public static void main(String args[]) {
        Platypus p1 = new Platypus("digger");
        Platypus p2 = new Platypus();
    }
}
```

大B：“在刚才讲的代码中，有2个不同参数列表的构造器。第一个构造器，给类的成员name赋值，第二个构造器，调用第一个构造器，给成员变量name一个初始值‘John/Mary Doe’。在构造器中，如果要使用关键字this,那么，必须放在第

一行，如果不这样，将导致一个编译错误。‘super’的用法，构造器和方法，都用关键字super指向超类，但是用的方法不一样。方法用这个关键字去执行被重载的超类中的方法。”

看下面的例子：

```
class Mammal {  
    void getBirthInfo() {  
        System.out.println("born alive.");  
    }  
}  
  
class Platypus extends Mammal {  
    void getBirthInfo() {  
        System.out.println("hatch from eggs");  
        System.out.print("a mammal normally is ");  
        super.getBirthInfo();  
    }  
}
```

大B：“从我们刚才讲的例子中，使用super.getBirthInfo()去调用超类Mammal中被重载的方法。构造器使用super去调用超类中的构造器。而且这行代码必须放在第一行，否则编译将出错。”

看下面的例子：

```
public class SuperClassDemo {  
    SuperClassDemo() {}  
}
```

```
class Child extends SuperClassDemo {  
    Child() {  
        super();  
    }  
}
```

大B：“在上面这个没有什么实际意义的例子中，构造器 `Child()` 包含了 `super`，它的作用就是将超类中的构造器 `SuperClassDemo` 实例化，并加到 `Child` 类中。”

小A：“编译器怎样自动加入代码？”

大B：“编译器自动加入代码到构造器，对于这个，Java程序员新手可能比较混淆。当我们写一个没有构造器的类，编译的时候，编译器会自动加上一个不带参数的构造器。”

例如：

```
public class Example {}
```

编译后将如下代码：

```
public class Example {  
    Example() {}  
}
```

在构造器的第一行，没有使用 `super`，那么编译器也会自动加上，例如：

```
public class TestConstructors {  
    TestConstructors() {}  
}
```

```
}
```

编译器会加上代码，如下：

```
public class TestConstructors {  
    TestConstructors() {  
        super;  
    }  
}
```

仔细想一下，就知道下面的代码

```
public class Example {}
```

经过会被编译器加代码形如：

```
public class Example {  
    Example() {  
        super;  
    }  
}
```

继承

构造器是不能被继承的。子类可以继承超类的任何方法。看看下面的代码：

```
public class Example {  
    public void sayHi {
```

```
system.out.println("Hi");  
}  
Example() {}  
}  
public class SubClass extends Example {  
}
```

大B：“我们来注意一下Java功能语句。修饰 不能用abstract, final, native, static, or synchronized 能 返回类型, 没有返回值, 没有void , 有返回值, 或者void , 命名 和类名相同;通常为名词, 大写开头, 通常代表一个动词的意思, 小写开头, this 指向同一个类中另外一个构造器, 在第一行指向当前类的一个实例, 不能用于静态方法, super 调用父类的构造器, 在第一行 调用父类中一个重载的方法。继承, 构造器不能被继承, 方法可以被继承 , 编译器自动加入一个缺省的构造器, 自动加入 (如果没有) 不支持 。编译器自动加入一个缺省的调用到超类的构造器, 自动加入 (如果没有) 不支持。”

本书由「ePUBw.COM」整理, ePUBw.COM 提供最新最全的优质电子书下载!!!

第十四章 汽车组装——生成器模式

14.1 汽车组装

时间：12月30日 地点：大B房间 人物：大B，小A

小A：“师兄，汽车是由哪些部件组装成的啊？”

大B：“假定汽车由三个部件组装而成：车身、引擎和轮胎，每个部件的制造以及最后的组装过程都很复杂。一个品牌汽车销售商一般不会自己来完成这些繁琐的过程，而是把它交给汽车零部件制造商。实际上有许多汽车零部件制造商，可以完成不同部件的生产和组装过程。”

小A：“那么，怎么获得一辆汽车的呢？”

大B：“首先，客户需要选定一个汽车制造商（比如某品牌），然后在选择一个汽车销售商（经纪人）为我们服务。然后，我们只需要告诉销售商我们需要哪个制造商的汽车，接下来，由销售商来代替我们监督汽车制造商一步一步为我们生产所需要的汽车（可以考虑订单式生产方式或对某部分特殊定制，如加长车身）。汽车生产完成后，由汽车制造厂提车就可以了。”

14.2 生成器模式

汽车,还有很多部件:车轮、方向盘、发动机还有各种小零件等等,部件很多,但远不止这些,如何将这些部件装配成一辆汽车,这个装配过程也很复杂(需要很好的组装技术),Builder模式就是为了将部件和组装过程分开。

小A:“什么是生成器模式啊?”

大B:“1、当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。2、当构造过程必须允许被构造的对象有不同的表示时。”

小A:“能不能说得简单一点?”

大B:“简单的说,它有点像工厂模式,但是最终生成‘产品’的是Director而非Factory,Director可以使用的builder来生成产品。而builder——生成器则遵循统一的接口,实现不同的内容,从而达到将一个复杂对象的构建与它的表示分离的目标。”

14.3 房屋

大B:“这样吧,我给你举个使用构建房屋的场景来说明‘生成器’吧!首先,这是我们最终需要生成的产品——房屋,它具有房间数量和门数量的属性。”

```
package com.alex.designpattern.builder;
```

```

/**
 * 最终我们需要的产品——房屋
 */
public class House {
    int roomNumber;
    int doorNumber;
    public House() {
        roomNumber = 0;
        doorNumber = 0;
    }
    public int getRoomNumber() {
        return roomNumber;
    }
    public int getDoorNumber() {
        return doorNumber;
    }
}

```

大B：“接下来就是房屋的真正构建者——‘生成器’的接口定义，以及它的一个实现。”

```

package com.alex.designpattern.builder;
/**
 * 房屋构建者的接口
 *
 */
public interface HouseBuilder {
    public void BuildRoom(int roomNo);
    public void BuildDoor(int room1, int room2);
    public House getHouse();
}

```



```

}
package com.alex.designpattern.builder;
public class ConcreteHouseBuilderA implements HouseBuilder {
private House house;
public ConcreteHouseBuilderA() {
house = new House();
}
public void BuildRoom(int roomNo) {
// you can create a new Room added to a House
house.roomNumber = house.roomNumber + roomNo;
}
public void BuildDoor(int room1, int room2) {
// you can create a new door associated with 2 room          // and added this door int
house.doorNumber = house.doorNumber + room1 + room2;
}
public House getHouse() {
return house;
}
}
}

```

大B：“这就是所谓的Director——最终构建房屋的‘表示者’。我们需要给它提供‘生成器’，然后由它来构建房屋。”

```

package com.alex.designpattern.builder;
/**
 * 房屋(构建)的“表示”者，通过它我们可以对同一种构建采用不同的表示方式
 *
 */
public class HouseDirector {
public static House CreateHouse(HouseBuilder concreteBuilder) {

```

```

concreteBuilder.BuildRoom(1);
concreteBuilder.BuildRoom(2);
concreteBuilder.BuildDoor(1, 2);
concreteBuilder.BuildDoor(2, 1);
return concreteBuilder.getHouse();
}
}

```

大B：“最后，当然是我们的测试启动类了，可以看到，使用生成器模式的简单过程就是：1、创建生成器对象。2、表示者使用此生成器对象构建最终产品。”

```

package com.alex.designpattern.builder;

/**
 * A test client to create a house
 *
 * but we do not know how the room and door be created
 *
 * * Builder ( 生成器模式 ) *
 *
 * 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。
 */
public class Test {
    public static void main(String[] args) {
        ConcreteHouseBuilderA myHouseBuilder = new ConcreteHouseBuilderA();
        House myHouse = HouseDirector.CreateHouse(myHouseBuilder);
        System.out.println("My house has room: " + myHouse.getRoomNumber());
        System.out.println("My house has door: " +

```

```
myHouse.getDoorNumber()); } }
```

14.4 为何使用生成器模式?

小A：“为什么要使用生成器模式？”

大B：“是为了将构建复杂对象的过程和它的部件解耦。注意：是解耦过程和部件。因为一个复杂的对象，不但有很多大量组成部分，如汽车，有很多部件。车轮、方向盘、发动机还有各种小零件等等，部件很多，但远不止这些，如何将这些部件装配成一辆汽车，这个装配过程也很复杂(需要很好的组装技术)，Builder模式就是为了将部件和组装过程分开。”

14.5 如何使用生成器模式?

小A：“那应该如何使用生成器模式?”

大B：“首先假设一个复杂对象是由多个部件组成的，Builder模式是把复杂对象的创建和部件的创建分别开来，分别用Builder类和Director类来表示。”

首先，需要一个接口，它定义如何创建复杂对象的各个部件：

```
public interface Builder {  
    //创建部件A    比如创建汽车车轮  
    void buildPartA();
```

```

//创建部件B 比如创建汽车方向盘
void buildPartB();
//创建部件C 比如创建汽车发动机
void buildPartC();
//返回最后组装成品结果（返回最后装配好的汽车）
//成品的组装过程不在这里进行,而是转移到下面的Director类中进行.
//从而实现了解耦过程和部件
Product getResult();
}

```

用Director构建最后的复杂对象，而在上面Builder接口中封装的是如何创建一个部件(复杂对象是由这些部件组成的)，也就是说Director的内容是如何将部件最后组装成成品：

```

public class Director {
    private Builder builder;
    public Director( Builder builder ) {
        this.builder = builder;
    }
    // 将部件partA partB partC最后组成复杂对象
    //这里是将车轮 方向盘和发动机组装成汽车的过程
    public void construct() {
        builder.buildPartA();
        builder.buildPartB();
        builder.buildPartC();
    }
}

```

Builder的具体实现ConcreteBuilder，通过具体完成接口Builder来构建或

装配产品的部件，定义并明确它所要创建的是什么具体东西，提供一个可以重新获取产品的接口。

```
public class ConcreteBuilder implements Builder {  
    Part partA, partB, partC;  
    public void buildPartA() {  
        //这里是具体如何构建partA的代码  
    };  
    public void buildPartB() {  
        //这里是具体如何构建partB的代码  
    };  
    public void buildPartC() {  
        //这里是具体如何构建partB的代码  
    };  
    public Product getResult() {  
        //返回最后组装成品结果  
    };  
}
```

复杂对象:产品Product:

```
public interface Product { }
```

复杂对象的部件:

```
public interface Part { }
```

我们看看如何调用Builder模式：

```
ConcreteBuilder builder = new ConcreteBuilder();  
Director director = new Director( builder );  
director.construct();  
Product product = builder.getResult();
```

14.6 生成器模式的应用

小A：“生成器模式的应如何去应用？”

大B：“在Java实际使用中，我们经常用到‘池’（Pool）的概念，当资源提供者无法提供足够的资源，并且这些资源需要被很多用户反复共享时，就需要使用池。‘池’实际是一段内存，当池中有一些复杂的资源的‘断肢’（比如数据库的连接池，也许有时一个连接会中断），如果循环再利用这些‘断肢’，将提高内存使用效率，提高池的性能。修改Builder模式中Director类使之能诊断“断肢”断在哪个部件上，再修复这个部件。”

14.7 汽车制造

大B：“我给你举个汽车制造作为例子，你应该就可以明白生成器模式的应如何去应用了。以汽车制造作为例子。假定汽车由三个部件组装而成：车身、引擎和轮胎，每个部件的制造以及最后的组装过程都很复杂。一个品牌汽车销售商一般不会自己来完成这些繁琐的过程，而是把它交给汽车零部件制造商。实际上有许多汽车零部件制造商，可以完成不同部件的生产和组装过程。”

小A：“那么，客户是怎么获得一辆汽车的呢？”

大B：“首先，客户需要选定一个汽车制造商（比如某品牌），然后在选择一个汽车销售商（经纪人）为我们服务。然后，我们只需要告诉销售商我们需要哪个制造商的汽车，接下来，由销售商来代替我们监督汽车制造商一步一步为我们生产所需要的汽车（可以考虑订单式生产方式或对某部分特殊定制，如加长车身）。汽车生产完成后，由汽车制造厂提车就可以了。”

如图14-1汽车制造建模

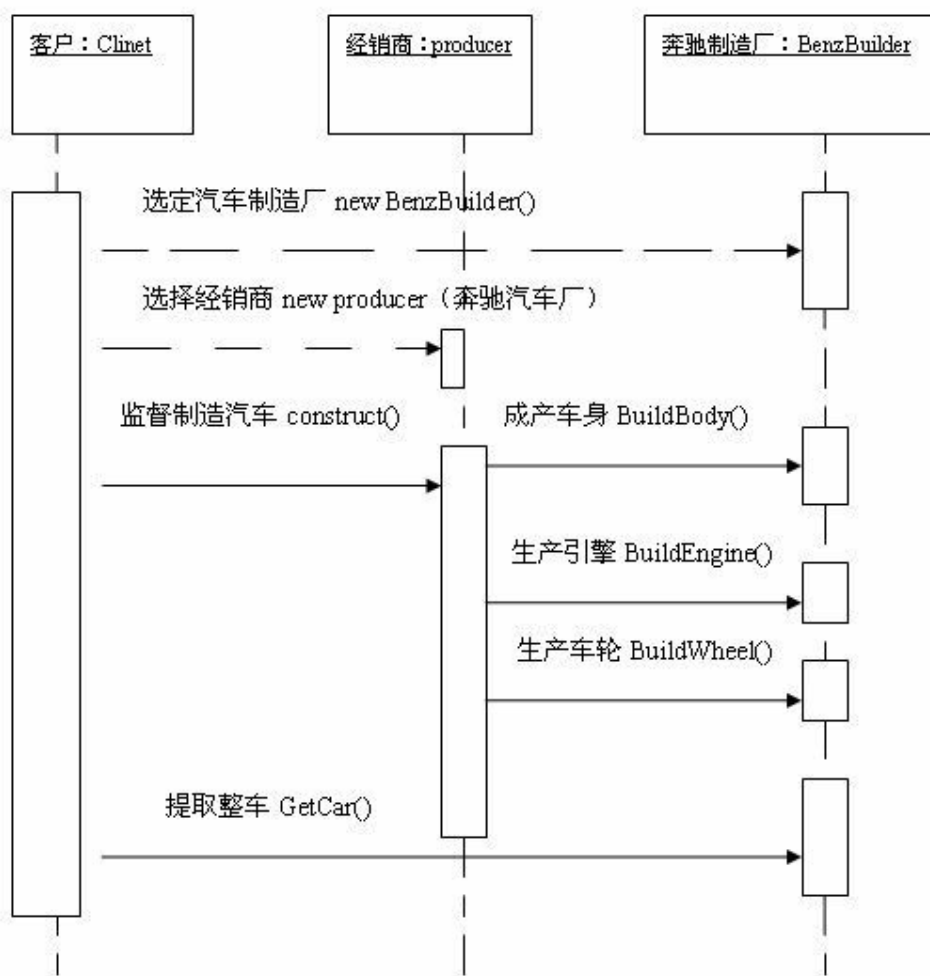


图14-1 汽车制造建模

用面向对象的方法对这个问题建模，得到的类图如图14-2汽车制造类图：

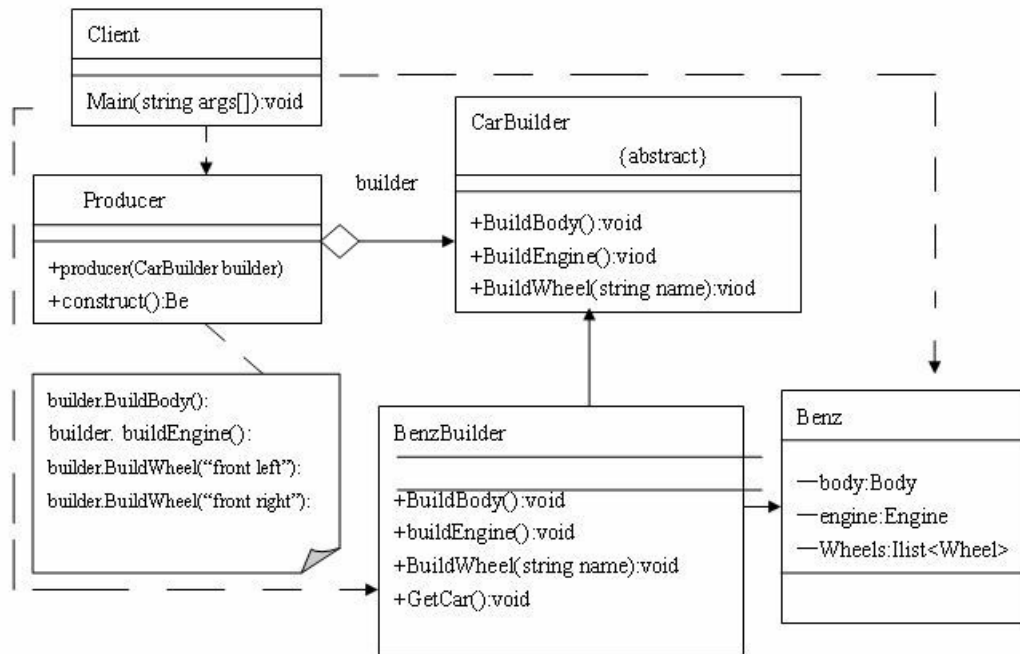


图14-2 汽车制造类图

程序代码如下：

```

namespace Builder
{
    //车身
    public class Body
    {
        private string name;
        public string Name
        {
            get { return name; }
            set { name = value; }
        }
        public Body(string name)
        {
            this.name = name;
        }
    }
}

```



```
}  
//引擎  
public class Engine  
{  
    private string name;  
    public string Name  
    {  
        get { return name; }  
        set { name = value; }  
    }  
    public Engine(string name)  
    {  
        this.name = name;  
    }  
}  
//车轮  
public class Wheel  
{  
    private string name;  
    public string Name  
    {  
        get { return name; }  
        set { name = value; }  
    }  
    public Wheel(string name)  
    {  
        this.name = name;  
    }  
}  
//Benz汽车  
public class Benz  
{
```

```

private Body body;
private Engine engine;
private IList wheels;
public void AddBody(Body body)
{
    this.body = body;
}
public void AddEngine(Engine engine)
{
    this.engine = engine;
}
public void AddWheel(Wheel wheel)
{
    if (wheels == null)
    {
        wheels = new List();
    }
    wheels.Add(wheel);
}
public void ShowMe()
{
    if ((this.body == null) || (this.engine == null) || (wheels == null))
    {
        Console.WriteLine("This car has NOT been completed yet!");
    }
    Else
    {
        Console.WriteLine("This is a car with a " + body.Name + " and a " + engine.Name + ".");
        Console.WriteLine("This car contains " + wheels.Count + " wheels:");
        for (int i = 0; i

```

小A：“这种类结构的设计，有什么好处呢？假设我们现在不想要刚才订的奔驰汽车了，而是想换成一辆路虎越野车怎么办？”

大B：“我们只需要告诉经销商，我们想改变汽车制造商就可以了！”

在上面的程序结构中，我们需要做的是：

1、 定义一个路虎汽车及其制造商，实现制造车身、引擎、轮胎和组装的方法：

```
//Benz汽车
public class Benz
{
    private Body body;
    private Engine engine;
    private IList wheels;
    //.....以下省略
}

public class LandRoverBuilder : CarBuilder
{
    private LandRover car;
    public override void BuildBody()
    {
        car = new Car();
        car.AddBody(new Body("business car body"));
    }
    public override void BuildEngine()
    {
        car.AddEngine(new Engine("benz engine"));
    }
}
```

```

public override void BuildWheel(string name)
{
    car.AddWheel(new Wheel(name));
}
//生产汽车，LandRover汽车装配过程
public LandRover GetCar()
{
    //do something like to assemble body and engine together.
    return car;
}
}

```

2、修改客户端调用，把LandRoverBuilder提供给Producer：

```

LandRoverBuilder builder = new LandRoverCarBuilder();
Producer director = new Producer(builder);
director.Construct();
LandRover car=builder.GetCar();
car.ShowMe();

```

OK，你就得到一辆路虎了！这就是GoF说的“将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示”的含义！用同样的方式，你甚至可以得到一辆拖拉机！

大B：“让我们来稍微扩展一下思路，讨论一下扩展问题。我们注意到，在产品这个层次上，每个制造厂直接依赖于具体的汽车，如BenzBuilder生成Benz，在客户端使用时也必须依赖于具体的汽车制造厂BenzBuilder和具体汽车Benz，在抽象制造厂CarBuilder定义中也没有包含GetCar的定义。”

小A：“这是为什么呢？”

大B：“我们想想把Benz这个汽车这个产品也抽象化，定义为Car，具体汽车Benz和LandRover从Car继承，在CarBuilder中定义GetCar方法依赖于抽象Car，从而在客户端也可以依赖于抽象。”

```
CarBuilder builder = new LandRoverCarBuilder();  
Producer director = new Producer(builder);  
director.Construct();  
Car car=builder.GetCar();  
car.ShowMe();
```

大B：“这种做法没错，你把这个结构图画出来就可以看出，这就成了一个典型的工厂模式！实际上，工厂模式和生成器模式是经常引起混淆和困扰的模式，不过仔细体会两种模式的意图就可以发现，他们关注的重点不同。工厂模式的重点在于产品的系列化的生成问题：工厂方法模式解决一种产品细节差异，抽象工厂模式解决多种产品的系列化创建——这些产品可以很简单，也可以比较复杂。但是，这些产品都是整体创建的。而生成器模式关注的是复杂产品的创建细节——如何一步一步的完成复杂产品部件的创建和最后的组装过程。这个创建复杂产品的过程可以差距巨大，所装配出来的产品也可以差距巨大。比如，上面的CarBuilder定义的抽象过程，只要进行合适的扩充，Producer通过使用具体的生成器就可以生产出小汽车、越野车、赛车、拖拉机，甚至任何由车身、引擎和轮胎可以组合出来的产品！正是由于这些产品之间的差异如此巨大，因此无法在抽象的CarBuilder中定义一个GetProduct之类的抽象方法。

我们再来看Producer这个类。它起到的是指导者的作用，指导生成器的使用方

法，也就是利用生成器一步步建造出产品的过程。这个过程一般来说是固定的。通过修改Construct()方法，就可以改变产品的建造过程。当然，如果一个产品的建造过程也是系统的变化因素，当然也可以利用类似工厂模式的方法对Producer进行抽象和封装。

最后我们来讨论一下CarBuilder提供给Producer的方式。生成器模式的核心是给督导者一个生成器，但是具体方式并没有限定。可以像本例这样使用聚合的方式，也可以直接把CarBuilder作为参数提供给Construct方法——这并没有什么本质的区别。”

14.8 生成器模式适用场景

小A：“师兄，生成器模式适用什么场景？”

大B：“1、当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。2、当构造过程必须允许被构造的对象有不同的表示时。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第十五章 运动协会——工厂方法模式

15.1 运动协会

时间：12月31日 地点：大B房间 人物：大B，小A

小A：“师兄，什么是运动协会？”

大B：“比如有个国家的运动员协会，他们是负责登记与注册职业运动员的（就好像我们国家的体育总局，呵呵，无论足球篮球还是乒乓球的运动员都必须在这里注册才能拿到我们国家职业运动员牌照）。一家体育俱乐部（比如篮球的广东宏远，足球的深圳健力宝）想获得球员为自己俱乐部效力，就必须通过这个运动员协会。”

小A：“怎样去实现‘运动员’接口？有几个客户端？”

大B：“根据DIP我们可以设计一个‘运动员’接口，‘足球运动员’和‘篮球运动员’（还有其他运动员）都实现‘运动员’这个接口。而‘运动员协会’就是一个简单工厂类，它负责实例化‘运动员’。我们这里的‘俱乐部’就是一个客户端（Client），不同的‘俱乐部’就是不同的客户端。对于不同的俱乐部对象（无论是八一还是深圳健力宝），他们都是面向‘运动员’接口编程，而不用管是‘足

球运动员’ 还是 ‘篮球运动员’ ，也就是说实现了 ‘运动员’ 接口的具体类 ‘足球运动员’ 无需暴露给客户端。这也满足了DIP。”

小A：“但具体的俱乐部（比如足球的深圳健力宝）如何确保自己获取的是自己想要的运动员（健力宝俱乐部需要的当然是足球运动员）呢？”

大B：“这就需要 ‘运动员协会’ 这一工厂类了。俱乐部通过调用 ‘运动员协会’ 的具体方法，返回不同的实例。这同时也满足了LoD，也就是 ‘深圳健力宝足球俱乐部’ 对象不直接与 ‘足球运动员：李毅’ 对象通信，而是通过他们共同的 ‘朋友’ —— ‘国家体育总局’ 通信。”

15.2 工厂方法模式

大B：“都知道Java最大的优点是它的完全OO化和它在多年的发展过程中吸收和总结了许多先进的框架与模式，其中工厂模式就是最常用的模式之一。”

小A：“师兄，能不能讲一下涉及到的OO原则的定义？”

大B：“好的。OCP（开闭原则，Open-Closed Principle）：一个软件的实体应当对扩展开放，对修改关闭。我的理解是，对于一个已有的软件，如果需要扩展，应当在不需修改已有代码的基础上进行。DIP（依赖倒转原则，Dependence Inversion Principle）：要针对接口编程，不要针对实现编程。我的理解是，对于不同层次的编程，高层次暴露给低层次的应当只是接口，而不是它的具体类。

LoD（迪米特法则，Law of Demeter）：只与你直接的朋友通信，而避免和陌生人通信。众所周知类（或模块）之间的通信越少，耦合度就越低，从而更有利于我们

对软件的宏观管理。老子论‘圣人之治’有相同的思想，《老子》云：‘是以圣人之治，虚其心，实其腹，弱其志，常使民无知无欲。’，又云：‘小国寡民，邻国相望，鸡犬之声相闻，民至老死，不相往来。’。佩服我们的老祖宗，N千年前就想到了西方N千年后才想到的东西，同时也佩服《Java与模式》的作者阎宏，可以用中国传统哲学思想这么生动的说明这一软件设计原则。”

小A：“这么说来，工厂方法模式有什么意义啊？”

大B：“定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类当中。核心工厂类不再负责产品的创建，这样核心类成为一个抽象工厂角色，仅负责具体工厂子类必须实现的接口，这样进一步抽象化的好处是使得工厂方法模式可以使系统在不修改具体工厂角色的情况下引进新的产品。”

15.3 工厂方法模式角色

小A：“工厂方法模式涉及到哪些角色？”

大B：“工厂方法模式的角色有：抽象工厂（Creator）角色、具体工厂（Concrete Creator）角色、抽象产品（Product）角色、具体产品（Concrete Product）角色。

抽象工厂（Creator）角色：是工厂方法模式的核心，与应用程序无关。任何在模式中创建的对象工厂类必须实现这个接口。

具体工厂（Concrete Creator）角色：这是实现抽象工厂接口的具体工厂类，

包含与应用程序密切相关的逻辑，并且受到应用程序调用以创建产品对象。

有两个这样的角色：BulbCreator与TubeCreator。

抽象产品（Product）角色：工厂方法模式所创建的对象超类型，也就是产品对象的共同父类或共同拥有的接口。

具体产品（Concrete Product）角色：这个角色实现了抽象产品角色所定义的接口。某具体产品有专门的具体工厂创建，它们之间往往一一对应。”

如图15-1工厂方法模式所示

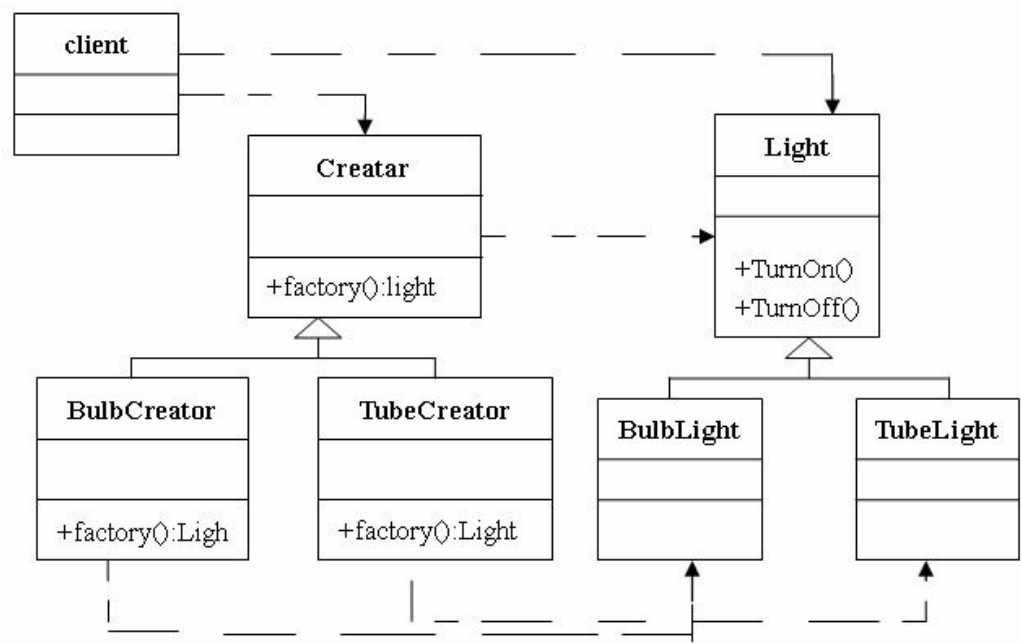


图15-1 工厂方法模式

一个简单的实例

```
// 产品 Plant接口
public interface Plant { }

//具体产品PlantA , PlantB
```

```
public class PlantA implements Plant {
    public PlantA () {
        System.out.println("create PlantA !");
    }
    public void doSomething() {
        System.out.println(" PlantA do something ...");
    }
}

public class PlantB implements Plant {
    public PlantB () {
        System.out.println("create PlantB !");
    }
    public void doSomething() {
        System.out.println(" PlantB do something ...");
    }
}

// 产品 Fruit接口
public interface Fruit { }

//具体产品FruitA , FruitB
public class FruitA implements Fruit {
    public FruitA() {
        System.out.println("create FruitA !");
    }
    public void doSomething() {
        System.out.println(" FruitA do something ...");
    }
}

public class FruitB implements Fruit {
    public FruitB() {
        System.out.println("create FruitB !");
    }
    public void doSomething() {
```

```
System.out.println(" FruitB do something ...");
}
}
// 抽象工厂方法
public interface AbstractFactory {
    public Plant createPlant();
    public Fruit createFruit() ;
}
//具体工厂方法
public class FactoryA implements AbstractFactory {
    public Plant createPlant() {
        return new PlantA();
    }
    public Fruit createFruit() {
        return new FruitA();
    }
}
public class FactoryB implements AbstractFactory {
    public Plant createPlant() {
        return new PlantB();
    }
    public Fruit createFruit() {
        return new FruitB();
    }
}
```

15.4 为何使用?

小A：“为什么会使用工厂模式？”

大B：“工厂模式是我们最常用的模式了，著名的Jive论坛，就大量使用了工厂模式，工厂模式在Java程序系统可以说是随处可见。”

15.5 为什么工厂模式是如此常用？

小A：“为什么工厂模式是如此常用？”

大B：“因为工厂模式就相当于创建实例对象的new，我们经常要根据类Class生成实例对象，如A a=new A() 工厂模式也是用来创建实例对象的，所以以后new时就要多个心眼，是否可以考虑实用工厂模式，虽然这样做，可能多做一些工作，但会给你系统带来更大的可扩展性和尽量少的修改量。我们以类Sample为例，如果我们要创建Sample的实例对象：Sample sample=new Sample();可是，实际情况是，通常我们都要在创建sample实例时做点初始化的工作，比如赋值查询数据库等。首先，我们想到的是，可以使用Sample的构造函数，这样生成实例就写成：Sample sample=new Sample(参数);但是，如果创建sample实例时所做的初始化工作不是象赋值这样简单的事，可能是很长一段代码，如果也写入构造函数中，那你的代码很难看了（就需要Refactor重整）。”

15.6 为什么说代码很难看

小A：“为什么说代码很难看。”

大B：“初学者可能没有这种感觉，我们分析如下，初始化工作如果是很长一段代码，说明要做的工作很多，将很多工作装入一个方法中，相当于将很多鸡蛋放在一个篮子里，是很危险的，这也是有背于Java面向对象的原则，面向对象的封装(Encapsulation)和分派(Delegation)告诉我们，尽量将长的代码分派‘切割’成每段，将每段再‘封装’起来(减少段和段之间耦合联系性)，这样，就会将风险分散，以后如果需要修改，只要更改每段，不会再发生牵一动百的事情。”

15.7 简单工厂模式与工厂方法模式大PK

小A：“什么是简单工厂模式？”

大B：“简单工厂模式又叫静态工厂模式，顾名思义，它是用来实例化目标类的静态类。”

小A：“简单工厂模式有什么优点？”

大B：“现在我就主要通过一个简单的实例说明简单工厂及其优点。”

小A：“嗯。好。”

大B：“如有个国家的运动员协会，他们是负责登记与注册职业运动员的（就好像我们国家的体育总局，呵呵，无论足球篮球还是乒乓球的运动员都必须在这里注册才能拿到我们国家职业运动员牌照）。一家体育俱乐部（比如篮球的广东宏远，足球的深圳健力宝）想获得球员为自己俱乐部效力，就必须通过这个运动员协会。根据DIP我们可以设计一个‘运动员’接口，‘足球运动员’和‘篮球运动员’（还

有其他运动员)都实现‘运动员’这个接口。而‘运动员协会’就是一个简单工厂类，它负责实例化‘运动员’。我们这里的‘俱乐部’就是一个客户端(Client)，不同的‘俱乐部’就是不同的客户端。对于不同的俱乐部对象(无论是八一还是深圳健力宝)，他们都是面向‘运动员’接口编程，而不用管是‘足球运动员’还是‘篮球运动员’，也就是说实现了‘运动员’接口的具体类‘足球运动员’无需暴露给客户端。这也满足了DIP。”

小A：“但具体的俱乐部(比如足球的深圳健力宝)如何确保自己获取的是自己想要的运动员(健力宝俱乐部需要的当然是足球运动员)呢？”

大B：“这就需要‘运动员协会’这一工厂类了。俱乐部通过调用‘运动员协会’的具体方法，返回不同的实例。这同时也满足了LoD，也就是‘深圳健力宝足球俱乐部’对象不直接与‘足球运动员：李毅’对象通信，而是通过他们共同的‘朋友’——‘国家体育总局’通信。”

下面给出各个类的程序。

Code: [Copy to clipboard]

运动员.java

```
public interface 运动员 {  
    public void 跑();  
    public void 跳();  
}
```

足球运动员.java

```
public class 足球运动员 implements 运动员 {  
    public void 跑(){  
        //跑啊跑  
    }  
}
```

```
public void 跳(){  
    //跳啊跳  
}  
}
```

篮球运动员.java

```
public class 篮球运动员 implements 运动员 {  
    public void 跑(){  
        //do nothing  
    }  
    public void 跳(){  
        //do nothing  
    }  
}
```

体育协会.java

```
public class 体育协会 {  
    public static 运动员 注册足球运动员(){  
        return new 足球运动员();  
    }  
    public static 运动员 注册篮球运动员(){  
        return new 篮球运动员();  
    }  
}
```

俱乐部.java

```
public class 俱乐部 {  
    private 运动员 守门员;  
    private 运动员 后卫;  
    private 运动员 前锋;  
    public void test() {  
        this.前锋 = 体育协会.注册足球运动员();  
        this.后卫 = 体育协会.注册足球运动员();  
        this.守门员 = 体育协会.注册足球运动员();  
        守门员.跑();  
    }  
}
```



```
后卫.跳();  
}  
}
```

大B：“这就是简单工厂模式的一个简单实例，你应该想象不用接口不用工厂而把具体类暴露给客户端的那种混乱情形吧？就好像没了体育总局，各个俱乐部在市场上自己胡乱的寻找仔细需要的运动员。简单工厂就解决了这种混乱。我们用OCP看看简单工厂，会发现如果要对系统进行扩展的话治需要增加实现产品接口的产品类（上例表现为‘足球运动员’，‘篮球运动员’类，比如要增加个‘乒乓球运动员’类），而无需对原有的产品类进行修改。”

小A：“这咋一看好像满足OCP。”

大B：“但是实际上还是需要修改代码的——对，就是修改工厂类。上例中如果增加‘乒乓球运动员’产品类，就必须相应的修改‘体育协会’工厂类，增加个‘注册乒乓球运动员’方法。所以可以看出，简单工厂模式是不满足OCP的。”

小A：“那工厂方法模式哩？”

大B：“我们刚刚讲了简单工厂模式，下面继续谈谈工厂方法模式。刚才点明了简单工厂模式最大的缺点——不完全满足OCP。为了解决这一缺点，设计师们提出了工厂方法模式。工厂方法模式和简单工厂模式最大的不同在于，简单工厂模式只有一个（对于一个项目或者一个独立模块而言）工厂类，而工厂方法模式有一组实现了相同接口的工厂类。下面我们通过修改刚才的实例来介绍工厂方法模式。我们在不改变产品类（‘足球运动员’类和‘篮球运动员’类）的情况下，修改下工厂类的结构。”

相关代码如下：

Code: [Copy to clipboard]

运动员.java

```
public interface 运动员 {  
    public void 跑();  
    public void 跳();  
}
```

足球运动员.java

```
public class 足球运动员 implements 运动员 {  
    public void 跑(){  
        //跑啊跑  
    }  
    public void 跳(){  
        //跳啊跳  
    }  
}
```

篮球运动员.java

```
public class 篮球运动员 implements 运动员 {  
    public void 跑(){  
        //do nothing  
    }  
    public void 跳(){  
        //do nothing  
    }  
}
```

体育协会.java

```
public interface 体育协会 {  
    public 运动员 注册();  
}
```

足球协会.java

```
public class 足球协会 implements 体育协会 {  
    public 运动员 注册(){  
        return new 足球运动员();  
    }  
}
```

篮球协会.java

```
public class 篮球协会 implements 体育协会 {  
    public 运动员 注册(){  
        return new 篮球运动员();  
    }  
}
```

俱乐部.java

```
public class 俱乐部 {  
    private 运动员 守门员;  
    private 运动员 后卫;  
    private 运动员 前锋;  
    public void test() {  
        体育协会 中国足协 = new 足球协会();  
        this.前锋 = 中国足协.注册();  
        this.后卫 = 中国足协.注册();  
        守门员.跑();  
        后卫.跳();  
    }  
}
```

大B：“很明显可以看到，‘体育协会’工厂类变成了‘体育协会’接口，而实现此接口的分别是‘足球协会’‘篮球协会’等等具体的工厂类。”

小A：“这样做有什么好处呢？”

大B：“很明显，这样做就完全OCP了。如果需要再加入（或扩展）产品类（比

如加多个‘乒乓球运动员’)的话就不再需要修改工厂类了，而只需相应的再添加一个实现了工厂接口（‘体育协会’接口）的具体工厂类。”

小A：“工厂方法模式是为了克服简单工厂模式的缺点（主要是为了满足OCP）而设计出来的。但是，工厂方法模式就一定比简单工厂模式好呢？”

大B：“不一定。1、结构复杂度。从这个角度比较，显然简单工厂模式要占优。简单工厂模式只需一个工厂类，而工厂方法模式的工厂类随着产品类个数增加而增加，这无疑会使类的个数越来越多，从而增加了结构的复杂程度。2、代码复杂度。代码复杂度和结构复杂度是一对矛盾，既然简单工厂模式在结构方面相对简洁，那么它在代码方面肯定是比工厂方法模式复杂的了。简单工厂模式的工厂类随着产品类的增加需要增加很多方法（或代码），而工厂方法模式每个具体工厂类只完成单一任务，代码简洁。3、客户端编程难度。工厂方法模式虽然在工厂类结构中引入了接口从而满足了OCP，但是在客户端编码中需要对工厂类进行实例化。而简单工厂模式的工厂类是个静态类，在客户端无需实例化，这无疑是个吸引人的优点。4、管理上的难度。这是个关键的问题。我们先谈扩展。众所周知，工厂方法模式完全满足OCP，即它有非常良好的扩展性。”

小A：“那是否就说明了简单工厂模式就没有扩展性呢？”

大B：“不是的。简单工厂模式同样具备良好的扩展性——扩展的时候仅需要修改少量的代码（修改工厂类的代码）就可以满足扩展性的要求了。尽管这没有完全满足OCP，但你认为不需要太拘泥于设计理论，要知道，sun提供的Java官方工具包中也有想到多没有满足OCP的例子啊（`Java.util.Calendar`这个抽象类就不满足OCP，具体原因大家可以分析下）。然后我们从维护性的角度分析下。假如某个具体产品类需要进行一定的修改，很可能需要修改对应的工厂类。当同时需要修改多个

产品类的时候，对工厂类的修改会变得相当麻烦（对号入座已经是个问题了）。反而简单工厂没有这些麻烦，当多个产品类需要修改是，简单工厂模式仍然仅仅需要修改唯一的工厂类。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第十六章 麦当劳的鸡腿—抽象工厂模式

16.1 麦当劳的鸡腿

时间：1月1日 地点：麦当劳 人物：大B和他的女朋友

这天大B和女朋友去麦当劳吃东西。

大B：“今天你想想去哪玩哩？”

大B的女朋友：“我们好久没去吃汉堡和鸡腿了。”

大B：“是喔！确实好久没吃过了。你想去麦当劳还是肯德基呢？”

大B的女朋友：“都好啦！”

大B：“那我们就去麦当劳吧！”

大B的女朋友：“嗯。好。”

大B：“走吧！”

来到麦当劳。

大B的女朋友：“我要吃鸡腿。”

大B：“好，和服务员说。”

大B的女朋友：“就和你说嘛！”

大B：“为什么。和服务员说：我要鸡腿，肯定不会给你拿来烤羊腿。”

大B的女朋友：“讨厌！哪有人这样的。”

16.2 抽象工厂模式

时间：1月1日 地点：大B房间 人物：大B，小A

大B：“我们可以认为麦当劳和肯德基就是生产食物的工厂，那么理所当然，汉堡和鸡腿是他们共同生产的两种食物，不管你去MDL还是KDJ，说：我要鸡腿，那肯定不会给你拿来烤羊腿。嘿嘿。这里，我们假定麦当劳和肯德基只生产这两种产品。我们是消费者，我们就是客户，就是产品的消费者，就是程序中对象的调用者。而麦当劳和肯德基，理所当然的，他们就是工厂，一个叫做麦当劳工厂，一个叫做肯德基工厂，他们是真正的生产者，而对于我们这些客户消费者（程序中对象的调用者）来说，不管是去麦当劳还是肯德基，我们都说一样的话（我们的要求是稳定的）：我要鸡腿。只要我们提出这个请求，那么肯定会得到我们想要的。而作为工厂（生产者），麦当劳和肯德基都生产鸡腿和汉堡，所以抽象出来的抽象工厂都具有生产鸡腿和生产汉堡的功能，这是接口中的两个方法。因为在这个接口中，还不知道到底要生产谁家的产品，所以只能返回个抽象的鸡腿或汉堡，等到麦当劳

或者肯德基工厂生产出来，就知道是谁家的了。（有标志嘛，哈哈，这就是动态创建对象）。管是麦当劳还是肯德基的鸡腿或汉堡，它的本质都是鸡腿或汉堡，所以可以抽象出来。那么鸡腿就派生出麦当劳的鸡腿和肯德基的鸡腿，而汉堡就派生出麦当劳的汉堡和肯德基的汉堡。而对于我们这些客户消费者（程序中对象的调用者）来说，不管是去麦当劳还是肯德基，我们都说一样的话（我们的要求是稳定的）：我要鸡腿。只要我们提出这个要求，那么肯定会得到我们想要的。不管是谁家的鸡腿，肯定是鸡腿不会是羊腿。所以，我们只要规定好是鸡腿（接口）就行了，而让工厂去绝对具体的制作过程。我们只伸手接过来一个鸡腿，狠狠的咬一口，恩，真香~：）

到现在为止，我们只和鸡腿（抽象的接口）还有抽象工厂（因为我们不管是麦当劳还是肯德基，我们只要鸡腿）打交道。你现在知不知道什么是抽象工厂模式？”

小A：“抽象工厂模式提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。抽象工厂（Abstract Factory）模式，又称工具箱（Kit 或 Toolkit）模式。”

<http://luchar.javaeye.com/blog/179617>

如图图16-1 标准抽象工厂模式所示

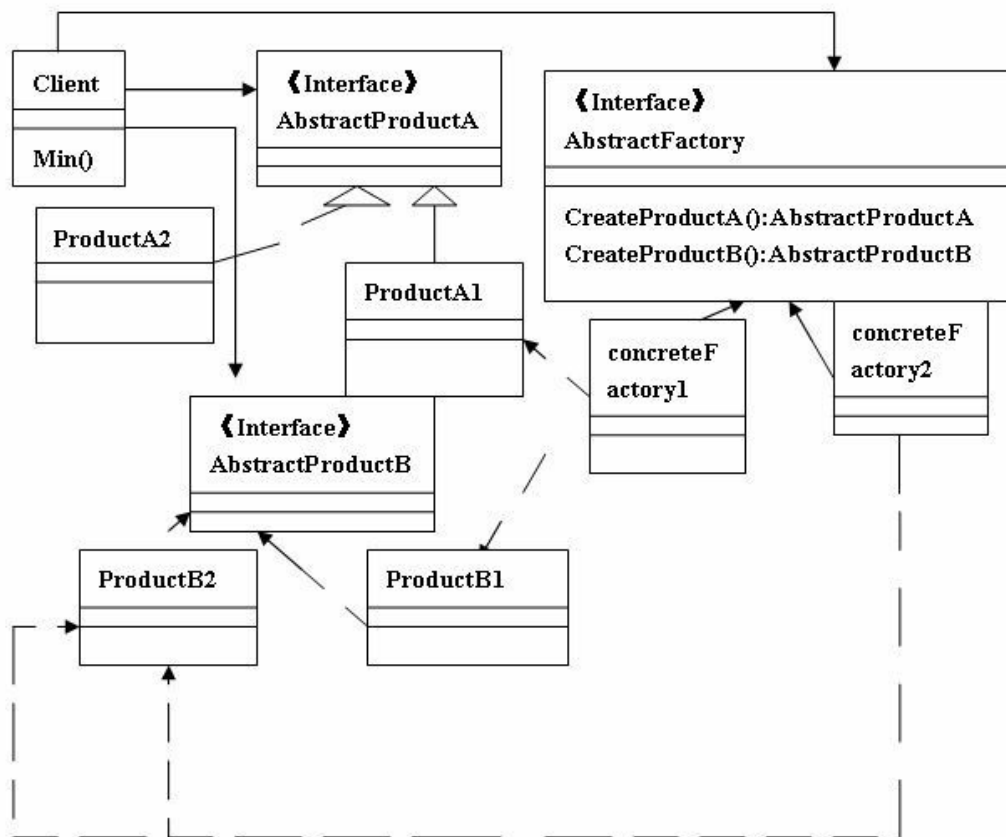


图16-1 标准抽象工厂模式

大B：“从模式定义中知道这个模式的意图内容为：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。工厂类层次的通信接口只有抽象工厂和创建产品族的各个工厂方法，这些工厂方法不带任何参数，并且返回具有抽象产品类型的具体产品实例。这些使得客户端可以不依赖具体产品的类，从而体现了模式的意图。意图中的‘而无需指定它们具体的类’可以理解为客户端在使用和创建具体产品时不给出具体产品的任何暗示。”

小A：“师兄，抽象工厂模式有什么动机啊？”

大B：“考虑一个支持多种视感 (l o o k - a n d - f e e l) 标准的用户界面工具包，例如M o t i f和Presentation Manager。不同的视感风格为诸如滚动条、窗口和按钮等用户界面‘窗口组件’定义不同的外观和行为。为保证视

感风格标准间的可移植性，一个应用不应该为一个特定的视感外观硬编码它的窗口组件。在整个应用中实例化特定视感风格的窗口组件类将使得以后很难改变视感风格。”

16.3 抽象工厂模式的角色

小A：“抽象工厂模式涉及到哪些角色？”

大B：“抽象工厂模式设计到以下的角色：

抽象工厂角色：担任这个角色的是工厂方法模式的核心，它是与应用系统的商业逻辑无关的。通常使用接口或抽象类实现。

具体工厂角色：这个角色直接在客户端的调用下创建产品的实例。这个角色含有选择合适的产品对象的逻辑，而这个逻辑是与应用系统的商业逻辑紧密相关的。通常使用具体的类实现。

抽象产品角色：担任这个角色的类是抽象工厂方法模式所创建的对象之父类，或它们共同拥有的接口。通常使用接口或抽象类实现这一角色。

具体产品角色：抽象工厂模式所创建的任何产品对象都是某一具体产品类的实例。这是客户端最终需要的东西。通常使用具体类实现这个角色。”

16.4 抽象工厂模式的使用

小A：“师兄，什么情况下使用抽象工厂模式？”

大B：“一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有形态的工厂模式都是重要的。这个系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。属于同一个产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来。系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现。”

16.5 抽象工厂模式的适用

小A：“抽象工厂模式的适用于哪些地方？”

大B：“一个系统要独立于它的产品的创建、组合和表示时。一个系统要由多个产品系列中的一个来配置时。当你要强调一系列相关的产品对象的设计以便进行联合使用时。当你提供一个产品类库，而只想显示它们的接口而不是实现时。”

小A：“能不能举些示例啊？”

大B：“有三种抽象的产品：墙、门、房间。对这三种抽象产品有两组具体实现：卧室和起居室。那么，我们的抽象工厂就可以根据客户的指令（即调用参数）去生产卧室和起居室的房间（墙和门包括在房间里）。”

16.6 抽象工厂模式的优点和缺点

大B：“你知道抽象工厂模式有什么优点吗？”

小A：“1、产品从客户代码中被分离出来。2、容易改变产品的系列。3、将一个系列的产品族统一到一起创建。”

大B：“它的缺点你知道吗？”

小A：“在产品族中扩展新的产品是很困难的，它需要修改抽象工厂的接口。”

16.7 抽象工厂模式是为了解决什么问题的呢？给了我们怎样的设计思路？

小A：“创建型模式抽象了对象实例化的过程，它帮助系统不依赖于对象如何创建，如何实现，何时创建。个类创建型模式使用继承使对象创建多样化，一个对象创建模式将对象的创建代理到其他类。那抽象工厂模式是为了解决什么问题的呢？给了我们怎样的设计思路？”

大B：“软件开发中我们经常会碰到一系列相关的对象需要创建，如果按照常规做法我们就要为不同的对象创建编写不同的代码，复用性和可维护性都降低了。而且这些相关对象创建的方式也许不同，那么客户代码创建的时候就要针对不同的对象编码，对象创建的方式还是一个容易改变的地方。基于这样的情况提出了抽象工厂模式，抽象工厂模式为创建一系列相关对象提供了统一的接口，客户只要调用这个接口即可，封装了变化，隔离了变化，让客户代码稳定起来。比如这样一个情况，我们做了一个桌面软件，这个软件的界面是可以改变的，它有几种风格：XP风

格、Win2000风格、苹果机风格。这个软件就是显示一个窗口，窗口有标题栏、滚动条，XP风格的界面有它自己的标题栏和滚动条，而苹果机风格的又不一样。”

小A：“我们常常怎么做？”

```
switch(type)
{
case "XP" :
setTitle(new XPTitle());
setScrollbar(new XPScrollbar());
break;
case "win2000" :
setTitle(new win2000Title());
setScrollbar(new win2000Scrollbar());
break;
case "macos" :
setTitle(new macosTitle());
setScrollbar(new macosScrollbar());
break;
}
```

小A：“这样做有什么坏处呢？”

大B：“这个例子太小实际上没有什么坏处，这样写可以。但是人总会出错，你看看下面这个。”

```
case "win2000" :
setTitle(new win2000Title());
setScrollbar(new XPScrollbar());
```

break;

大B：“你的界面将是Win2000的标题栏，XP风格的滚动条，这就造成了不一致性抽象工厂就是为了解决这种一系列相关对象创建工作的。如图16-2抽象工厂的类图”

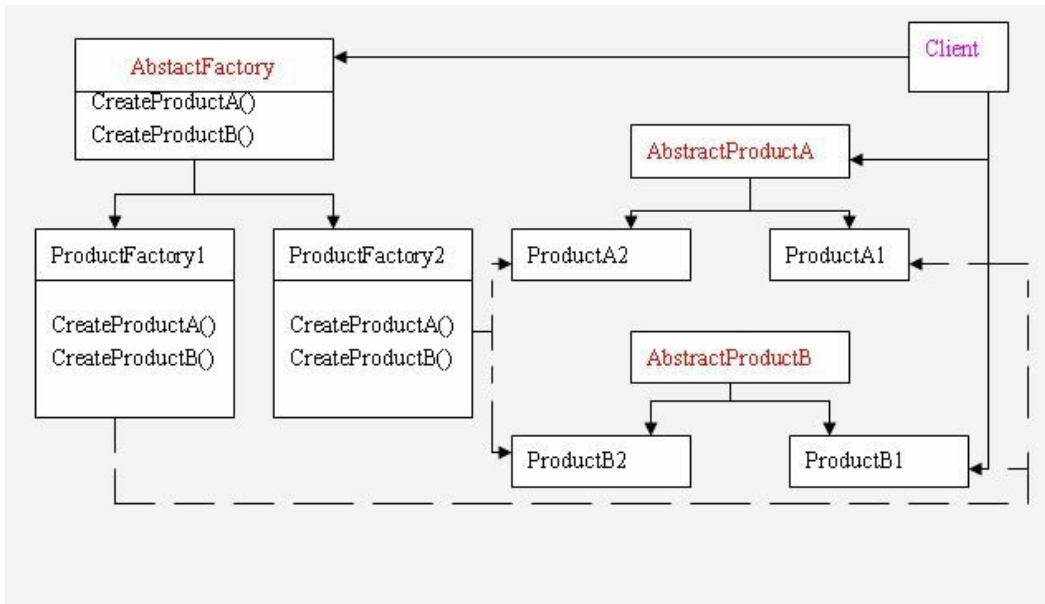


图16-2 抽象工厂的类图

大B：“在上面这个例子中标题栏和滚动条都是我们的产品，我们还应该写一个 **WindowManager** 类，专门来管理这些产品的创建的，而我们的WinXP团队、Win2000团队、苹果机团队实现这个 **WindowManager**，WinXP团队只会制造出XP风格的产品。”

16.8 “工厂模式”与“抽象工厂模式”的区别

小A：“‘工厂模式’与‘抽象工厂模式’有什么区别？”

大B：“在了解他们的区别之间有必要对两个概念做进一步的认识，那就是“产品族”和“产品等级”他们很容易区别在模式设计到底用那种工厂方法。”

小A：“什么是产品族呢？”

大B：“产品等级：简单的说就是不同类的产品就叫不同的产品族，比如电脑，苹果，桌子.....等就是一个产品族。他们的物理特性和外型都不相同。”

小A：“什么又是产品等级呢？”

大B：“产品族：就是同一种东西的不同类型。例如，PC有联想的，IBM，方正.....的等等，这些对象就构成一个产品等级。而抽象工厂模式解决的就是如果产生不同等级，不同产品族的产品（对象）结构模仿一个果园，假设果园应该能够出产水果，蔬菜，这时水果和蔬菜就是不同的东西，也即使是前面所说的产品等级；水果和蔬菜分别会有北方的，还会有塑料大棚里面生产的热带水果，蔬菜，这时的分类就是产品等级。而这样的结构就可以用文中提到的抽象工厂模式！”

16.9 奥迪audi车

大B：“你喜欢奥迪audi车吗？”

小A：“嘿嘿！喜欢。”

大B：“我就举个奥迪audi车的例子，让你更好地理解抽象工厂模式。”

小A：“好啊！”

大B：“介绍了工厂方法的使用，从那个程序中可以看到，奥迪audi车是从audi_car_factory_imple工厂中创建出来的，而大众3W车是从threeW_car_factory_imple工厂中创建出来的，那么如果这2家汽车生产大厂由总部在北京市，现在发展到上海，深圳等城市创建生气汽车的分厂，该怎么办？是不是得将原来的奥迪汽车工厂类。”

```
public class audi_car_factory_imple implements Icar_factory {
    public Icar_interface create_car() {
        car_audi_imple car_audi_imple_ref = new car_audi_imple();
        car_audi_imple_ref.setName("奥迪A6");
        car_audi_imple_ref.setSpeed(300);
        return car_audi_imple_ref;
    }
}
```

改成类似如下的模样：

```
public class audi_car_factory_imple implements Icar_factory {
    public Icar_interface create_car(String area_car) {
        if (area_car.equals("北京")){创建一个北京的奥迪汽车}
        if (area_car.equals("上海")){创建一个上海的奥迪汽车}
        if (area_car.equals("深圳")){创建一个深圳的奥迪汽车}
        return car_audi_imple_ref;
    }
}
```

小A：“发现一个问题，不同地域的汽车却在一个工厂中出现。”

大B：“这是不合乎常理的，因为北京奥迪在北京分厂创建，上海奥迪在上海分厂创建，这样才对。所以如果遇到分‘大系’来创建对象的时候，抽象工厂方法是肯定要使用的时候了。”

小A：“什么是‘大系’？”

大B：“这里的大系指的就是从地域上来分。这个例子就应该以‘用抽象工厂来定义具体工厂的抽象，而由具体工厂来创建对象’。比如在玩“极品飞车”这款游戏，每个地图处都有造车的工厂，每个造车的工厂都因为有造车的档次不同而划分为高级车厂，低级车厂，那么这样的场景正是应用抽象工厂的好时机，再来理解一下这句话“用抽象工厂来定义具体工厂的抽象，而由具体工厂来创建对象”，用抽象造车工厂来定义具体造车工厂的抽象，而由具体的造车工厂来创建汽车，这就是抽象工厂与工厂方法的不同，工厂方法中对象的创建是由工厂方法来确定的，创建的对象都是不分类并且实现一个接口的，而抽象工厂就是在工厂方法的基础上对创建车的对象的行为进行分类，比如北京车厂，上海车厂等。”

【抽象工厂模式解释】

类型：创建模式

提供一个创建一系列相关或相互依赖对象的接口，而无需指定他们具体的类。

【抽象工厂模式-Java代码实现】

新建抽象工厂接口：

```
package car_factory_interface;  
import car_interface.Icar_interface;
```

```

public interface Icar_factory {
    public Icar_interface create_threeW_car();
    public Icar_interface create_audi_car();
}

```

新建抽象工厂接口的高级车adv工厂实现类：

```

package car_factory_imple;
import car_factory_interface.Icar_factory;
import car_imple.car_3w_imple_adv;
import car_imple.car_audi_imple_adv;
import car_interface.Icar_interface;
public class car_factory_adv implements Icar_factory {
    public Icar_interface create_audi_car() {
        car_audi_imple_adv car_audi_imple_adv = new car_audi_imple_adv();
        car_audi_imple_adv.setName("奥迪A6");
        car_audi_imple_adv.setSpeed(300);
        return car_audi_imple_adv;
    }
    public Icar_interface create_threeW_car() {
        car_3w_imple_adv car_3w_imple_adv_ref = new car_3w_imple_adv();
        car_3w_imple_adv_ref.setName("大众A6");
        car_3w_imple_adv_ref.setSpeed(300);
        return car_3w_imple_adv_ref;
    }
}

```

新建抽象工厂接口的普通车low工厂实现类：

```

package car_factory_imple;

import car_factory_interface.Icar_factory;

import car_imple.car_3w_imple_low;

import car_imple.car_audi_imple_low;

import car_interface.Icar_interface;

public class car_factory_low implements Icar_factory {

    public Icar_interface create_audi_car() {

        car_audi_imple_low car_audi_imple_low_ref = new car_audi_imple_low();

        car_audi_imple_low_ref.setName("奥迪A6");

        car_audi_imple_low_ref.setSpeed(300);

        return car_audi_imple_low_ref;

    }

    public Icar_interface create_threeW_car() {

        car_3w_imple_low car_3w_imple_low_ref = new car_3w_imple_low();

        car_3w_imple_low_ref.setName("大众A6");

        car_3w_imple_low_ref.setSpeed(300);

        return car_3w_imple_low_ref;

    }

}

```

上面已经有抽象工厂和具体工厂的实现类了。

新建汽车接口：

```

package car_interface;

public interface Icar_interface {

    public void start();

    public void stop();

}

```

新建汽车父类：

```
package car_imple;

import car_interface.Icar_interface;

public class base_car_imple implements Icar_interface {

    private int speed;

    private String name;

    public int getSpeed() {

        return speed;

    }

    public void setSpeed(int speed) {

        this.speed = speed;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public void start() {

        // TODO Auto-generated method stub

    }

    public void stop() {

        // TODO Auto-generated method stub

    }

}
```

新建大众高级车：

```
package car_imple;
```

```

import car_interface.Icar_interface;

public class car_3w_imple_adv extends base_car_imple {

public void start() {

System.out.println("豪华版：" + this.getName() + " 车以专利技术起动了 最高速度为："
+ this.getSpeed());

}

public void stop() {

System.out.println("豪华版：" + this.getName() + " 车以专利技术停车了");

}

}

```

新建大众普通车：

```

package car_imple;

import car_interface.Icar_interface;

public class car_3w_imple_low extends base_car_imple {

public void start() {

System.out.println("普通版：" + this.getName() + " 车以专利技术起动了 最高速度为："
+ this.getSpeed());

}

public void stop() {

System.out.println("普通版：" + this.getName() + " 车以专利技术停车了");

}

}

```

新建大众普通车：

```

package car_imple;

import car_interface.Icar_interface;

```

```

public class car_audi_imple_adv extends base_car_imple {
    public void start() {
        System.out.println("富华版：" + this.getName() + " 车以专利技术起动了 最高速度为："
        + this.getSpeed());
    }
    public void stop() {
        System.out.println("富华版：" + this.getName() + " 车以专利技术停车了");
    }
}

```

新建奥迪普通车：

```

package car_imple;
import car_interface.Icar_interface;
public class car_audi_imple_low extends base_car_imple {
    public void start() {
        System.out.println("普通版：" + this.getName() + " 车以专利技术起动了 最高速度为："
        + this.getSpeed());
    }
    public void stop() {
        System.out.println("普通版：" + this.getName() + " 车以专利技术停车了");
    }
}

```

新建客户端运行类：

```

package run_main;
import car_factory_imple.car_factory_adv;
import car_factory_interface.Icar_factory;

```

```
import car_interface.Icar_interface;

public class run_main {

    public static void main(String[] args) {

        Icar_factory Icar_factory_ref = new car_factory_adv();

        Icar_interface Icar_interface_ref = Icar_factory_ref

            .create_threeW_car();

        Icar_interface_ref.start();

        Icar_interface_ref.stop();

    }

}
```

程序运行结果如下：

富华版：大众A6 车以专利技术起动了 最高速度为：300

富华版：大众A6 车以专利技术停车了

抓一篇阎宏的小文字：

一开始只在后花园中种蔬菜类的时候可以用简单工厂模式，由工厂负责生成具体的蔬菜类，

但是如果后花园要引进水果类的时候简单模式就行不通了，因此需要使用工厂方法模式，将产品类族分开。

但是如果后花园的规模继续扩大到地域范围的分割时，比如说一个在北京，一个在上海的时候，工厂方法模式就不够了，因为对两个后花园来说，每个后花园的植物是要被种在一起的，并且两个后花园用工厂方法模式是无法体现其区别的

从程序中可以看到，工厂是抽象的，工厂的实现是不样的，不同的工厂创建出

不同汽车。而工厂方法仅仅是用一个工厂去创建很多汽车。

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质
电子书下载！！！！

第十七章 兰州拉面馆——原型模式

17.1 兰州拉面馆

时间：1月2日 地点：兰州拉面馆 人物：大B，小A

这天大B出去办事，中午经过小A学校，正好约小A出来吃饭。

大B：“出来校门口，我请你吃饭。”

小A：“好。”

大B：“想吃什么？”

小A：“简单点好。对了，我们校门口那家兰州拉面馆很不错喔！你不是也喜欢吃面吗？我们去试一下？”

大B：“嘿嘿！不错喔！”

小A：“兰州拉面馆有牛肉拉面（大碗）、牛肉拉面（小碗）、牛肉刀削面、羊肉拉面、羊肉刀削面。”

大B：“嘿嘿！什么都好。我出来办事，肚子饿了喔！”

小A：“嘿嘿！如果说要吃手擀面、挂面、珍珠翡翠鲍鱼面，恐怕没有喔！”

大B：“别啦！随便就好。”

17.2 原型模式

牛肉拉面，牛肉刀削面等可以看作是原型，面馆可以看作是原型管理器。原型模式在创建对象不是直接创建的，也就是说，不是通过外部调用类的构造函数调用的，而是通过已存在的对象实例克隆出一个对象，这个克隆对象和他的源对象具有相同的属性和状态，也就是说面馆里的牛肉刀削面每一碗状态都是一样的。

大B：“你现在知不知道什么是原型模式？”

小A：“有点意识。但不是很清楚。”

大B：“原型模式是允许一个对象再创建另外一个可定制的对象，根本无需知道任何如何创建的细节，工作原理是：通过将一个原型对象传给那个要发动创建的对象，这个要发动创建的对象通过请求原型对象拷贝它们自己来实施创建。”

小A：“原型模式有什么意图？”

大B：“用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。”

参考图17-1原型模式结构图：

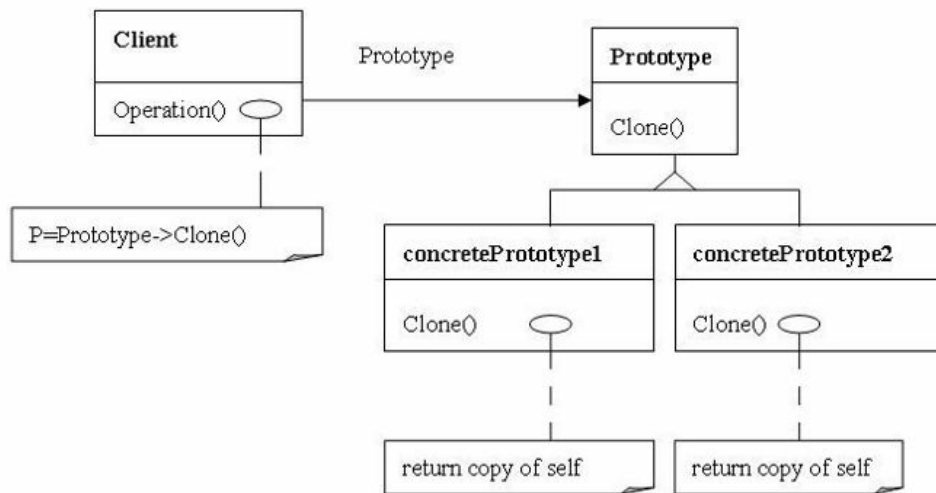


图17-1 原型模式结构图

17.3 原型模式应如何使用？

小A：“师兄，原型模式应该如何使用？”

大B：“因为Java中的提供`clone()`方法来实现对象的克隆，所以Prototype模式实现一下子变得很简单。”

以勺子为例：

```
public abstract class AbstractSpoon implements Cloneable
{
    String spoonName;
    public void setSpoonName(String spoonName) {this.spoonName = spoonName;}
    public String getSpoonName() {return this.spoonName;}
    public Object clone()
    {
```

```
Object object = null;
try {
    object = super.clone();
} catch (CloneNotSupportedException exception) {
    System.err.println("AbstractSpoon is not Cloneable");
}
return object;
}
}
```

有两个具体实现(ConcretePrototype)：

```
public class SoupSpoon extends AbstractSpoon
{
    public SoupSpoon()
    {
        setSpoonName("Soup Spoon");
    }
}

public class SaladSpoon extends AbstractSpoon
{
    public SaladSpoon()
    {
        setSpoonName("Salad Spoon");
    }
}
```

调用Prototype模式很简单：

```
AbstractSpoon spoon = new SoupSpoon();
```

```
AbstractSpoon spoon = new SaladSpoon();
```

当然也可以结合工厂模式来创建AbstractSpoon实例。

在Java中Prototype模式变成clone()方法的使用，由于Java的纯洁的面向对象特性，使得在Java中使用设计模式变得很自然，两者已经几乎是浑然一体了。

17.4 为什么需要原型模式？

小A：“为什么需要原型模式？”

大B：“引入原型模式的本质在于利用已有的一个原型对象，快速的生成和原型对象一样的实例。你有一个A的实例a:A a = new A();现在你想生成和car1一样的一个实例b，按照原型模式，应该是这样：A b = a.Clone();而不是重新再new一个A对象。通过上面这句话就可以得到一个和a一样的实例，确切的说，应该是它们的数据成员是一样的。Prototype模式同样是返回了一个A对象而没有使用new操作。”

17.5 原型模式的优点和缺点

小A：“原型模式有什么优点吗？”

大B：“原型模式的优点：1、Prototype模式允许动态增加或减少产品类。由于创建产品类实例的方法是产批类内部具有的，因此增加新产品对整个结构没有影

响。2、Prototype模式提供了简化的创建结构。工厂方法模式常常需要有一个与产品等级结构相同的等级结构，而Prototype模式就不需要这样。3、Portotype模式具有给一个应用软件动态加载新功能的能力。由于Prototype的独立性较高，可以很容易动态加载新功能而不影响老系统。4、产品类不需要非得有任何事先确定的等级结构，因为Prototype模式适用于任何的等级结构。”

小A：“原型模式又有些什么缺点呢？”

大B：“原型模式的缺点：每一个类必须配备一个克隆方法。而且这个克隆方法需要对类的功能进行通盘考虑，这对全新的类来说不是很难，但对已有的类进行改造时，不一定是件容易的事。”

17.6 原型模式满足了哪些面向对象的设计原则？

小A：“师兄，原型模式满足了哪些面向对象的设计原则？”

大B：“依赖倒置原则：上面的例子，原型管理器（ColorManager）仅仅依赖于抽象部分（ColorPrototype），而具体实现细节（ConcteteColorPrototype）则依赖与抽象部分（ColorPrototype），所以Prototype很好的满足了依赖倒置原则。”

17.7 原型模式实现要点

小A：“原型模式实现要点有哪些？”

大B：“1、使用原型管理器，体现在一个系统中原型数目不固定时，可以动态的创建和销毁，2、实现克隆操作，在.NET中可以使用Object类的MemberwiseClone()方法来实现对象的浅表拷贝或通过序列化的方式来实现深拷贝。3、Prototype模式同样用于隔离类对象的使用者和具体类型（易变类）之间的耦合关系，它同样要求这些‘易变类’拥有稳定的接口。”

17.8 原型模式适用性

小A：“什么情况下，应当使用原型模式？”

大B：“1、当一个系统应该独立于它的产品创建，构成和表示时；2、当要实例化的类是在运行时刻指定时，例如，通过动态装载；3、为了避免创建一个与产品类层次平行的工厂类层次时；4、当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。”

原型模式同工厂模式，同样对客户隐藏了对象的创建工作，但是，与通过对一个类进行实例化来构造新对象不同的是，原型模式是通过拷贝一个现有对象生成新对象的，达到了‘隔离类对象的使用者和具体类型（易变类）之间的耦合关系’的目的。

17.9 克隆对象分为浅拷贝和深拷贝

小A：“克隆对象有哪些？”

大B：“克隆对象分为浅拷贝和深拷贝。我给你详细讲一下。浅拷贝就是克隆的对象和它的源对象共享引用的对象，举个例子，可能不恰当，假设牛肉刀削面引用一个对象：Money，表示它值多少钱，这里说的对象是说它是System.Object继承的(c#)，也就是说不同的浅拷贝对象，他们的价钱是一样的，当克隆对象的价钱变过之后，它所引用的对象的价钱也就随之改变了，比如面馆。调低了牛肉刀削面的价格，由5块钱调到了4块钱，那么每碗面的价格都变到了四块钱。但对值类型而言，每个浅拷贝对象都有自己的拷贝，也就是说，当改变克隆对象的值类型时，它的源对象相应属性不会改变。深拷贝就是完全的拷贝。不仅值类型有自己的拷贝，连引用对象也有自己的一份拷贝，修改克隆对象的任何属性，也不会对源对象产生任何影响。原型管理器，就是提供原型注册使用，当创建对象使，可以使用里面的对象进行克隆，当有新的实例对象时，也可以将他们加到原型管理器里。说的有点乱，程序员，还是用代码交流最好。”

下面的程序分别实现了原型克隆时浅拷贝和深拷贝。

```
//  
//理解深拷贝和浅拷贝  
//  
//浅表副本创建与原始对象具有相同类型的新实例，然后复制原始对象的非静态字段。  
//如果字段是值类型的，则对该字段执行逐位复制。如果字段是引用类型，则复制该  
//引用但不复制被引用的对象；这样，原始对象中的引用和副本中的引用指向同一个对象。  
//相反，对象的深层副本复制对象中字段直接或间接引用的全部内容。
```


//
//例如，如果 x 是一个具有对对象 A 和对象 B 的引用的 Object，并且对象 A 还具
//有对对象 M 的引用，则 x 的浅表副本是对象 Y，而 Y 同样具有对对象 A 和对象 B
//的引用。相反，x 的深层副本是对象 Y，而对象 Y 具有对对象 C 和对象 D 的直接引
//用以及对对象 N 的间接引用，其中 C 是 A 的副本，D 是 B 的副本，而 N 是 M 的副本。

```
using System;
using System.Collections;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using System.Data;
namespace Prototype
{
    ///
    /// Prototype类型实例
    ///
    class TestPrototypeApp
    {
        ///
        /// 应用程序的主入口点。
        ///
        [STAThread]
        static void Main(string[] args)
        {
            //定义原型管理器
            NoodleManager noodleManager=new NoodleManager();
            //客户要求下面三碗面
            Noodle beefNoodle=(Noodle)noodleManager["牛肉拉面"].Clone();
            // Noodle beefNoodle=(Noodle)noodleManager["牛肉拉面"].DeepClone();
            Noodle muttonNoodle=(Noodle)noodleManager["羊肉拉面"].Clone();
            Noodle beefCutNoodle=(Noodle)noodleManager["牛肉刀削面"].Clone();
            //修改克隆对象中的引用对象的属性，验证它是浅拷贝还是深拷贝
```

```

beefNoodle.TbName="，哈哈！，克隆对象改名了，你改不改";
//显示原始对象的NoodleName和TbName
Console.WriteLine(noodleManager["牛肉拉面"].NoodleName+noodleManager["牛肉拉
面"].TbName+"\n");
//显示克隆对象的NoodleName和TbName
Console.WriteLine(beefNoodle.NoodleName+beefNoodle.TbName+"\n");
//将新的产品加入原型管理器，以备以后克隆时使用，下面是定义了一种新的面条 - 羊肉刀削面，
//并把它添加到面条管理器中，如果以后再有客户点这个面，直接克隆即可。
noodleManager["羊肉刀削面"]=new CutNoodle("羊肉刀削面");
//克隆一碗羊肉刀削面
Noodle muttonCutNoodle=(Noodle)noodleManager["羊肉刀削面"].Clone();
Console.WriteLine(noodleManager["羊肉刀削面"].NoodleName+"\n");
Console.WriteLine(muttonCutNoodle.NoodleName+"\n");
Console.ReadLine();
}
}
//抽象产品 - 面条
//序列化属性，为深拷贝时使用，每个派生类都要加上此属性才能实现深拷贝
[Serializable]
public abstract class Noodle
{
//定义一个DataTable对象，主要是为了验证对比类中含有引用对象时的深拷贝和浅拷贝时的不同，
//你也可以采用别的任何引用对象
protected DataTable dataTable=new DataTable();
public string TbName
{
get{return dataTable.TableName;}
set{dataTable.TableName=value;}
}
//字段
protected string noodleName;
//特性

```

```

public string NoodleName
{
    get{return noodleName;}
    set{noodleName=value;}
}

public abstract Noodle Make(string name);
//浅克隆的接口

public abstract Noodle Clone();
//深克隆的接口

public abstract Noodle DeepClone();
}

//具体产品，拉面
[Serializable]
public class PullNoodle:Noodle
{
    public PullNoodle(string name)
    {
        this.NoodleName=name;
        this.TbName=name+"table";
        Console.WriteLine("PullNoodle is made\n");
    }
    //实现浅拷贝
    public override Noodle Clone()
    {
        return (Noodle)this.MemberwiseClone();
    }
    //实现深拷贝，“淹咸菜”的过程，先将对象序列化到内存流，再反序列化，即可得到深克隆
    public override Noodle DeepClone()
    {
        //定义内存流
        MemoryStream ms=new MemoryStream();
        //定义二进制流

```

```

IFormatter bf=new BinaryFormatter();
//序列化
bf.Serialize(ms,this);
//重置指针到起始位置,以备反序列化
ms.Position=0;
//返回反序列化的深克隆对象
return (Noodle)bf.Deserialize(ms);
}

public override Noodle Make(string name)
{
return new PullNoodle(name);
}
}

//具体产品 - 刀削面
[Serializable]
public class CutNoodle:Noodle
{
public CutNoodle(string name)
{
this.NoodleName=name;
this.TbName=name+"table";
Console.WriteLine("CutNoodle is made\n");
}
}

//实现浅克隆
public override Noodle Clone()
{
return (Noodle)this.MemberwiseClone();
}

public override Noodle Make(string name)
{
return new CutNoodle(name);
}
}

```

//实现深克隆

```
public override Noodle DeepClone()
{
    MemoryStream ms=new MemoryStream();
    IFormatter bf=new BinaryFormatter();
    bf.Serialize(ms,this);
    ms.Position=0;
    return (Noodle)bf.Deserialize(ms);
}
}
```

//定义原型管理器，用于存储原型集合，这里采用的是HashTable

```
class NoodleManager
{
    //定义HashTable
    protected Hashtable noodleHt=new Hashtable();
    protected Noodle noodle;
    public NoodleManager()
    {
        //初始化时加入三种基本原型
        noodle=new PullNoodle("牛肉拉面");
        noodleHt.Add("牛肉拉面",noodle);
        noodle=new PullNoodle("羊肉拉面");
        noodleHt.Add("羊肉拉面",noodle);
        noodle=new CutNoodle("牛肉刀削面");
        noodleHt.Add("牛肉刀削面",noodle);
    }
    //索引器，用于添加，访问Noodle对象
    public Noodle this[string key]
    {
        get{ return (Noodle)noodleHt[key];}
        set{ noodleHt.Add(key,value);}
    }
}
```

}

}

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质
电子书下载！！！！

第十八章 月光宝盒——备忘录模式

18.1 月光宝盒

时间：1月3日 地点：大B房间 人物：大B，小A

这天，大B和小A聊起了《大话西游》。

小A：“师兄，你看过《大话西游》吗？”

大B：“看过好几回哩，挺好看的。”

小A：“是啊！我也是这么觉得。”

小A：“嘿嘿！如果时间可以倒流就好了。”

大B：“你是不是看《大话西游》看晕了？”

小A：“是啊！如果我也个月光宝盒那好了。”

大B：“是不是你也想来个时间倒流，然后也来个跨越时空的爱情？”

小A：“呵呵。如果可能那当然好。”

18.2 备忘录模式

俗话说：世上难买后悔药。所以凡事讲究个“三思而后行”，但总常见有人做“痛心疾首”状：当初我要是……。如果真的有《大话西游》中能时光倒流的“月光宝盒”，那这世上也许会少一些伤感与后悔——当然这只能是痴人说梦了。

但是在我们手指下的程序世界里，却有的后悔药买。我们讲的备忘录模式便是程序世界里的“月光宝盒”。

小A：“什么叫备忘录模式？”

大B：“备忘录（Memento）模式又称标记（Token）模式。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原先保存的状态。在讲命令模式的时候，我们曾经提到利用中间的命令角色可以实现undo、redo的功能。从定义可以看出备忘录模式是专门来存放对象历史状态的，这对于很好的实现undo、redo功能有很大的帮助。所以在命令模式中undo、redo功能可以配合备忘录模式来实现。其实单就实现保存一个对象在某一时刻的状态的功能，还是很简单的——将对象中要保存的属性放到一个专门管理备份的对象中，需要的时候则调用约定好的方法将备份的属性放回到原来的对象中去。但是你要好好看看为了能让你的备份对象访问到原对象中的属性，是否意味着你就要全部公开或者包内公开对象原本私有的属性呢？如果你的做法已经破坏了封装，那么就要考虑重构一下了。”

18.3 使用备忘录模式的原因

小A：“为什么要使用备忘录模式？”

大B：“想要恢复对象某时的原有状态。”

18.4 备忘录模式适用的情况举例

小A：“能不能举些例子说一下备忘录模式适用的情况？”

大B：“有很多备忘录模式的应用，只是我们已经见过，却没细想这是备忘录模式的使用罢了，略略举几例：1、备忘录在jsp+javabean的使用：在一系统中新增帐户时，在表单中需要填写用户名、密码、联系电话、地址等信息，如果有些字段没有填写或填写错误，当用户点击‘提交’按钮时，需要在新增页面上保存用户输入的选项，并提示出错的选项。这就是利用JavaBean的scope= ‘request’ 或 scope= ‘session’ 特性实现的，即是用备忘录模式实现的。2、修理汽车的刹车时。首先移开两边的挡板，露出左右刹车片。只能卸下一片，这时另一片作为一个备忘录来表明刹车是怎样安装的。在这片修理完成后，才可以卸下另一片。当第二片卸下时，第一片就成了备忘录。3、都说人生没有后悔药可买，我们都在为所做的事付出着代价，但在软世界里却有‘后悔药’，我改变了某东西的某些状态之后，只要我们之前保存了该东西的某状态，我们就可以通过备忘录模式实现该东西的状态还原，其实这何尝不是一个能使时光倒流的‘月光宝盒’，总‘神奇’一词了得。”

18.5 “月光宝盒” 备忘录模式的组成部分

小A：“那‘月光宝盒’备忘录模式有哪些组成部分？”

大B：“1、备忘录（Memento）角色：备忘录角色存储‘备忘发起角色’的内部状态。‘备忘发起角色’根据需要决定备忘录角色存储‘备忘发起角色’的哪些内部状态。为了防止‘备忘发起角色’以外的其他对象访问备忘录。备忘录实际上有两个接口，‘备忘录管理者角色’只能看到备忘录提供的窄接口——对于备忘录角色中存放的属性是不可见的。‘备忘发起角色’则能够看到一个宽接口——能够得到自己放入备忘录角色中属性。2、备忘发起（Originator）角色：‘备忘发起角色’创建一个备忘录，用以记录当前时刻它的内部状态。在需要时使用备忘录恢复内部状态。3、备忘录管理者（Caretaker）角色：负责保存好备忘录。不能对备忘录的内容进行操作或检查。我们称它为宽接口；而另一个则可以只是一个标示，我们称它为窄接口。备忘录角色要实现这两个接口类。这样对于‘备忘发起角色’采用宽接口进行访问，而对于其他的角色或者对象则采用窄接口进行访问。这种实现比较简单，但是需要人为的进行规范约束——而这往往是没有力度的。第二种方法便很好的解决了第一种缺陷：采用内部类来控制访问权限。将备忘录角色作为‘备忘发起角色’的一个私有内部类。好处我不详细解释了，看看代码吧就明白了。下面的代码是一个完整的备忘录模式的教学程序。它便采用了第二种方法来实现备忘录模式。还有一点值得指出的是，在下面的代码中，对于客户程序来说‘备忘录管理者角色’是不可见的，这样简化了客户程序使用备忘录模式的难度。下面采用‘备忘发起角色’来调用访问‘备忘录管理者角色’，也可以参考门面模式在客户程序与备忘录角色之间添加一个门面角色。”

```
class Originator{  
    //这个是要保存的状态  
    private int state= 90;
```

//保持一个“备忘录管理者角色”的对象

```
private Caretaker c = new Caretaker();
```

//读取备忘录角色以恢复以前的状态

```
public void setMemento(){
```

```
Memento memento = (Memento)c.getMemento();
```

```
state = memento.getState();
```

```
System.out.println("the state is "+state+" now");
```

```
}
```

//创建一个备忘录角色，并将当前状态属性存入，托给“备忘录管理者角色”存放。

```
public void createMemento(){
```

```
c.saveMemento(new Memento(state));
```

```
}
```

//this is other business methods...

//they maybe modify the attribute state

```
public void modifyState4Test(int m){
```

```
state = m;
```

```
System.out.println("the state is "+state+" now");
```

```
}
```

//作为私有内部类的备忘录角色，它实现了窄接口，可以看到在第二种方法中宽接口已经不再需要

//注意：里面的属性和方法都是私有的

```
private class Memento implements MementoIF{
```

```
private int state ;
```

```
private Memento(int state){
```

```
this.state = state ;
```

```
}
```

```
private int getState(){
```

```
return state;
```

```
}
```

```
}
```

```
}
```

//测试代码——客户程序

```
public class TestInnerClass{
```

```

public static void main(String[] args){
    Originator o = new Originator();
    o.createMemento();
    o.modifyState4Test(80);
    o.setMemento();
}
}
//窄接口
interface MementoIF{}
// "备忘录管理者角色"
class Caretaker{
    private MementoIF m ;
    public void saveMemento(MementoIF m){
        t his.m = m;
    }
    public MementoIF getMemento(){
        return m;
    }
}
}

```

大B：“第三种方式是不太推荐使用的：使用clone方法来简化备忘录模式。由于Java提供了clone机制，这使得复制一个对象变得轻松起来。使用了clone机制的备忘录模式，备忘录角色基本可以省略了，而且可以很好的保持对象的封装。但是在为你的类实现clone方法时要慎重啊。 在上面的代码中，我们简单的模拟了备忘录模式的整个流程。在实际应用中，我们往往需要保存大量‘备忘发起角色’的历史状态。这时就要对我们的‘备忘录管理者角色’进行改造，最简单的方式就是采用容器来按照顺序存放备忘录角色。这样就可以很好的实现undo、redo功能了。”

18.6 备忘录模式适用情况

大B：“使用了备忘录模式来实现保存对象的历史状态可以有效地保持封装边界。使用备忘录可以避免暴露一些只应由‘备忘发起角色’管理却又必须存储在‘备忘发起角色’之外的信息。把‘备忘发起角色’内部信息对其他对象屏蔽起来，从而保持了封装边界。但是如果备份的‘备忘发起角色’存在大量的信息或者创建、恢复操作非常频繁，则可能造成很大的开销。”

小A：“那使用备忘录模式的前提是什么？”

大B：“1、必须保存一个对象在某一个时刻的(部分)状态，这样以后需要时它才能恢复到先前的状态。

2、如果一个用接口来让其它对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。”

18.7 备忘录模式的实现

小A：“备忘录模式要怎么去实现？”

大B：“我详细给你说。”

1、备忘录模式中的角色

发起人：创建含有内部状态的备忘录对象，并使用备忘录对象存储状态

负责人：负责人保存备忘录对象，但不检查备忘录对象的内容

备忘录：备忘录对象将发起人对象的内部状态存起来，并保证其内容不被发起人对象之外的对象像读取

注意：在备忘录的角色中，定义了他必须对不同的人提供不同的接口，对发起人提供宽接口，对其它任何人提供窄

接口。也许你说我都提供宽接口得了。对这也就是备忘录的一种实现，叫做白箱备忘录，不过这种方法的封装没有设计

好，安全性不够好。

2、白箱备忘录的实现：

```
public class Originator{
    private String state;
    public Memento CreateMemento(){
        return new Memento(state);
    }
    public void restoreMemento(Memento memento){
        this.state = memento.getState();
    }
    public String getState(){
        return this.state;
    }
    public void setState(String state){
        this.state=state;
        System.out.println("Current state = " + this.state);
    }
}
```

```

}
public class Memento{
private String state;
public Memento(String state){
this.state = state;
}
public String getState(){
return this.state;
}
public void setState(){
this.state = state;
}
}

public class Caretaker{
private Memento memento;
public Memento retrieveMemento(){
return this.memento;
}
public void saveMemento(Memento memento){
this.memento = memento;
}
}

public class Client {
private static Originator o = new Originator();
private static Caretaker c = new Caretaker();
public static void main(Sting[] args){
o.setState("ON");
c.saveMemento(o.createMemento());
o.setState("OFF");
o.restoreMemento(c.retrieveMemento());
}
}

```

白箱的优点：实现简单

白箱的缺点：上边说了，破坏了封装，安全性有些问题。

说明：这里白箱的实现只保存了一个状态，其实是可以保存多个状态的。

3、双接口的实现，宽窄接口（黑箱）

如何实现宽窄接口呢，内部类也许是个好方法。我们把备忘录类设计"成发起人"的内部类，但这样还有的问题是同一package中的其它类也能访问到，为了解决这个问题，我们可以把"备忘录"的方法设计成私有的方法，这样就可以保证封装，又保证发起人能访问到。实现如下：

定义窄接口。

```
public interface NarrowMemento{
    public void narrowMethod();
}

class Originator {
    private String state;
    private NarrowMemento memento;
    public Originator(){
    }

    public NarrowMemento createMemento(){
        memento = new Memento(this.state);
        return memento;
    }

    public void restoreMemento(NarrowMemento memento){
        Memento aMemento = (Memento)memento;
        this.setState(aMemento.getState());
    }
}
```



```

}
public String getState(){
return this.state;
}
public void setState(String state){
this.state = state;
}
//内部类
protected class Memento implements NarrowMemento{
private String savedState;
private Memento(String someState){
saveState = someState;
}
private void setState(String someState){
saveState = someState;
}
private String getState()
{
return saveState;
}
public void narrowMethod(){
System.out.println("this is narrow method");
}
}
public NarrowMemento getNarrowMemento(){
return memento;
}
}
public class Caretaker{
private NarrowMemento memento;
public NarrowMemento retrieveMemento(){
return this.memento;
}
}

```

```

}

public void saveMemento(NarrowMemento memento){
    this.memento = memento;
}

}

public class Client{
    private static Originator o = new Originator();
    private static Caretaker c = new Caretaker();
    public static void main(String[] args){
        //use wide interface
        o.setState("On");
        c.saveMemento(o.createMemento());
        o.setState("Off");
        o.restoreMemento(c.retrieveMemento());
        //use narrow interface
        NarrowMemento memento = o.getNarrowMemento();
        memento.narrowMethod();
    }
}

```

大B：“还要注意的是：1、前边两个例子都是记录了单个状态(单check点)，要实现多个状态点很容易，只须要把记录state的字符串换成一个list，然後添加，取得。如果须要随机须得状态点，也可以用map来存放。这样多个check点就实现了。2、一般情况下可以扩展负责人的功能，让负责人的功能更强大，从而让客户端的操做更少些。解放客户端。3、自述历史模式，这个就是把发起人，负责人写在一个类中，平时的应用中这种方法比较常见。”

18.8 备忘录角色的作用

小A：“备忘录角色有什么作用啊？”

大B：“1、将发起人对象的内部状态存储起来，备忘录能根据发起人对象的判断来决定存储多少发起人对象的内部状态。2、备忘录能保护其内容不被发起人对象之外的所有对象所读取。”

小A：“发起人角色有什么作用？”

大B：“1、创建一个含有当前内部状态的备忘录对象。2、使用备忘录对象存储其内部状态。”

小A：“那负责人角色又有什么作用？”

大B：“负责保存备忘录对象和不检查备忘录对象的内容。”

18.9 备忘录模式与命令模式的区别

小A：“备忘录模式与命令模式有一些相似之处，他们都保存状态，他们都可以拥有前进与后退，但是他们到底在设计上与实现上有哪些差别呢？”

大B：“1、异同点：从UML我们可以清晰的看到区别，一个保存Object的状态，一个保存命令。相同：都可以前进后退。不同：执行对象不同，保存状态的对象不通，所执行的操作也不相同。由于两种模式时所对应的需求截然不同，应该说备忘录更加稳定一些，而命令的执行则更加广泛，可能一个子类的Command对应一个Receiver。所以相对而言Command模式会更加灵活一些。应用：Command模式：

将命令当作一个对象进行保存，进行Redo，Undo操作。”

例子：在绘图系统中经常需要进行Redo，Undo操作。Memento模式：获取和保存对象的内部状态。例子：网上购物时购物车既可以理解为Memento。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第五部分

操作型模式

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第十九章 儿子的功课——操作型模式

19.1 儿子的功课

时间：1月4日 地点：大B的朋友小李家 人物：大B的朋友小李、小李的老婆LL、小李的儿子强强

强强：“老妈，这个题不会做了帮帮我。”

LL：“要先这样，然后在这样就行了。”

强强：“老妈，这个又不会了。”

LL：“这个我也不会，问你老爸去吧！”

小李：“这么简单呀！你们俩去看电视吧，我做完了叫你。”

19.2 操作型模式

大B：“我们在编写一个Java方法时，完成的是整个编码工作的一个基本单位，层次要高于一条语句。设计的方法必须参与整个应用设计、架构和测试计划中；不

过在面向对象编程中，编写方法是中心。”

小A：“这样说来方法真的很重要喔。”

大B：“尽管方法如此重要，但是对于方法是什么，方法是如何运行的，却很容易使人感到困惑。我们必须知道Java方法的语法细节。最令人不可思议的是，很多开发人员和图书的作者经常混淆方法和操作这两个概念。算法和多态性这两个概念更加抽象，但最终也是由方法实现的。”

小A：“是啊！”

大B：“我们只有弄清楚诸如算法、多态性、方法和操作等术语的不同意义，才能够准确地理解众多设计模式中所涉及的重要概念。特别是State（状态）模式、Strategy（策略）模式，以及Intepreter模式，都需要跨越几个类的方法来实现一个操作。只有当我们对方法和操作的含义达成了共识，这样的表达才具有意义。”

19.3 要点

小A：“操作型模式有什么要点呢？”

大B：“State模式将所有与一个特定状态相关的行为都放入一个State的子类对象中，在对象状态切换时，切换响应的对象；但同时维持State的接口，这样实现了具体操作与状态转换之间的解藕。为不同的状态引入不同的对象使得状态转换变得更加明确，而且可以保证不会出现状态不一致的情况，因为转换是原子性的 - -

即要么彻底转换过来，要么不转换。如果State对象没有实例变量，那么各个上下文可以共享同一个State对象，从而节省对象开销。这种模式避免了我们写大量的if else或switch case语句，但是很有可能会导致某些系统有过多的具体状态类，并且由此导致开发人员可能会对所有的状态类有所遗漏。”

19.4 操作和方法

小A：“在有关类的众多术语中，尤其需要注意区分操作与方法这两个概念。”

大B：“是的。UML是这样定义操作和方法的：1、操作就是能够被类的实例调用的服务的规范2、方法则是操作的实现。”

大B：“注意，操作是在方法之上的抽象概念操作定义类所提供的服务，并给出调用该服务的接口。多个类可以用不同的方法实现同一个操作。例如，很多类都以自己的方式实现toString()操作。而每个类都是通过实现方法来提供某种操作，操作的实现代码构成了类的方法。通过对方法和操作进行定义，我们可以澄清很多设计模式的结构。操作的含义是从方法的概念上抽象而来的。由于设计模式也是从类和方法升华而来，因而，在很多设计模式中，操作都发挥着非常重要的作用。例如，在Composite模式中，一个操作既被应用于叶节点对象上，又被应用于组合对象上。而在Proxy模式中，一个中介者对象与目标对象具有相同的操作，从而使用得中介者对象可以管理对目标对象的访问。同样，责任链(Chain of Responsibility)模式是在一个对象链上分发某个操作。每个对象的方法要么直接实现该方法的业务，要么将对该方法的调用转发到责任链上的下一个对象。”

19.5 签名

小A：“表面上，操作的含义与签名的含义很类似。”

大B：“是啊！两个词指的都是方法的接口。”

小A：“喔。”

大B：“当编写方法时，遵循签名就可以之：方法签名包括方法名、形式参数的数目和类型。注意，方法签名不包括返回类型。但是，如果一个方法重写了另一个方法，而这两个方法的返回类型不同，那么编译时就会报错。在客户端调用方法时，方法签名指明应该调用哪个方法。操作是可请求的服务的规范。术语签名与操作的含义很类似，但是名字本身并不同义。这两个术语的不同之处主要体现在所使用的上下文环境中。当研究不同类中的方法可能会有相同接口时，使用术语操作。当研究Java如何将一个方法调用映射到接收对象的方法时（具体方法），使用术语签名。签名依赖于方法名和参数，但不依赖于返回类型。”

19.6 异常

小A：“运行正常时，方法会返回正确的结果；有时候，方法也会抛出异常，或者调用其他抛出异常的方法。当方法正常返回结果时，程序会从上次的调用点之后继续执行。”

大B：“当发生异常时，则需要应用不同的规则集。当抛出异常时，Java执行环

境必须找到与该异常相符的try/catch语句。如果在当前的方法调用的栈中没有发现相符的try/catch语句，程序会停止，甚至崩溃。”

19.7 算法和多态性

大B：“算法和多态性是编程中的主要思想，但是靠这些术语，我们很难表述其具体含义。如果想向其他人表示一个方法，你可以编辑源代码，然后据此向其他人仔细讲述。”

小A：“喔。”

大B：“在某些情况下，算法也许完全包含在一个方法中。但是算法的实现经常依赖于多个方法的相互使用。”

小A：“嗯。”

大B：“Introduction to Algorithms(算法导论)给出了算法的定义：算法是定义良好的计算过程，把数据值或者数据集合作为输入，并输出某数据值或者数据集。算法是一个过程——包含一些指令序列，接收输入，产生输出。”

小A：“嗯。”

大B：“单个方法也许是个算法：它接受输入——其参数列表——并产生输出作为返回值。然而，在面向对象编程时很多算法会需要多个方法。算法就是需要完成某项任务的过程。它们也许表现为某方法的一部分，或者调用多个方法。在面向对象应用程序中，需要多个方法的算法经常依赖多态性来充许单个操作

的多种实现。”

小A：“多态性是方法调用关于依赖被调用的操作和调用接收者类的基本原则。”

大B：“是啊！比如，你也许关心当Java遇到表达式isTree()时，会执行哪个方法。这关键是看方法的依赖关系。”

小A：“喔。”

大B：“如果对象m是Machine类的实例Java会调用Machine.isTree()。如果m是MachineComposite的一个实例，Java会调用MachineComposite.isTree()。非正式地说，多态性意味着要为合适的对象调用合适的方法。很多设计模式都使用多态性，在某些情况下，多态性与该模式的目标紧紧相连。”

小A：“操作、方法、签名以及算法这四个概念很容易让人混淆。”

大B：“但是，搞清楚这些术语间区别将有助于我们描述一些重要的概念。”

小A：“是啊！”

大B：“操作，类似于方法签名，定义了服务的规范。当谈到许多方法可能会有相同的接口的时候，我们可以使用操作这个术语。当讨论方法查询规划的时候，我们可以使用签名这个术语。一个方法的定义包括方法签名、修饰符、返回类型以及方法体；而方法签名又包括方法名和参数列表。一个方法通常有一个方法签名，并实现一个操作。启动一个方法的常见方式是调用它。方法结束的常见方式是让它返回，但是对任务程序而言，当遇到不可处理的异常时，任何方法都会停止执行。算法是一个接收输入并产生输出的过程。方法也接收输入、产生输出，另外它还包含

一个过程化的方法体，因此常常有人将方法体看作是一个算法。然而，一个算法的过程可能会涉及很多操作和方法，也可能仅仅是另外一个方法的一部分。算法这个术语最好是在谈到产生某个结果的过程时使用。很多设计模式都涉及到把一个操作分散到几个类中去。因而我们可以说这些模式依赖于多态性，即具体调用哪个方法依赖于收到调用的对象的类型。不同的类可以用不同的方式来实现同一个操作。换句话说，Java支持多态性机制。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第二十章 订单处理——模板方法模式

20.1 订单处理

时间：1月5日 地点：大B房间 人物：大B，小A

这天，大B和小A在讨论怎样去处理订单的问题。

小A：“一个客户可以在个订货单中订购多个货物(也称为订货单项目)，货物的销售价是根据货物的进货价进行计算的。”

大B：“有些货物可以打折的，有些是不可以打折的。每一个客户都有一个信用额度，每张订单的总价不能超出该客户的信用额度。”

小A：“那我们应该怎样去处理这个订单？”

大B：“处理一个订单需要的步聚：1、遍历订货单的订货单项目列表，累加所有货物的总价格(根据订货单项目计算出销售价) 2、根据客户号获得客户的信用额度 3、把客户号，订单的总价格，及订单项目列表写入到数据库。”

小A：“但是我们并不能确定怎么计算出货物的销售价，怎样根据客户号获得客户的信用额度及把订单信息写入数据库这些方法的具体实现？”

大B：“所以用一个抽象类AbstractOrder确定订单处理的逻辑，把不能确定的方法定义为抽象方法，由子类去完成具体的实现。”

20.2 模板方法模式

小A：“通常我们会遇到这样的一个问题：我们知道一个算法所需的关键步骤，并确定了这些步骤的执行顺序。但是某些步骤的具体实现是未知的，或者是某些步骤的实现与具体的环境相关。”

大B：“模板方法模式把我们不知道具体实现的步骤封装成抽象方法，提供一些按正确顺序调用它们的具体方法(这些具体方法统称为模板方法)，这样构成一个抽象基类。子类通过继承这个抽象基类去实现各个步骤的抽象方法，而工作流程却由父类来控制。”

小A：“什么是模板方法模式？”

大B：“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。”

小A：“要如何去实现它哩？”

大B：“模板模式，是众多模式之中用得比较多的模式，在具体的应用中，我们已经经意或者不经意的采用了这种模式。其是写定义，后实现，然后再调用，将实现与调用分开，从而利用增强了程序的延展性。模板模式是利用了虚函数的多态

性，我们可以实现，也可以不实现。”

参考图20-1模板方法模式结构图

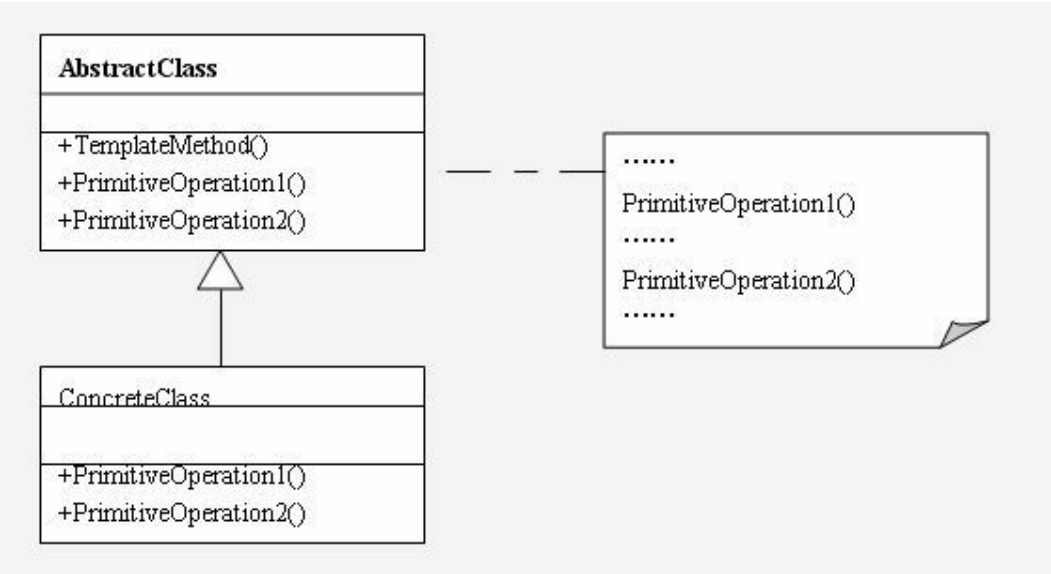


图20-1 模板方法模式结构图

代码

```
public abstract class AbstractOrder {
public Order placeOrder(int customerId , List orderItemList){
int total = 0;
for(int i = 0; i  getSpendingLimit(customerId)){
throw new BusinessException( "超出信用额度" + getSpendingLimit(customerId));
}
int orderId = saveOrder(customerId, total, orderItemList);
return new OrderImpl(orderId,total);
}
public abstract int getOrderItemPrice(OrderItem orderItem);
public abstract int getSpendingLimit(int customerId);
public abstract int saveOrder(int customerId, int total, List orderItemList);
}
render_code();
```

AbstractOrder在placeOrder方法中确定了定单处理的逻辑，placeOrder方法也称为模板方法。在placeOrder中调用了三个抽象方法。子类只需要去实现三个抽象方法，而无需要去关心定单处理的逻辑。

20.3 模板方法模式结构

小A：“师兄，能给我讲讲模板方法模式的结构吗？”

大B：“在模板方法模式中两个参与者进行协作。”

小A：“哪两种参与者？”

大B：“抽象模板类和具体类。”

小A：“什么是抽象模板类？”

大B：“定义一个或多个抽象操作，由子类去实现。这些操作称为基本操作。定义并实现一个具体操作，这个具体操作通过调用基本操作给确定顶级逻辑。这个具体操作称为模板方法。”

小A：“什么是具体类啊？”

大B：“实现抽象模板类所定义的抽象操作。 如上面的订单处理所示：AbstractOrder就是抽象模板类，placeOrder即是抽象模板方法。

GetOrderItemPrice，getSpendingLimit和saveOrder三个抽象方法为基本操作。具体子类能过需要去实现这三个抽象方法。不同的子类可能有着不同的实现方

式。”

代码

```
Public class ConcreteOrder extends AbstractOrder{  
    public int getOrderItemPrice(OrderItem orderItem){  
        //计算货物的售价  
        .....  
    }  
    public int getSpendingLimit(int customerId){  
        //读取客户的信用额度  
        .....  
    }  
    public int saveOrder(int customerId, int total, List orderItemList){  
        //写入数据库  
        .....  
    }  
}
```

大B：“render_code()；ConcreteOrder为AbstractOrder的具体子类，ConcreteOrder需要去完成具体的三个基本操作。同时他也具有了父类一样的处理逻辑。把具体的实现延迟到了子类去实现，这就是模板方法模式所带来的好处。”

20.4 模板方法模式的目的

小A：“模板方法模式有什么目的？”

大B：“在一个方法中实现一个算法，并将算法中某些步骤的定义推迟，从而使得其他类可以重新定义这些步骤。在编写一个方法的时候，考虑到算法的某些步骤可能会有不同的实现方式，我们可能会首先定出算法的框架。这样，在定义方法的时候，我们可以将某些步骤定义为抽象方法，或者是将它们定义为存根方法，也可以将它们定义为某个单独接口中的方法，这就是模板方法模式的实现。模板方法的典型例子就是排序。java.util.Collections类将sort()方法定义为一个模板方法，而将其中的比较交给用户来实现，sort()方法所接受的List中的对象需要实现Comparator接口，这就是将实际算法推迟并交给其它类来实现的模板方法模式。”

例：

我们有一个抽象类ShowSubClassName，有两个子类SubClassA和SubClassB继承了该抽象类，抽象类中定义的抽象方法showName()被推迟到了两个子类中实现（分别输出自己的class name）。

```
public abstract class ShowSubClassName {  
    /**  
    * 这个方法的实现，我们交给子类完成  
    */  
    public abstract void showName();  
    public void show() {  
        showName();  
    }  
}  
  
public class SubClassA extends ShowSubClassName {  
    public void showName() {  
        System.out.println(this.getClass().getName());  
    }  
}
```

```

}

public class SubClassB extends ShowSubClassName {
    public void showName() {
        System.out.println(this.getClass().getName());
    }
}

public class Test {
    public static void main(String[] args) {
        ShowSubClassName classA = new SubClassA();
        classA.show();
        ShowSubClassName classB = new SubClassB();
        classB.show();
    }
}

```

20.5 模板方法模式的缺陷

小A：“模板方法模式有什么缺陷？”

大B：“组合优先于继承，而模板方法模式是少数几个必须从非纯虚类（C++的称呼，就是java的interface）继承来实现的模式之一。新手往往颇费周折才能理解其要义，因为它的复杂性-概念复杂性和实现复杂性。而用ioc模式代替继承才是降低复杂性的更好方案。1、概念复杂性。从面向对象方法的本质来讲，父类负责抽象，子类负责具象。而模板方法恰好反过来了，父类实现了大部分功能，而子类实现少数纯虚方法，然后由父类的方法调用。违反了面向对象的一般性思维的原因是这个模式的初衷并不是抽象，而是最大限度的重用。2、实现复杂性。这种重用方式

的代价就是每个子类身上都背负了父类强加给它的包袱。”

举个例子：

```
public abstract class ApplicationContext{
    protected String path ;
    public abstract InputStream getStream();
    public void build(){
        getStream();
    }
    public ApplicationContext(String path){
        this.path = path;
    }
}

public class ClassPathContext extends ApplicationContext{
    public ClassPathContext(String path){
        //super(path) 执行之前绝对不能使用path变量，因为父类还没有初始化。
        super(path); //为了能够执行父类构建器，强加给子类的代码。
    }
    public InputStream getStream(){... ..}
}
```

调用代码：

```
ApplicationContext context = new ClassPathContext("path");
context.build();
```

大B：“这里父类带给子类的负担有两点：1、父类的非默认构建器子类必须重写。2、子类在构建器里使用父类的成员变量时要注意顺序。由此可见，这个模式在

提高重用性的同时也增加了复杂性： 1、子类的编写者必须了解父类的实现。2、父类的改动很容易波及到子类。 用IOC组合方式进行就漂亮得多，把子类要实现的方法用interface来描述。这时两个父子关系的类就转变成调用与被调用的关系，之间通过interface形成契约。”

```
public class ApplicationContext{
    Reader reader;

    public void setReader(Reader reader){this.reader = reader;}

    public void build(){
        reader.getInputStream(path);
    }
}

public interface Reader{
    public InputStream getStream(String path);
}
```

调用代码：

```
ApplicationContext context = new ApplicationContext("path");
context.setReader(new ReaderImpl());
context.build();
```

大B：“这两种实现最大的区别是ClassPathContext和ApplicationContext 是紧耦合的。而ReaderImpl和ApplicaitionContext是正交的。其次就是IOC方式的实现代码要麻烦，这是Java严谨的语言机制决定的，如果用C# 的Delegate Method来实现要简洁许多。”

20.6 模板方法模式与控制反转

小A：“怎样去理解模板方法模式与控制反转？”

大B：“‘不要给我们打电话，我们会给你打电话’这是著名的好莱坞原则。在好莱坞，把简历递交给演艺公司后就只有回家等待。由演艺公司对整个娱乐项的完全控制，演员只能被动式的接受公司的差使，在需要的环节中，完成自己的演出。模板方法模式充分的体现了‘好莱坞’原则。由父类完全控制着子类的逻辑，这就是控制反转。子类可以实现父类的可变部份，却继承父类的逻辑，不能改变业务逻辑。”

20.7 模板方法模式与开闭原则

小A：“什么是“开闭原则”？”

大B：“是指一个软件实体应该对扩展开放，对修改关闭。也就是说软件实体必须是在不被修改的情况下被扩展。模板方法模式意图是由抽象父类控制顶级逻辑，并把基本操作的实现推迟到子类去实现，这是通过继承的手段来达到对象的复用，同时也遵守了开闭原则。父类通过顶级逻辑，它通过定义并提供一个具体方法来实现，我们也称之为模板方法。通常这个模板方法才是外部对象最关心的方法。在上面的订单处理例子中，`public Order placeOrder(int customerId, List orderItemList)`这个方法才是外部对象最关心的方法。所以它必须是public的，才能被外部对象所调用。子类需要继承父类去扩展父类的基本方法，但是它也可以

覆写父类的方法。如果子类去覆写了父类的模板方法，从而改变了父类控制的顶级逻辑。这违反了‘开闭原则’。我们在使用模板方法模式时，应该总是保证子类有正确的逻辑。所以模板方法应该定义为final的。 所以AbstractOrder 类的模板方法placeOrder方法应该定义为final。”

代码

```
public final Order placeOrder(int customerId , List orderItemList)
render_code();
```

大B：“因为子类不能覆写一个被定义为final的方法。从而保证了子类的逻辑永远由父类所控制。模板方法模式中，抽象类的模板方法应该声明为final的。”

20.8 模板方法模式与对象的封装性

小A：“又应该怎样去理解模板方法模式与对象的封装性？”

大B：“面向对象的三大特点：继承，封装，多态。对象有内部状态和外部的行为。封装是为了信息隐藏，通过封装来维护对象内部数据的完整性。使得外部对象不能够直接访问一个对象的内部状态，而必须通过恰当的方法才能访问。在Java语言中，采用给对象属性和方法赋予指定的修改符(public, protected, private)来达到封装的目的，使得数据不被外部对象恶意的访问及方法不被错误调用从而破坏对象的封装性。降低方法的访问级别，也就是最大化的降低方法的可见度是一种很重要的封装手段。最大化降低方法的可见度除了可以达到信息隐藏外，还能有效

的降低类之间的耦合度，降低一个类的复杂度。还可以减少开发人员发生的错误调用。一个类应该只公开外部需要调用的方法。而所有为公开方法服务的方法都应该声明为protected或private。如是一个方法不是需要对外公开的方法，但是它需要被子类进行扩展的或调用。那么把它定义为protected. 否则应该为private。显而易见，模板方法模式中的声明为abstract的基本操作都是需要迫使子类去实现的，它们仅仅是为模板方法placeOrder服务的。它们不应该被AbstractOrder所公开，所以它们应该protected。”

代码

```
protected abstract int getOrderItemPrice(OrderItem orderItem);  
protected abstract int getSpendingLimit(int customerId);  
protected abstract int saveOrder(int customerId, int total, List orderItemList);  
render_code();
```

模板方法模式中，基本方法应该声明为protected abstract

20.9 模板方法模式与钩子方法(hookMethod)

大B：“刚才讨论模板方法模式运用于一个业务对象。事实上，框架频繁使用模板方法模式，使得框架实现对关键逻辑的集中控制。”

小A：“我们需要为基本Spring的应用做一个测试用例的基类. 用于对类的方法进行单元测试。我们知道Spring应用把需要用到的对象都定义在外部的xml文件中，也称为context。”

大B：“通常我们会把context分割成多个小的文件，以便于管理。在测试时我们需要读取context文件，但是并不是每次都读取所有的文件。读取这些文件是很费时间的。所以我们想把它缓存起来，只要这个文件被读取过一次，我们就把它们缓存起来。所以我们通过扩展Junit的TestCase类来完成一个测试基类。我们需要实现缓存的逻辑，其它开发人员只需要实现读取配置文件的方法即可。它不用管是否具有缓存。”

代码

```
public AbstractCacheContextTests extends TestCase{
    private static Map contextMap = new HashMap();
    protected ConfigurableApplicationContext applicationContext;
    protected boolean hasCachedContext(Object contextKey) {
        return contextKeyToContextMap.containsKey(contextKey);
    }
    protected ConfigurableApplicationContext getContext(Object key) {
        String keyString = contextKeyString(key);
        ConfigurableApplicationContext ctx =
            (ConfigurableApplicationContext) contextKeyToContextMap.get(keyString);
        if (ctx == null) {
            if (key instanceof String[]) {
                ctx = loadContextLocations((String[]) key);
            }
            contextKeyToContextMap.put(keyString, ctx);
        }
        return ctx;
    }
    protected String contextKeyString(Object contextKey) {
        if (contextKey instanceof String[]) {
```

```

return StringUtils.arrayToCommaDelimitedString((String[]) contextKey);
}
else {
return contextKey.toString();
}
}

protected ConfigurableApplicationContext loadContextLocations(String[] locations) {
return new ClassPathXmlApplicationContext(locations);
}

//覆写TestCase的setUp方法，在运行测试方法之前从缓存中读取context文件,如果缓存中不存在则初始化applicationContext,并放入缓存。

protected final void setUp() throws Exception {
String[] contextFiles = getConfigLocations();
applicationContext = getContext(contextFiles);
}

//读取context文件，由子类实现

protected abstract String[] getConfigLocations();
}

```

大B：“render_code();这样子类只需要去实现getConfigLocations方法，提供需要读取的配置文件字符数组就可以了。至于怎么去读取context文件内容，怎么实现缓存，则无需关心。 AbstractCacheContextTests保证在运行所有测试之前去执行读取context文件的动作。 注意这里setUp方法被声明为protected,是因为setUp方法是TestCase类的方法。在这里setUp方法被定义为final了，是确保子类不能去覆写这个方法，从而保证了父类控制的逻辑。”

小A：“如果使用过Junit会发生什么问题？”

大B：“TestCase的setUp方法，就是在这个测试类的测试方法运行之前作一些

初始化动作。如创建一些所有测试方法都要用到的公共对象等。我们在这里把setUp方法声明为final之后，子类再也无法去扩展它了，子类同时还需要一些额外的初始化动作就无法实现了。可能你会说：‘把setUp方法的final修饰符去掉就可以了啊’。这样是可以的，但是去掉final修饰符后，子类是可以覆写setUp方法，而去执行一些额外的初始化。而可怕的是，父类从此失去了必须读取context文件及缓存context内容的逻辑。为了解决这个问题，我们实现一个空方法onSetUp。在setUp方法中调用onSetUp方法。这样子类就可以通过覆写onSetUp方法来进行额外的初始化。”

//覆写TestCase的setUp方法，在运行测试方法之前从缓存中读取context文件,如果缓存中不存在则初始化applicationContext,并放入缓存.

代码

```
protected n class="keyword">final void setUp() throws Exception {
    String[] contextFiles = getConfigLocations();
    applicationContext = getContext(contextFiles);
    onSetUp();
}
protected void onSetUp(){
}
//读取context文件，由子类实现
protected abstract String[] getConfigLocations();
}
render_code();
```

小A：“为什么不把onSetUp声明为abstract呢？”

大B：“这是因为子类不一定总是需要覆写onSetUp方法。可以说onSetUp方法是为了对setUp方法的扩展。像onSetUp这样的空方法就称之为勾子方法(HookMethod)”

20.10 模板方法模式与策略模式

小A：“模板方法模式与策略模式有什么不同？”

大B：“模板方法模式与策略模式的作用相常类似。有时可以用策略模式替代模板方法模式。模板方法模式通过继承来实现代码复用，策略模式使用委托，委托比继承具有更大的灵活性。继承经常被错误的使用。策略模式把不确定的行为集中到一个接口中，并在主类委托这个接口。”

思考刚才的订单处理例子，改为策略模式后。

1、把不确定的行为抽取为一个接口。

代码

```
Public interface OrderHelper{  
    public int getOrderItemPrice(OrderItem orderItem);  
    public int getSpendingLimit(int customerId);  
    public int saveOrder(int customerId, int total, List orderItemList);  
}  
render_code();
```

2、而把这个具体类调用这个接口的相应方法来实现具体的逻辑。

代码

```
public class Order {
    private OrderHelper orderHelper;

    public void setOrderHelper(OrderHelper orderHelper){
        this.orderHelper = orderHelper;
    }

    public Order placeOrder(int customerId , List orderItemList){
        int total = 0;
        for(int i = 0; i < orderItemList.size(); i++){
            if(orderHelper.getSpendingLimit(customerId) < total){
                throw new BusinessException( "超出信用额度" + orderHelper.getSpendingLimit(customerId));
            }
            total += orderHelper.getOrderPrice(orderItemList.get(i));
        }
        int orderId = orderHelper.saveOrder(customerId, total, orderItemList);
        return new OrderImpl(orderId,total);
    }
}

render_code();
```

大B：“这样Order类不再是一个抽象类，而是一个具体类。Order类委托OrderHelper接口来完成placeOrder方法所需的基本操作。像在这种情况下使用策略模式更具有优势，策略模式不需要继承来实现。而是通过一个委托对象来实现。OrderHelper接口无需要去继续任何指定的类。而相对来说，采用策略来实现会更复杂一些。由此可见，模板方法模式主要应用于框架设计中，以确保基类控制处理流程的逻辑顺序（如框架的初始化）。像上面的测试基类中。框架通常需要控制反转。而在一些情况中，优先级先考虑使用策略模式：当需要变化的操作非常多

时，采用策略模式把这些操作抽取到一个接口。当那些基本操作的实现需要与其它类相关时，应该使用策略模式。通过委托接口把行为与实现完全分离出来（比如数据存取）。比如订单处理的saveOrder方法，是写入数据库的。它的实现与你采用何种持久化模式相关。当某些基本操作的实现可能需要在运行时改变时，可以通过在运行时改变委托对象来实现，而继承则不能。所以采用策略模式。”

20.11 支持在屏幕上绘图的类View

大B：“我给你举个例子，你就可以更好在理解模板方法模式了。”

小A：“好。”

大B：“一个支持在屏幕上绘图的类View。一个视图只有在进入焦点状态后时才可以设定合适的特定绘图状态，因而只有成为‘焦点’之后才可以进行绘图。View类强制其子类遵循这个规则。我们用Display模板方法来解决这个问题。View定义两个具体方法，SetFocus和ResetFocus,分别设定和清除绘图状态。View的DoDisplay钩子操作实施真正的绘图功能。”

参考图20-2结构图

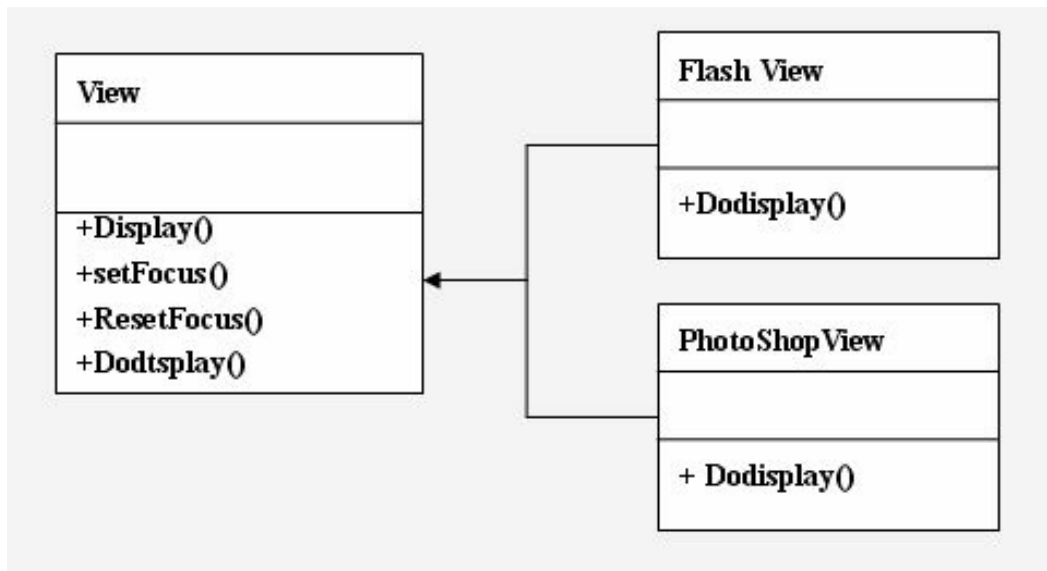


图20-2 结构图

程序代码：

```

#include
class View
{
public:
void Display()
{
//      cout<<"模版方法定义算法框架"<<"获得焦点"<<"失去焦点"<<"基类绘图函数"<<"实现falsh绘图"<<"实现
photoshop绘图"<<"Display()";
pview=new PhotoShopView;
pview->Display();
return 1;
}
}
  
```

运行结果：

获得焦点

实现falsh绘图

失去焦点

获得焦点

实现photoshop绘图

失去焦点

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质
电子书下载！！！！

第二十一章 金融危机何时休——状态模式

21.1 金融危机何时休

时间：1月6日 地点：大B房间 人物：大B，小A

小A：“现在每天一打开电视、报纸，铺天盖地而来的全是这次‘愈演愈烈’的金融危机！”

大B：“这次危机预计要到2009年上半年股市才会平缓，世界经济要恢复则要数年左右，中国大概3年左右，美国可能需十数年。此次危机已造成世界金融也蒸发了35万亿美元左右。全球约有上千万人在此次危机中失业，世界各国投入的救市资金已经超过3万亿美元，将来会更多。”

小A：“是啊！中国股市缩水60%市值蒸发20万亿RMB，实体经济也开始大规模受损，房产市场冷淡，中小企业艰难度日，大型企业融资困难，消费市场缩减，继而冲击其他与以上有关的每个行业，迫使中国政府决定在2010年左右投入4万亿RMB用于救市和刺激消费，但总体上对中国的影响并不大。”

大B：“这是因为中国前三季度的数据显示中国还是保持了较高的增长率，社会固定资产增加值超过了去年同期水平发展虽然有所放缓，但经济发展水平还是很高

的。虽然金融不景气但是中国金融资产的占有国民经济总量中很少。同时中国自身的消费市场巨大，我们的正常生活不会受太大影响。”

小A：“这是因为中国政府的应对政策，除了拿出4万亿用于就是之外，银行将进行数次降息，同时对陷入困境的中小企提供贷款，和减税政策，为大型企业提供融资渠道。”

大B：“追寻此次危机的根源是现有的金融体制存在严重缺陷，所以这次危机过后肯定会制定新的金融经济秩序。而中国将在其中发挥主导作用，中国很可能借助此次千载难逢的机遇，美欧经济不景气，美元这一世界货币疲软的机遇崛起世界经济新格局将因此形成。”

小A：“现在大家都在相互疑问：金融危机何时休啊？”

21.2 状态模式

金融危机就是我们所说状态模式。

小A：“什么是状态模式？”

大B：“允许一个对象在其内部状态改变时改变它的行为。这个对象看起来似乎修改了它的类。看起来，状态模式好像是神通广大——居然能够‘修改自身的类’！能够让程序根据不同的外部情况来做出不同的响应，最直接的方法就是在程序中将这些可能发生的外部情况全部考虑到，使用if else 语句来进行代码响应选择。但是这种方法对于复杂一点的状态判断，就会显得杂乱无章，容易产生错误；

而且增加一个新的状态将会带来大量的修改。这个时候 ‘能够修改自身’ 的状态模式的引入也许是个不错的主意。”

21.3 状态模式的角色

小A：“状态模式由哪些角色组成呐？”

大B：“状态模式可以有效的替换充满在程序中的if else语句：将不同条件下的行为封装在一个类里面，再给这些类一个统一的父类来约束他们。状态模式是由下面几种角色组成：1、使用环境（Context）角色：客户程序是通过它来满足自己的需求。它定义了客户程序需要的接口；并且维护一个具体状态角色的实例，这个实例来决定当前的状态。2、状态（State）角色：定义一个接口以封装与使用环境角色的一个特定状态相关的行为。3、具体状态（Concrete State）角色：实现状态角色定义的接口。结构非常简单也与策略模式非常相似。”

参考图21-1状态模式类图

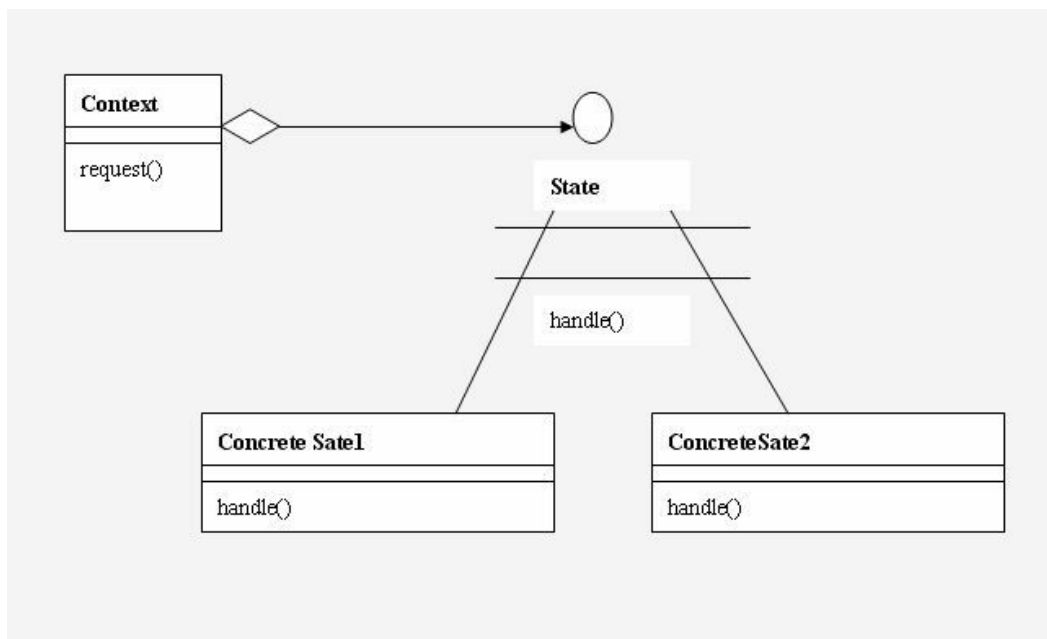


图21-1 状态模式类图

21.4 状态模式的特征

小A：“状态模式都有些什么特征？”

大B：“抽象状态角（State）色：定义一个接口，用以封装语境（Context）对象的一个特定的状态所对应的行为。具体状态（ConcreteState）角色：一个具体状态类，实现了语境（Context）对象的一个状态所对应的行为。语境（Context）角色：定义客户端所感兴趣的接口，并且保留一个具体状态类的实例，这个具体状态类的实例给出此环境对象的现有状态。通过使用多态，可以动态地改变语境（Context）对象的属性State的内容，使其从指向一个具体状态类到指向另外一个具体状态类，从而改变语境（Context）对象的行为。”

21.5 状态模式优点

小A：“状态模式有什么优点？”

大B：“1、封装转换过程，也就是转换规则。2、枚举可能的状态，因此，需要事先确定状态种类。”

21.6 状态模式实质

小A：“状态模式的实质是什么？”

大B：“使用状态模式前，客户端外界需要介入改变状态，而状态改变的实现是琐碎或复杂的。使用状态模式后，客户端外界可以直接使用事件Event实现，根本不必关心该事件导致如何状态变化，这些是由状态机等内部实现。这是一种Event-condition-State，状态模式封装了condition-State部分。每个状态形成一个子类，每个状态只关心它的下一个可能状态，从而无形中形成了状态转换的规则。如果新的状态加入，只涉及它的前一个状态修改和定义。”

小A：“都有些什么方法实现状态转换？”

大B：“一个在每个状态实现next()，指定下一个状态；还有一种方法，设定一个StateOwner，在StateOwner设定stateEnter状态进入和stateExit状态退出行为。状态从一个方面说明了流程，流程是随时间而改变，状态是截取流程某个时间片。”

21.7 状态模式和策略模式的比较

小A：“怎样去比较状态模式和策略模式？”

大B：“在状态模式中，状态的变迁是由对象的内部条件决定，外界只需关心其接口，不必关心其状态对象的创建和转化；而策略模式里，采取何种策略由外部条件(C)决定。Strategy模式与State模式的结构形式几乎完全一样。但它们的应用场景（目的）却不一样，State模式重在强调对象内部状态的变化改变对象的行为，Strategy模式重在外部对策略的选择，策略的选择由外部条件决定，也就是说算法的动态的切换。但由于它们的结构是如此的相似，我们可以认为状态模式是完全封装且自修改的策略模式。”

小A：“公认的事实：策略和状态模式是孪生兄弟。”

大B：“就像你所知道的，策略模式通过可互换的算法规则来创建非常成功的业务模式。不管怎么样，状态以非常高尚的方式帮助对象学习通过他们内部的状态来控制他们的行为。他总是无意中告诉他的对象客户，‘跟着我重复就行了，我足够好，我足够聪明...’”

21.8 何时使用状态模式？

小A：“什么时候使用状态模式？”

大B：“State模式在实际使用中比较多，适合‘状态的切换’。因为我们经常

会使用If elseif else 进行状态切换，如果针对状态的这样判断切换反复出现，我们就要联想到是否可以采取State模式了。不只是根据状态，也有根据属性。如果某个对象的属性不同，对象的行为就不一样，这点在数据库系统中出现频率比较高，我们经常会在一个数据表的尾部，加上property属性含义的字段，用以标识记录中一些特殊性质的记录，这种属性的改变(切换)又是随时可能发生的，就有可能要使用State 。”

21.9 是否使用状态模式?

小A：“要怎么去决定是否使用状态模式?”

大B：“在实际使用，类似开关一样的状态切换是很多的，但有时并不是那么明显，取决于你的经验和对系统的理解深度。这里要说的是‘开关切换状态’和‘一般的状态判断’是有一些区别的，‘一般的状态判断’也是有 if..elseif结构。”

例如：

```
if (which==1) state="hello";  
else if (which==2) state="hi";  
else if (which==3) state="bye";
```

大B：“这是一个‘一般的状态判断’，state值的不同是根据which变量来决定的，which和state没有关系。”

如果改成：

```
if (state.equals("bye")) state="hello";  
else if (state.equals("hello")) state="hi";  
else if (state.equals("hi")) state="bye";
```

大B：“这就是‘开关切换状态’，是将state的状态从‘hello’切换到‘hi’，再切换到‘bye’；在切换到‘hello’，好象一个旋转开关，这种状态改变就可以使用State模式了。如果单纯有上面一种将‘hello’-->‘hi’-->‘bye’-->‘hello’这一个方向切换，也不一定需要使用State模式，因为State模式会建立很多子类，复杂化，但是如果又发生另外一个行为：将上面的切换方向反过来切换，或者需要任意切换，就需要State了。”

请看下例：

```
public class Context{  
    private Color state=null;  
    public void push(){  
        //如果当前red状态 就切换到blue  
        if (state==Color.red) state=Color.blue;  
        //如果当前blue状态 就切换到green  
        else if (state==Color.blue) state=Color.green;  
        //如果当前black状态 就切换到red  
        else if (state==Color.black) state=Color.red;  
        //如果当前green状态 就切换到black  
        else if (state==Color.green) state=Color.black;  
        Sample sample=new Sample(state);  
        sample.operate();  
    }  
}
```



```

}
public void pull(){
//与push状态切换正好相反
if (state==Color.green) state=Color.blue;
else if (state==Color.black) state=Color.green;
else if (state==Color.blue) state=Color.red;
else if (state==Color.red) state=Color.black;
Sample2 sample2=new Sample2(state);
sample2.operate();
}
}

```

大B：“在上例中，我们有两个动作push推和pull拉，这两个开关动作，改变了Context颜色，至此，我们就需要使用State模式优化它。另外注意：state的变化，只是简单的颜色赋值，这个具体行为是很简单的，State适合巨大的具体行为，因此，实际使用中也不一定非要使用State模式，这会增加子类的数目，简单的变复杂。”

例如：银行帐户，经常会在Open 状态和Close状态间转换。

例如：经典的TcpConnection ，Tcp的状态有创建、侦听、关闭三个，并且反复转换，其创建、侦听、关闭的具体行为不是简单一两句就能完成的，适合使用State 。

例如：信箱POP帐号，会有四种状态，start HaveUsername Authorized quit，每个状态对应的行为应该还是比较大的.适合使用State 。

例如：在工具箱挑选不同工具，可以看成在不同工具中切换，适合使用State 。如具体绘图程序，用户可以选择不同工具绘制方框、直线、曲线，这种状态切换

可以使用State 。

21.10 如何使用状态模式?

小A：“如何使用状态模式?”

大B：“状态模式可以允许客户端改变状态的转换行为，而状态机则是能够自动改变状态，状态机是一个比较独立的而且复杂的机制，具体可参考一个状态机开源项目：

状态模式在工作流或游戏等各种系统中有大量使用，甚至是这些系统的核心功能设计，例如政府OA中，一个批文的状态有多种：未办；正在办理；正在批示；正在审核；已经完成等各种状态，使用状态机可以封装这个状态的变化规则，从而达到扩充状态时，不必涉及到状态的使用者。在网络游戏中，一个游戏活动存在开始；开玩；正在玩；输赢等各种状态，使用状态模式就可以实现游戏状态的总控，而游戏状态决定了游戏的各个方面，使用状态模式可以对整个游戏架构功能实现起到决定的主导作用。

State需要两种类型实体参与：1、state manager 状态管理器，就是开关，如上面例子的Context实际就是一个state manager在state manager中有对状态的切换动作。2、用抽象类或接口实现的父类，不同状态就是继承这个父类的不同子类。”

以上面的Context为例。我们要修改它，建立两个类型的实体。

第一步：

首先建立一个父类：

```
public abstract class State{  
    public abstract void handlepush(Context c);  
    public abstract void handlepull(Context c);  
    public abstract void getcolor();  
}
```

父类中的方法要对应state manager中的开关行为，在state manager中这个例子就是Context中，有两个开关动作push推和pull拉。那么在状态父类中就要有具体处理这两个动作：handlepush() handlepull()；同时还需要一个获取push或pull结果的方法getcolor()

下面是具体子类的实现：

```
public class BlueState extends State{  
    public void handlepush(Context c){  
        //根据push方法"如果是blue状态的切换到green" ;  
        c.setState(new GreenState());  
    }  
    public void handlepull(Context c){  
        //根据pull方法"如果是blue状态的切换到red" ;  
        c.setState(new RedState());  
    }  
    public abstract void getcolor(){ return (Color.blue)}  
}
```

同样 其他状态的子类实现如blue一样。

第二步：

要重新改写State manager 也就是本例的Context：

```
public class Context{
    private Sate state=null; //我们将原来的 Color state 改成了新建的State state;
    //setState是用来改变state的状态 使用setState实现状态的切换
    public void setState(State state){
        this.state=state;
    }
    public void push(){
        //状态的切换的细节部分,在本例中是颜色的变化,已经封装在子类的handlepush中实现,这里无需关心
        state.handlepush(this);
        //因为sample要使用state中的一个切换结果,使用getColor()
        Sample sample=new Sample(state.getColor());
        sample.operate();
    }
    public void pull(){
        state.handlepull(this);
        Sample2 sample2=new Sample2(state.getColor());
        sample2.operate();
    }
}
```

至此，我们也就实现了State的refactorying过程。

以上只是相当简单的一个实例，在实际应用中，handlepush或handlepull的处理是复杂的。

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质
电子书下载！！！！

第二十二章 还钱——策略模式

22.1 还钱

时间：1月7日 地点：大B家 人物：大B和大B的朋友小王

小王两年前借大B的钱至今未还，正好这天小王来大B家做客（正是要帐的好机会）。

首先旁敲侧击法

大B：“前两天我见到咱们的老同学小李了，他还了我1000元钱，说是两年前借的，他要是不还呀，我都忘了。”

小王：“是呀！小李的记性就是好，比我强多了。”

居然没奏效，下一招诱蛇出洞

大B：“听说你最近生意不错呀！怎么样成本都收回来了嘛？债务都还上了吗？”

小王：“还算红火，债务基本上还清了。”

看来只能单刀直入了

大B：“那你借我的2000元也该还了吧！”

22.2 策略模式

小A：“这年头借钱的真的是孙子，欠钱的是老子。”

大B：“是啊！难呐！这就是我要讲的策略模式了。”

小A：“喔？策略模式？什么是策略模式啊？”

小A：“策略模式就是定义一系列的算法，把他们一个个封装起来，并且使它们可相互替换。Strategy模式使算法可独立于使用它的客户而变化。”

22.3 策略模式涉及的角色

小A：“它所涉及到哪些角色呢？”

大B：“策略模式中分成三种角色。抽象策略角色：通常用一个抽象类或者接口来实现，主要是定义这个算法所完成的功能。具体策略角色：包装了相关算法和行为。环境角色：持有策略类的引用。”

下面我们还是看一个小例子，很多时候看代码更能够懂得其中的意思，不是那么抽象，这个例子要实现的功能是加减乘除。

首先建立抽象策略角色：

```
Operation.java
package org.kangta.straty;

/**
 *
 * @author Administrator
 * 抽象策略角色
 *
 */
public interface Operation {
    public void op(double a,double b);
}
```

再建立具体策略角色：四个Add.java、 Sub.java、 Div.java、 Multi.java

```
Add.java
package org.kangta.straty;

/**
 * 具体策略角色
 * @author Administrator
 *
 */
public class Add implements Operation {
    public void op(double a, double b) {
        // TODO Auto-generated method stub
        double result = a + b;
        System.out.println(result);
    }
}
```


Sub.java

```
package org.kangta.straty;

/**
 * 具体策略角色
 * @author Administrator
 *
 */

public class Sub implements Operation {
    public void op(double a, double b) {
        // TODO Auto-generated method stub
        double result = a - b;
        System.out.println(result);
    }
}
```

Div.java

```
package org.kangta.straty;

/**
 * 具体策略角色
 * @author Administrator
 *
 */

public class Div implements Operation {
    public void op(double a, double b) {
        // TODO Auto-generated method stub
        if(b != 0)
        {
            double result = a / b;
            System.out.println(result);
        }
        else
        {
            System.out.println("除0了!");
        }
    }
}
```

```

}
}
}
Multi.java
package org.kangta.straty;

/**
 * 具体策略角色
 * @author Administrator
 *
 */
public class Multi implements Operation {
    public void op(double a, double b) {
        // TODO Auto-generated method stub
        double result = a * b;
        System.out.println(result);
    }
}

```

OK！抽象策略角色和具体策略角色都已经建立成功了，现在来建立环境角色

```

Calc.java
package org.kangta.straty;

/**
 * 环境角色
 * @author Administrator
 *
 */
public class Calc {
    public final static Add add = new Add();
    public final static Sub sub = new Sub();
    public final static Div div = new Div();
}

```

```
public final static Multi multi = new Multi();  
}
```

都建立好了测试一下

Test.java

```
package org.kangta.straty.test;  
import org.kangta.straty.Calc;  
public class Test {  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Calc c = new Calc();  
        c.add.op(11,22);  
        c.sub.op(22,11);  
        c.div.op(33, 11);  
        c.multi.op(33, 33);  
    }  
}
```

测试成功

点评策略模式：

策略模式的优点：

提供管理相关算法族的办法

提供可替代继承关系的办法

避免了使用多重条件判断语句

策略模式的缺点：

客户端必须知道所有的策略类，自己去决定使用那一个

造成很多策略类

22.4 购物车系统

大B：“假设现在要设计一个贩卖各类书籍的电子商务网站的购物车（Shopping Cat）系统。一个最简单的情况就是把所有货品的单价乘上数量，但是实际情况肯定比这要复杂。”

小A：“一般会有哪些情况哩？”

大B：“比如：1、站可能对所有的儿童类图书实行每本一元的折扣；2、计算机类图书提供每本7%的促销折扣，而对电子类图书有3%的折扣；3、其余的图书没有折扣。4、还会有新的打折策略。由于有这样复杂的折扣算法，使得价格计算问题需要系统地解决。”

方案一：业务逻辑放在各具体子类

/*

*各子类实现销售价格算法

```

*/

public abstract class Book {
    private double price;
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    public abstract double getSalePrice() ;
}

public class CsBook extends Book{
    public CsBook(String name,double price)
    {
        this.setName(name);
        this.setPrice(price);
    }
    public double getSalePrice()
    {
        return this.getPrice()*0.93;
    }
}

public class ChildrenBook extends Book {
    public ChildrenBook(String name, double price) {
        this.setName(name);

```

```

    this.setPrice(price);
}
public double getSalePrice() {
    return this.getPrice() - 1;
}
}

public class Client {
    public static void main(String args[])
    {
        Book csBook1=new CsBook("Think in java",45);
        Book childrenBook1=new ChildrenBook("Hello",20);
        System.out.println(csBook1.getName()+":"+csBook1.getSalePrice());
        System.out.println(childrenBook1.getName()+":"+childrenBook1.getSalePrice());
    }
}

```

问题：每个子类必须都各自实现打折算法，即使打折算法相同。所以code reuse不好

方案二：

```

//把打折策略代码提到父类来实现code reuse
public abstract class Book {
    private double price;
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}

```

```

}
public double getPrice() {
return price;
}
public void setPrice(double price) {
this.price = price;
}
//销售价格
public static double toSalePrice(Book book)
{
if (book instanceof ChildrenBook)
{
return book.getPrice()-1;
}
else if (book instanceof CsBook)
{
return book.getPrice()*0.93;
}
return 0;
}
}
public class Client {
public static void main(String args[])
{
Book csBook1=new CsBook("Think in java",45);
Book childrenBook1=new ChildrenBook("Hello",20);
System.out.println(csBook1.getName()+":"+Book.toSalePrice(csBook1));
System.out.println(childrenBook1.getName()+":"+Book.toSalePrice(childrenBook1));
}
}

```

toSalePrice方法是比较容易change的地方，如果策略复杂用if判断比较乱，并且策略修改或增加时需改变原代码。

方案三：策略模式

code reuse时最好用合成(HAS-A)而不用(IS-A)，更加灵活。

```
public abstract class Book {
    private double price;
    private String name;
    private DiscountStrategy discountStrategy;//折扣策略
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    public DiscountStrategy getDiscountStrategy() {
        return discountStrategy;
    }
}
```

22.5 策略模式的优点、缺点

大B：“如果具体策略类有一些共同的行为，则应该把它们抽取到抽象策略角色中，此时抽象策略角色为抽象类。策略模式的优点：a、提供了管理相关算法族的方法。b、可以避免使用多重条件转移语句。”

小A：“那策略模式有什么缺点？”

大B：“首先是必须知道所有的具体策略类及它们的区别，再生成许多的策略类。”

22.6 使用场合

大B：“一般会在哪些场合使用它？”

小A：“1、系统有许多类，而他们的区别仅仅在于它们的行为。2、动态选择几种算法中的一种。3、一个对象有很多行为。”

22.7 实现步骤

大B：“要注意策略模式的实现步骤喔。”

小A：“实现步骤？”

大B：“对啊。1、定义抽象角色类，定义好各个实现的共同抽象方法。2、定义具体策略类，具体实现父类的共同方法。3、定义环境角色类，私有化申明抽象角色

变量，重载构造方法，执行抽象方法。”

代码：

```
abstract public class Tool {
    abstract public void setup();
}

public class DevelopTool extends Tool{
    @Override
    public void setup() {
        System.out.println("develope tool setup");
    }
}

public class ApplicationTool extends Tool {
    @Override
    public void setup() {
        System.out.println("application tool setup");
    }
}

public class ToolSetUp {
    private Tool tool;
    public ToolSetUp(Tool tool) {
        this.tool = tool;
    }
    public void setup() {
        tool.setup();
    }
}
```

22.8 策略模式-状态模式/模版模式的区别

小A：“状态模式和策略模式有什么不同？”

大B：“状态模式侧重状态方面，一般不会接受新的状态对象，即系统已经定义足够的状态。策略侧重不同的行为的改变在统一的接口下，强调多态下面行为的执行过程，处理过程，可以从用户那里接受参数，只要用户提供的策略符合接口。”

小A：“与模版模式又有什么不同呢？”

大B：“模版模式就是算法在父类中，子类不会完全改写算法，可以改写部分，或称关键部分，但整体的算法不变，可以节省大量代码。策略模式所有的算法均在子类中完成，强调行为即算法的不同，可以使程序更灵活。”

22.9 策略模式的应用

小A：“策略模式应该怎么去应用它？”

大B：“1、如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态的让一个对象在许多行为中选择一种行为。2、如果系统需要动态地在几种算法中选择一种。那么这些算法可以包装到一个个的具体算法类里面，而这些算法类都是一个抽象算法类的子类。3、一个系统的算法使用的数据不可以让客户端知道。策略模式可以避免让客户端涉及到不必要接触到的复发的和只与算法有关的数据。”

使用案例：

AWT中的LayoutManager, Swing中的Border.

代码例子

1、抽象策略

```
package com.eekq.strategy;

public interface IStrategy {
    /**策略方法*/
    public abstract double add();
}
```

2、具体策略，这里我以两个具体策略为例

```
package com.eekq.strategy;

public class ConcreteStrategy1 implements IStrategy{
    /**示意性算法*/
    public double add(){
        //TODO自动生成方法存根
        System.out.println(this.getClass().getName() + "的加法运算");
        return 0;
    }
}

package com.eekq.strategy;

public class ConcreteStrategy2 implements IStrategy{
    public double add() {
        //TODO自动生成方法存根
        System.out.println(this.getClass().getName() + "的加法运算");
    }
}
```

```
return 0;
}
}
```

3、环境角色

```
package com.eekq.strategy;
public class Context {
    /**环境角色类*/
    private IStrategy strategy;
    public Context(IStrategy strategy) {
        this.strategy = strategy;
    }
    /**策略方法*/
    public double add() {
        this.strategy.add();
        return 0;
    }
}
```

4、客户端调用

```
package com.eekq.strategy;
public class Main {
    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO 自动生成方法存根
    }
}
```

```
Context context = new Context(new ConcreteStrategy1());
context.add();//执行算法1
context = new Context(new ConcreteStrategy2());
context.add();//执行算法2
}
}
```

5、执行结果：

```
com.eekq.strategy.ConcreteStrategy1的加法运算
com.eekq.strategy.ConcreteStrategy2的加法运算
```

22.10 简单的模拟鸭子应用学习

大B：“游戏中会出现各种鸭子，一边游戏戏水，一边呱呱叫，我们设计了一个鸭子超类(Duck)，并让各种鸭子继承此类。”

Java代码

```
Public abstract class Duck {
void quack();//呱呱叫
void swin();//游泳
abstract void display();//鸭子外观不同,所以是抽象的
//other
}

Public class MallardDuck extends Duck{
```

```

Public void display(){
//外观是绿头
}
}
Public class RedheadDuck extends Duck {
Public void display(){
//外观是红头
}
}
//other extends
public abstract class Duck {
void quack();//呱呱叫
void swin();//游泳
abstract void display();//鸭子外观不同,所以是抽象的
//other
}
public class MallardDuck extends Duck {
public void display() {
//外观是绿头
}
}
public class RedheadDuck extends Duck {
public void display() {
//外观是红头
}
}
//other extends

```

大B：“现在多了个主意，要让鸭子能飞，那么你可能想到只要在Duck类中加上fly()方法，然后让所有的鸭子都来继承这个方法。”

Java代码

```
Public abstract class Duck {  
Void quack();//呱呱叫  
Void swin();//游泳  
abstract void display();//鸭子外观不同,所以是抽象的  
fly();//刚加的会飞的方法,所有的子类都会继承  
//other  
}  
  
public abstract class Duck {  
void quack();//呱呱叫  
void swin(); //游泳  
abstract void display();//鸭子外观不同,所以是抽象的  
fly();//刚加的会飞的方法,所有的子类都会继承  
//other  
}
```

大B：“但是，可怕的问题就出来了：“塑料橡皮鸭子”也会飞了，忽略了一件事，并非所有的鸭子都会飞。能想到继承，把橡皮鸭子类中的fly()方法覆盖掉，什么也不做。可是，如果以后要加入木头鸭子，不会飞也不会叫，你又想到了，把fly()从超类中取出来，做一个Flyable()接口，同样的方式，设计一个会叫的接口Quackable()。”

小A：“但是你没发现这么一来重复的代码会很多吗？”

大B：“现在我们知道使用继承并不能很好的解决问题，因为鸭子的行为在子类里不断的改变，让所有的子类都有这些行为是不恰当的。”

第一个设计原则：

找出应用中可能需要变化之处，把他独立出来，不要和那些不需要变化的代码混在一起。现在知道Duck()类里的fly()和quack()会随着鸭子的不同而改变，我们把它从Duck()类里取出来，写2个接口：

Java代码

```
public interface FlyBehavior{
    public void fly();//飞行行为必须实现的接口
}

public interface QuackBehavior{
    public void quack();//叫行为必须实现的接口
}

public interface FlyBehavior{public void fly();//飞行行为必须实现的接口
}

public interface QuackBehavior{
    public void quack();//叫行为必须实现的接口
}
```

这样的设计可以让这2个动作行为与鸭子类无关，可以被其他的对象复用，也可以新增加一些行为。

第二个设计原则：

针对接口编程，而不是针对实现编程。

现在来整合下鸭子的行为：

Java代码

1、Duck()抽象类

```

public abstract class Duck {
//为行为接口类型声明2个引用变量,所有的鸭子子类都继承它们
FlyBehavior flyBehavior;
QuackBehavior quackBehavior;
public Duck(){
}
//设定鸭子的行为,而不是在鸭子的构造器内实例化
public void setFlyBehavior (FlyBehavior fb) {
flyBehavior =fb;
}
public void setQuackBehavior(QuackBehavior qb){
quackBehavior=qb;
}
abstract void display();//抽象外观方法
public void performFly() {
flyBehavior.fly();//委托给会飞的行为类
}
public void performQuack(){
quackBehavior.quack();//委托给会叫的行为类
}
public void swim(){
System.out.println("All ducks float, even decoys!");
}
}

```

2、FlyBehavior()接口与2个行为实现类

```

public interface FlyBehavior {
public void fly();
}

```

```

public class FlyNoWay implements FlyBehavior {
    public void fly(){
        System.out.println("I can't fly");
    }
}

public class FlyWithWings implements FlyBehavior {
    public void fly(){
        System.out.println("I'm flying!!");
    }
}

```

3、QuackBehavior()接口与2个行为实现类

```

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(">");
    }
}

```

4、来看看如何设定FlyBehavior和QuackBehavior的实例变量(继承)

```

public class MallardDuck extends Duck {
    public MallardDuck() {
        //绿头鸭子使用Quack类处理呱呱叫,所以当performQuack被调用时,叫的责任被委托给Quack对象,
        //而我们得到了真正的呱呱叫
    }
}

```

```
quackBehavior=newQuack();
//使用FlyWithWings作为其FlyBehavior类型
flyBehavior=newFlyWithWings();
}
public void display(){
System.out.println("I'm a real Mallard duck");
}
}
```

5、制造一个新的鸭子类型

```
public class ModelDuck extends Duck{
public ModelDuck() {
flyBehavior=newFlyNoWay();//一开始我们的鸭子是不会飞的
quackBehavior=newQuack();
}
public void display(){
System.out.println("I'm a model duck");
}
}
```

6、建立一个新的FlyBehavior类型,火箭动力的飞行行为

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

7、测试类

```
public class MiniDuckSimulator {
    public static void main(String[] args) {
        Duck mallard =new MallardDuck();
        //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()方法
        mallard.performQuack();
        mallard.performFly();
        Duck model =new ModelDuck();
        //第一次调用performFly会被委托给flyBahavion对象,
        //也就是FlyNoWay实例,该对象是在模型鸭子构造器中设置的
        model.performFly();
        //这会调用继承来的setter方法,把FlyRocketPowered火箭动力飞行行为设定到模型鸭子中,牛吧...
        model.setFlyBehavior(new FlyRocketPowered());
        //如果成功了,就意味着模型鸭子可以动态的改变它的飞行行为
        model.performFly();
    }
}
```

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第二十三章 饭店点菜——命令模式

23.1 饭店点菜

时间：1月8日 地点：肥羊王 人物：大B和他的女朋友，小A

这天，小A约了大B和他的女朋友一起去肥羊王吃火锅。刚到坐定，一个笑得挺甜美的服务生来了。

肥羊王服务生：“先生，小姐吃点什么？”

大B的女朋友：“来个鸳鸯火锅。”

肥羊王服务生：“好的。请问还要点什么菜？”

大B的女朋友：“来两盘肥羊、猪脑花、鸭血、水发海带、白菜、粉丝、白萝卜、豆腐、金针菇、鱼丸、猪肉丸、牛肉丸各一碟。先上这些，不够我一会再点。”

肥羊王服务生：“好的，马上来，请各位稍等！”

大B的女朋友：“等等，再来三碟辣椒酱。”

肥羊王服务生：“好的，请各位稍等！”

23.2 命令模式

每次和大B的女朋友吃饭都是她负责点菜的，嘿嘿，我们负责吃，每次都吃得饱饱的。

大B：“别光顾着吃，这就是我们讲的命令模式？”

小A：“命令模式？”

大B：“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，以及支持可撤消的操作。”

23.3 命令模式涉及到哪些角色

小A：“命令模式涉及到哪些角色？”

大B：“1、命令角色（Command）：声明执行操作的接口。有Java接口或者抽象类来实现。2、具体命令角色（Concrete Command）：将一个接收者对象绑定于一个动作；调用接收者相应的操作，以实现命令角色声明的执行操作的接口。3、客户角色（Client）：创建一个具体命令对象（并可以设定它的接收者）。4、请求者角色（Invoker）：调用命令对象执行这个请求。5、接收者角色（Receiver）：知道如何实施与执行一个请求相关的操作。任何类都可能作为一个接收者。”

23.4 Struts中Action的使用

大B：“由于在JUnit中，参杂了其它的模式在里面，使得命令模式的特点不太明显。我给你讲讲以命令模式在Web开发中最常见的应用——Struts中Action的使用作为例子。”

小A：“嗯。好的。”

大B：“在Struts中Action控制类是整个框架的核心，它连接着页面请求和后台业务逻辑处理。按照框架设计，每一个继承自Action的子类，都实现execute方法——调用后台真正处理业务的对象来完成任务。”

大B：“需要注意的是：继承自DispatchAction的子类，则可以一个类里面处理多个类似的操作。”

下面我们将Struts中的各个类与命令模式中的角色对号入座。

先来看下命令角色——Action控制类

```
public class Action {  
    .....  
    /*  
    *可以看出，Action中提供了两个版本的执行接口，而且实现了默认的空实现。  
    */  
    public ActionForward execute( ActionMapping mapping,  
    ActionForm form,  
    ServletRequest request,  
    ServletResponse response)
```



```

throws Exception {
try {
return execute(mapping, form, (HttpServletRequest) request,
(HttpServletResponse) response);
} catch (ClassCastException e) {
return null;
}
}

public ActionForward execute( ActionMapping mapping,
ActionForm form,
HttpServletRequest request,
HttpServletResponse response)
throws Exception {
return null;
}
}

```

下面的就是请求者角色，它仅仅负责调用命令角色执行操作。

```

public class RequestProcessor {
.....
protected ActionForward processActionPerform(HttpServletRequest request,
HttpServletResponse response,
Action action,
ActionForm form,
ActionMapping mapping)
throws IOException, ServletException {
try {
return (action.execute(mapping, form, request, response));
} catch (Exception e) {
return (processException(request, response,e, form, mapping));
}
}
}

```

```
}  
}  
}
```

大B：“Struts框架为我们提供了以上两个角色，要使用struts框架完成自己的业务逻辑，剩下的三个角色就要由我们自己来实现了。”

小A：“那要怎么去实现啊？”

大B：“实现的步骤如下：1、很明显我们要先实现一个Action的子类，并重写execute方法。在此方法中调用业务模块的相应对象来完成任务。2、实现处理业务的业务类。3、配置struts-config.xml配置文件，将自己的Action和Form以及相应页面结合起来。4、编写jsp，在页面中显式的制定对应的处理Action。”

23.5 Undo、事务及延伸

大B：“在定义中提到，命令模式支持可撤销的操作。”

小A：“但是在上面的举例中并没有体现出来啊。”

大B：“其实命令模式之所以能够支持这种操作，完全得益于在请求者与接收者之间添加了中间角色。为了实现undo功能，首先需要有一个历史列表来保存已经执行过的具体命令角色对象；修改具体命令角色中的执行方法，使它记录更多的执行细节，并将自己放入历史列表中；并在具体命令角色中添加undo方法，此方法根据记录的执行细节来复原状态，很明显，首先程序员要清楚怎么来实现，因为它和execute的效果是一样的。同样，redo功能也能够照此实现。命令模式还有一个常

见的用法就是执行事务操作。这就是为什么命令模式还叫做事务模式的原因吧。它可以在请求被传递到接收者角色之前，检验请求的正确性，甚至可以检查和数据库中数据的一致性，而且可以结合组合模式的结构，来一次执行多个命令。使用命令模式不仅仅可以解除请求者和接收者之间的耦合，而且可以用来做批处理操作，这完全可以发挥你自己的想象——请求者发出的请求到达命令角色这里以后，先保存在一个列表中而不执行；等到一定的业务需要时，命令模式再将列表中全部的操作逐一执行。”

小A：“哦，命令模式实在太灵活了。真是一个很有用的东西啊！”

23.6 优点

小A：“命令模式都有哪些优点呐？”

大B：“从刚才我和你讲的内容可以看出命令模式有以下优点：1、命令模式将调用操作的请求对象与知道如何实现该操作的接收对象解耦。2、具体命令角色可以被不同的请求者角色重用。3、你可将多个命令装配成一个复合命令。4、增加新的具体命令角色很容易，因为这无需改变已有的类。”

23.7 命令模式的适用环境

小A：“命令模式的适用哪些环境？”

大B：“1、需要抽象出待执行的动作，然后以参数的形式提供出来——类似于过程设计中的回调机制。而命令模式正是回调机制的一个面向对象的替代品。2、在不同的时刻指定、排列和执行请求。一个命令对象可以有与初始请求无关的生存期。3、需要支持取消操作。4、支持修改日志功能。这样当系统崩溃时，这些修改可以被重做一遍。5、需要支持事务操作。”

在此写了7个Java类来描述说明Command设计模式的实现方式；

- 1、 Control.java 命令控制者对象类
- 2、 Tv.java 命令接收者对象类
- 3、 Command.java 命令接口类
- 4、 CommandChannel.java 频道切换命令类
- 5、 CommandOff.java 关机命令类
- 6、 CommandOn.java 开机命令类
- 7、 CommandTest.java 带有main方法的测试类(命令发送者)

===== 1、 Control.java

```
package command;
//命令控制者
public class Control {
    private Command onCommand, offCommand, changeChannel;
    public Control(Command on, Command off, Command channel) {
        onCommand = on;
```

```

offCommand = off;
changeChannel = channel;
}
public void turnOn() {
onCommand.execute();
}
public void turnOff() {
offCommand.execute();
}
public void changeChannel() {
changeChannel.execute();
}
}

```

===== 1 end

===== 2、 Tv.java

```

package command;
//命令接收者
public class Tv {
public int currentChannel = 0;
public void turnOn() {
System.out.println("The televisino is on.");
}
public void turnOff() {
System.out.println("The television is off.");
}
public void changeChannel(int channel) {
this.currentChannel = channel;
System.out.println("Now TV channel is " + channel);
}
}

```

```
}  
}
```

===== 2 end

===== 3、 Command.java

```
package command;  
//命令接口  
public interface Command {  
    void execute();  
}
```

===== 3 end

===== 4、 CommandChannel.java

```
package command;  
//频道切换命令  
public class CommandChannel implements Command {  
    private Tv myTv;  
    private int channel;  
    public CommandChannel(Tv tv, int channel) {  
        myTv = tv;  
        this.channel = channel;  
    }  
    public void execute() {  
        myTv.changeChannel(channel);  
    }  
}
```

```
}
```

```
===== 4 end
```

```
===== 5、 CommandOff.java
```

```
package command;  
//关机命令  
public class CommandOff implements Command {  
    private Tv myTv;  
    public CommandOff(Tv tv) {  
        myTv = tv;  
    }  
    public void execute() {  
        myTv.turnOff();  
    }  
}
```

```
===== 5 end
```

```
===== 6、 CommandOn.java
```

```
package command;  
//开机命令  
public class CommandOn implements Command {  
    private Tv myTv;  
    public CommandOn(Tv tv) {  
        myTv = tv;  
    }  
}
```

```
public void execute() {  
    myTv.turnOn();  
}  
}
```

===== 6 end

===== 7、 CommandTest.java

```
package command;  
//命令发送者  
public class CommandTest{  
    public static void main(String[] args){  
        //命令接收者  
        Tv myTv = new Tv();  
        //开机命令  
        CommandOn on = new CommandOn(myTv);  
        //关机命令  
        CommandOff off = new CommandOff(myTv);  
        //频道切换命令  
        CommandChannel channel = new CommandChannel(myTv, 2);  
        //命令控制对象  
        Control control = new Control( on, off, channel);  
        //开机  
        control.turnOn();  
        //切换频道  
        control.changeChannel();  
        //关机  
        control.turnOff();  
    }  
}
```


23.8 Command模式应用场景

大B：“Command模式通常可应用到以下场景：1、Multi-level undo（多级undo操作）如果系统需要实现多级回退操作，这时如果所有用户的操作都以command对象的形式实现，系统可以简单地用stack来保存最近执行的命令，如果用户需要执行undo操作，系统只需简单地popup一个最近的command对象然后执行它的undo()方法既可。2、Transactional behavior（原子事务行为）借助command模式，可以简单地实现一个具有原子事务的行为。当一个事务失败时，往往需要回退到执行前的状态，可以借助command对象保存这种状态，简单地处理回退操作。3、Progress bars（状态条）假如系统需要按顺序执行一系列的命令操作，如果每个command对象都提供一个getEstimatedDuration()方法，那么系统可以简单地评估执行状态并显示出合适的状态条。4、Wizards（导航）通常一个使用多个wizard页面来共同完成一个简单动作。一个自然的方法是使用一个command对象来封装wizard过程，该command对象在第一个wizard页面显示时被创建，每个wizard页面接收用户输入并设置到该command对象中，当最后一个wizard页面用户按下‘Finish’按钮时，可以简单地触发一个事件调用execute()方法执行整个动作。通过这种方法，command类不包含任何跟用户界面有关的代码，可以分离用户界面与具体的处理逻辑。5、GUI buttons and menu items（GUI按钮与菜单条等等）Swing系统里，用户可以通过工具条按钮，菜单按钮执行命令，可以用command对象来封装命令的执行。6、Thread pools（线程池）通常一个典型的线程池实现类可能有一个名为addTask()的public方法，用来添加一项工作任务到任务队列

中。该任务队列中的所有任务可以用command对象来封装，通常这些command对象会实现一个通用的接口比如Java.lang.Runnable。 7、Macro recording (宏记录) 可以用command对象来封装用户的一个操作，这样系统可以简单通过队列保存一系列的command对象的状态就可以记录用户的连续操作。这样通过执行队列中的command对象，就可以完成 ‘Play back’ 操作了。8、 Networking通过网络发送command命令到其他机器上运行。9、 Parallel Processing (并发处理) 当一个调用共享某个资源并被多个线程并发处理时。”

23.9 命令模式的实现

小A：“命令模式怎样去实现它？”

大B：“命令模式里边一般都有以下几个角色：客户端，请求者，命令接口，命令实现，接受者。下边是简单命令模式的实现代码实现。”

```
public class Client{
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        Command commandOne = new ConcreteCommandOne(receiver);
        Command commandTwo = new ConcreteCommandTwo(receiver);
        Invoker invoker = new Invoker(commandOne,commandTwo);
        invoker.actionOne();
        invoker.actionTwo();
    }
}

public class Invoker
```

```
{
private Command commandOne;
private Command commandTwo;
public Invoker(Command commandOne,Command commandTwo) {
this.commandOne = commandOne;
this.commandTwo = commandTwo;
}
public void actionOne(){
commandOne.execute();
}
public void actionTwo(){
commandTwo.execute();
}
}
public interface Command{
void execute();
}
public class ConcreteCommandOne implements Command{
private Receiver receiver
public ConcreteCommandOne(Receiver receiver) {
this.receiver = receiver;
}
public void execute(){
receiver.actionOne();
}
}
public class ConcreteCommandTwo implements Command{
private Receiver receiver
public ConcreteCommandTwo(Receiver receiver) {
this.receiver = receiver;
}
public void execute(){
```

```
receiver.actionTwo();  
}  
}  
public class Receiver{  
    public Receiver(){  
        //  
    }  
    public void actionOne(){  
        System.out.println("ActionOne has been taken.");  
    }  
    public void actionTwo(){  
        System.out.println("ActionTwo has been taken.");  
    }  
}
```

23.10 命令模式的功能,好处,或者说为什么使用命令模式?

大B：“上边的代码是否看起来很傻呢，本来可以这样简单实现的。”

```
public class Client{  
    public static void main(String[] args) {  
        Receiver receiver = new Receiver();  
        receiver.actionOne();  
        receiver.actionTwo();  
    }  
}
```

```
public class Receiver{  
    public Receiver(){  
        //  
    }  
    public void actionOne(){  
        System.out.println("ActionOne has been taken.");  
    }  
    public void actionTwo(){  
        System.out.println("ActionTwo has been taken.");  
    }  
}
```

大B：“看多简洁，如果是像上边如此简单的需求，这个才应该是我们的选择，但是有些情况下这样的写法不能解决的，或者说解决起来不好，所以引入命令模式。1、我们须要Client和Receiver同时开发,而且在开发过程中分别须要不停重购，改名。2、如果我们要求Redo，Undo等功能。3、我们须要命令不按照调用执行，而是按照执行时的情况排序，执行。4、开发后期,我们发现必须要log哪些方法执行了,如何在尽量少更改代码的情况下实现.并且渐少重复代码。5、在上边的情况下，我们的接受者有很多，不止一个。”

小A：“当我们遇到这些情况时应该怎样去解决？”

大B：“解决办法：情况一、我们可以定义一个接口，让Receiver实现这个接口，Client按照接口调用。情况二、我们可以让Receiver记住一些状态，例如执行前的自己的状态，用来undo，但自己记录自己的状态 实现起来比较混乱，一般都是一个累记录另一个类的状态。情况三、很难实现。情况四、我们须要在每个Action，前后加上log。”

小A：“情况五、相对好实现，但是再加上这个，是否感觉最终的实现很混乱呢？”

大B：“好，我们再来看看命令模式,在命令模式中,我们增加一些过渡的类，这些类就是上边的命名接口和命令实现,这样就很好的解决了情况一、情况二。我们再加入一个Invoker，这样情况三和情况四就比较好解决了。”

如下加入Log和排序后的Invoker

```
public class Invoker{
    private List cmdList = new ArrayList();
    public Invoker(){
    }
    public add(Command command) {
        cmdList.add(command);
    }
    public remove(Command command) {
        cmdList.remove(command);
    }
    public void action(){
        Command cmd;
        while((cmd =getCmd()) != null) {
            log("begin"+cmd.getName());
            cmd.execute();
            log("end"+cmd.getName());
        }
    }
    public Command getCmd(){
        //按照自定义优先级，排序取出cmd
    }
}
```

```

}
public class Client{
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        Command commandOne = new ConcreteCommandOne(receiver);
        Command commandTwo = new ConcreteCommandTwo(receiver);
        Invoker invoker = new Invoker();
        invoker.add(commandOne);
        invoker.add(commandTwo);
        invoker.action();
    }
}

```

23.11 命令模式与其它模式的配合使用

小A：“命令模式怎样与其它模式的配合使用？”

大B：“1、看上边的Invoker的实现是否很像代理模式呢，Invoker的这种实现其实就是一种代理模式。2、需求：有个固定命令组合会多次被执行。解决：加入合成模式，实现方法如下，定义一个宏命令类。

```

public class MacroCommand implements Command{
    private List cmdList = new ArrayList();
    public add(Command command) {
        cmdList.add(command);
    }
    public remove(Command command) {

```

```

cmdList.remove(command);
}
public void execute(){
    Command cmd;
    for(int i=0;i

```

3、需求：须要redo undo解决：加入备忘录模式，一个简单的实现如下

```

public class ConcreteCommandOne implements Command{
    private Receiver receiver
    private Receiver lastReceiver;
    public ConcreteCommandOne(Receiver receiver) {
        this.receiver = receiver;
    }
    public void execute(){
        record();
        receiver.actionOne();
    }
    public void undo(){
        //恢复状态
    }
    public void redo(){
        lastReceiver.actionOne();
        //
    }
    public record(){
        //记录状态
    }
}

```


4、需求：命令很多类似的地方

解决：使用原型模式，利用clone

23.12 足球

大B：“为了把命令模式讲清楚，我举一个印象深刻的例子以便理解，那就借用的足球的例子吧。”

小A：“好的。”

大B：“我设计了五个类，分别是：球队老板，老板的命令(接口)，教练，命令的内容，球员。”

球员的示例代码

```
public class 球员 {  
    public void run() {  
        球场上奔跑;  
    }  
    public void Norun() {  
        球场上不奔跑;  
    }  
    public void shot() {  
        射门;  
    }  
    public void Noshot() {  
        不射门;  
    }  
}
```

```
}  
public void hoo() {  
    积极比赛;  
}  
}
```

教练类的示例代码

```
public class 命令的内容 implements 老板的命令 {  
    球员 team;  
    public 命令的内容 ( 球员 ateam) {  
        this.team = ateam;  
    }  
    //赢球的方法  
    public void victory() {  
        team.hoo();  
        team.run();  
        team.shot();  
    }  
    //输球的方法  
    public void fail() {  
        team.Norun();  
        team.Noshot();  
    }  
}
```

老板的命令类的示例代码

```
public interface 老板的命令 {
```

```
void victory();  
void fail();  
}
```

教练的示例代码

```
public class 教练 {  
    private 老板的命令 bossCommand;  
    public 教练(老板的命令 abossCommand) {  
        this.bossCommand = abossCommand;  
    }  
}
```

老板的示例代码

```
public class 老板 {  
    public static void main(String[] args) {  
        球员 team = new 球员();  
        老板的命令 bossCommand = new 命令的内容(team);  
        教练 drillmaster = new 教练(bossCommand);  
        drillmaster.victory();//赢球  
        drillmaster.fail();//输球  
    }  
}
```

大B：“在实际的应用中老板就相当于用户本人，球员相当于具体的实施类，在具体的实施类里面有n多的方法，你可以通过一个命令类来表明你要的操作，而不是老板类来直接来控制球员类，其中的顺序是这样的：老板发出命令给教练，教练根

据命令中的具体内容给球员，球员作出行为给老板挣钱，这就是老板的命令模式，哈哈，要是老板要打假球就发出drillmaster.fail(); //输球这样的命令就行了。可怜的球迷呀！最后被骂的还是教练和球员，老板要是实在看不下去了，就发出 drillmaster.victory(); //赢球 就可以了，被媒体吹捧的是教练和球员，而老板有了钱，但我们可不知道内幕的原因，因为看起来老板没有参加实际的操作。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第二十四章 苹果汁——解释器模式

24.1 苹果汁

时间：1月9日 地点：大B房间 人物：大B，小A

这天，大小和小A一起讨论该怎么样榨苹果汁。

大B：“休息天我想做点苹果汁给我女朋友喝，但以前每次做出来的汁都有变色的，有什么方法可以做没有变色的吗？”

小A：“最简单的方法：买一台榨汁机，苹果去皮，切成小块，不要核，然后按照操作说明操作。一般是把果肉放到榨汁机中间部位，通电，然后就自动且快速地出来苹果汁。”

大B：“没想到你会做喔！能说一下具体要怎么做吗？”

小A：“首先要准备好原料。也就是苹果100克，凉开水适量。下面我来说说制作方法吧。1、将苹果洗净去皮，然后用刮子或匙慢慢刮成泥状，即可喂食。2、将苹果洗净，去皮，切成黄豆大小的碎丁，加入凉开水适量，上笼蒸20-30分钟，待稍凉后即可喂食。3、刮子或匙一定要洗净消毒。这样做出来的苹果汁是泥香，易消化，美味可口。”

大B：“嘿嘿！不错嘛！下次我就按你的方法试试。”

24.2 解释器模式

大B：“听你这么说来，榨苹果汁就好像是解释器模式。”

小A：“解释器模式？”

大B：“对，定义语言的语法，并且建立一个解释器来解释该语言中的句子。它属于类的行为模式。这里的语言意思是使用规定格式和语法的代码。如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。而且当文法简单、效率不是关键问题的时候效果最好。这也就是解释器模式应用的环境了。”

24.3 解释器模式的组成

大B：“抽象表达式角色：声明一个抽象的解释操作，这个接口为所有具体表达式角色，抽象语法树中的节点，都要实现的。”

小A：“什么叫做抽象语法树呢？”

大B：“抽象语法树的每一个节点都代表一个语句，而在每个节点上都可以执行解释方法。这个解释方法的执行就代表这个语句被解释。由于每一个语句都代表这

个语句被解释。由于每一个语句都代表一个常见的问题的实例，因此每一个节点上的解释操作都代表对一个问题实例的解答。”

小A：“这样啊。”

大B：“解释器模式还包括终结符表达式角色：具体表达式。a、现与文法中的终结符相关联的解释操作b、且句子中的每个终结符需要该类的一个实例与之对应。

3、终结符表达式角色：具体表达式。a、法中的每条规则 $R ::= R_1 R_2 \dots R_n$ 都需要一个非终结符表带式角色b、于从 R_1 到 R_n 的每个符号都维护一个抽象表达式角色的实例变量c、现解释操作，解释一般要递归地调用表示从 R_1 到 R_n 的那些对象的解释操作4、下文（环境）角色：包含解释器之外的一些全局信息。5、户角色：a、建（或者被给定）表示该文法定义的语言中的一个特定的句子的抽象语法树b、用解释操作。”

24.4 解释器模式的优缺点

大B：“释器模式提供了一个简单的方式来执行语法，而且容易修改或者扩展语法。一般系统中很多类使用相似的语法，可以使用一个解释器来代替为每一个规则实现一个解释器。而且在解释器中不同的规则是由不同的类来实现的，这样使得添加一个新的语法规则变得简单。但是解释器模式对于复杂文法难以维护。可以想象一下，每一个规则要对应一个处理类，而且这些类还要递归调用抽象表达式角色，多如乱麻的类交织在一起是多么恐怖的一件事啊！”

小A：“嘿嘿，是啊。”

24.5 解释器模式适用性

大B：“当有一个语言需要解释执行，并且你可将该语言中的句子表示为一个抽象语法树时，可使用解释器模式。”

小A：“那在什么情况下该模式效果最好？”

大B：“当存在以下情况时该模式效果最好：1、该文法简单对于复杂的文法，文法的类层次变得庞大而无法管理。此时语法分析程序生成器这样的工具是更好的选择。它们无需构建抽象语法树即可解释表达式，这样可以节省空间而且还可能节省时间。2、效率不是一个关键问题最高效的解释器通常不是通过直接解释语法分析树实现的，而是首先将它们转换成另一种形式。例如，正则表达式通常被转换成状态机。但即使在这种情况下，转换器仍可用解释器模式实现，该模式仍是有用的。”

24.6 解释器模式参与者

小A：“解释器模式都有哪些参与者？”

大B：“1、AbstractExpression(抽象表达式)—声明一个抽象的解释操作，这个接口为抽象语法树中所有的节点所共享。2、TerminalExpression(终结符表达式)—实现与文法中的终结符相关联的解释操作。—一个句子中的每个终结符需要该类的一个实例。3、NonterminalExpression(非终结符表达式)—对文法中的每一条规则 $R ::= R_1 R_2 \dots R_n$ 都需要一个NonterminalExpression类。—为从 R_1 到 R_n

的每个符号都维护一个AbstractExpression类型的实例变量。—为文法中的非终结符实现解释(Interpret)操作。解释(Interpret)一般要递归地调用表示R1到Rn的那些对象的解释操作。4、Context (上下文) —包含解释器之外的一些全局信息。5、Client (客户) —构建(或被给定)表示该文法定义的语言中一个特定的句子的抽象语法树。该抽象语法树由NonterminalExpression和TerminalExpression的实例装配而成。—调用解释操作。”

24.7 解释器模式效果

小A：“解释器模式有什么优点和缺点？”

大B：“解释器模式有下列的优点和不足：1、易于改变和扩展文法因为该模式使用类来表示文法规则，你可使用继承来改变或扩展该文法。已有的表达式可被增量式地改变，而新的表达式可定义为旧表达式的变体。2、也易于实现文法定义抽象语法树中各个节点的类的实现大体类似。这些类易于直接编写，通常它们也可用一个编译器或语法分析程序生成器自动生成。3、复杂的文法难以维护解释器模式为文法中的每一条规则至少定义了一个类(使用BNF定义的文法规则需要更多的类)。因此包含许多规则的文法可能难以管理和维护。可应用其他的设计模式来缓解这一问题。但当文法非常复杂时，其他的技术如语法分析程序或编译器生成器更为合适。4、增加了新的解释表达式的方式解释器模式使得实现新表达式‘计算’变得容易。例如：你可以在表达式类上定义一个新的操作以支持优美打印或表达式的类型检查。如果你经常创建新的解释表达式的方式，那么可以考虑使用Visitor模式以避免修改这些代表文法的类。”

24.8 加减乘除

大B：“来举一个加减乘除的例子吧，实现思路来自于《Java与模式》中的例子。每个角色的功能按照上面提到的规范来实现。”

//上下文（环境）角色，使用HashMap来存储变量对应的数值

```
class Context
{
    private Map valueMap = new HashMap();
    public void addValue(Variable x , int y)
    {
        Integer yi = new Integer(y);
        valueMap.put(x , yi);
    }
    public int LookupValue(Variable x)
    {
        int i = ((Integer)valueMap.get(x)).intValue();
        return i ;
    }
}
```

//抽象表达式角色，也可以用接口来实现

```
abstract class EXpression
{
    public abstract int interpret(Context con);
}
```

//终结符表达式角色

```
class Constant extends Expression
{
    private int i ;
```

```

public Constant(int i)
{
    this.i = i;
}

public int interpret(Context con)
{
    return i ;
}

class Variable extends Expression
{
    public int interpret(Context con)
    {
        //this为调用interpret方法的Variable对象
        return con.LookupValue(this);
    }
}

//非终结符表达式角色
class Add extends Expression
{
    private Expression left ,right ;
    public Add(Expression left , Expression right)
    {
        this.left = left ;
        this.right= right ;
    }
    public int interpret(Context con)
    {
        return left.interpret(con) + right.interpret(con);
    }
}

class SuBTract extends Expression

```

```

{
private Expression left , right ;
public Subtract(Expression left , Expression right)
{
this.left = left ;
this.right= right ;
}
public int interpret(Context con)
{
return left.interpret(con) - right.interpret(con);
}
}

class Multiply extends Expression
{
private Expression left , right ;
public Multiply(Expression left , Expression right)
{
this.left = left ;
this.right= right ;
}
public int interpret(Context con)
{
return left.interpret(con) * right.interpret(con);
}
}

class Division extends Expression
{
private Expression left , right ;
public Division(Expression left , Expression right)
{
this.left = left ;
this.right= right ;
}
}

```

```

}
public int interpret(Context con)
{
    try{
        return left.interpret(con) / right.interpret(con);
    }catch(ArithmeticException ae)
    {
        System.out.println("被除数为0!");
        return -11111;
    }
}
}
}
//测试程序, 计算 (a*b)/(a-b+2)
public class Test
{
    private static Expression ex ;
    private static Context con ;
    public static void main(String[] args)
    {
        con = new Context();
        //设置变量、常量
        Variable a = new Variable();
        Variable b = new Variable();
        Constant c = new Constant(2);
        //为变量赋值
        con.addValue(a , 5);
        con.addValue(b , 7);
        //运算, 对句子的结构由我们自己来分析, 构造
        ex = new Division(new Multiply(a , b), new Add(new Subtract(a , b) , c));
        System.out.println("运算结果为 : "+ex.interpret(con));
    }
}

```

大B：“解释器模式并没有说明如何创建一个抽象语法树，因此它的实现可以多种多样，在上面我们是直接在Test中提供的，当然还有更好、更专业的实现方式。对于终结符，GOF建议采用享元模式来共享它们的拷贝，因为它们要多次重复出现。但是考虑到享元模式的使用局限性，我建议还是当你的系统中终结符重复的足够多的时候再考虑享元模式。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第六部分

扩展型模式

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第二十五章 多功能的手机——扩展型模式

25.1 多功能的手机

时间：1月10日 地点：大B房间 人物：大B，小A

生活水平的逐渐提高，人们对手机的需求不仅仅局限于通讯，年轻人买手机追求的不仅是外表的与众不同，还有就是功能的时尚，都要显示出年轻人独有的特点。

大B：“具有 Java扩展功能的手机其手机功能和内容要比其他那些不支持Java 的手机要丰富一些！”

小A：“是啊！支持 Java 的手机可以通过下载一些 Java 软件来实现一些手机本身原本没有的功能！”

大B：“嗯！就像SIEMENS 2128 本身并没有 直接收发 E-mail 的功能，但是只要你下载一个 收发 E-mail 的 Java 工具软件。以后只要你在手机上直接进入该 Java 工具 输入帐号跟密码就可以收发E-mail！ ”

小A：“还有个 Java QQ。”

大B：“对！手机上使用的 Java QQ 跟普通的基于短信的‘移动QQ’是不一样的。普通的基于短信的‘移动QQ’是以短信的形式来实现电脑QQ跟手机之间的信息收发。每条一毛。这时手机上看不到对方的头像的也不知道哪几个人在线，虽然可以通过查询来知道，但也只是即时的情况如想知道变化则需不停的查。费力，费钱，费时。电脑上收发的来自手机‘移动QQ’的消息跟收发来自其他电脑QQ的消息是不同的。大家都知道，收发普通QQ的消息是通过电脑的QQ上的那个小头像来完成的，而收发‘移动QQ’则是通过QQ头像旁边的那个小手机来进行！Java QQ则跟普通电脑上网用QQ类似，通过头像收发消息。以至于对方根本分辨不出你是用的电脑上网还是手机上网。而且你在手机上使用 Java QQ 时还可以看到手机上还显示对方的头像图标，如果通过 GPRS 上 Java QQ 还可以实现 24 小时不间断在线。也就是说你QQ上的头像24小时都是彩色的，显示你在线。而且只有在你收发消息产生数据流量的时候才会记费花钱，不产生流量则无须花钱。”

大B：“再有，即便花钱也相当便宜，每 1KB 的流量才 3分钱 左右，也就是说你 收发消息时每条才 3分钱 左右，跟基于短信的‘移动QQ’相比，是不是便宜多了？另外，现在还有很多 Java 编写的游戏，一旦你已经对手机上已有的游戏玩腻了。随时还可以下载一个 Java 游戏来玩玩儿！很多以前 8 位机上的游戏现在都有 可以下载到手机的 Java 版本，比如‘泡泡龙’，呵呵~~ 总之，Java 的功能 N 多，另外还有电子书，铃声编辑器，电话本管理器，电子地图……”

小A：“即便有 Java ，在某种情况下，要实现某些功能还需要其他方面比如硬件，网络的支持的。”

大B：“是啊！像前面的 Java QQ 一般就需要 GPRS 的网络支持才行，还好，一般支持 Java 的手机都支持 GPRS！其他还要强调的是，有些手机，像 NOKIA

6610/7210等。对 Java 程序下载有个大小要求（64K），这会导致一些稍大的 Java 程序不能下载用不了。”

小A：“Java QQ 好象就有 69K 左右！不过 SIEMENS 好象到没有这个限制——只要你剩余的可支配的动态足够的话！”

25.2 扩展型模式

曾几何时，手机已经不仅仅是人们通信交流的简单方式了，从它的悄然诞生，到走进千家万户，真可谓是忽如一夜春风来，千树万树梨花开。

大B：“从我们刚才聊了功能手机中，可以知道什么是扩展型模式。”

小A：“扩展型模式？”

大B：“扩展模式是指向模式添加元素，通常是对象类和属性。缺省的模式中带有可用于各个目录条目的大量对象类和属性。扩展模式之前，请先查看缺省模式中是否有可以使用而不需扩展模式的现有元素。

25.3 扩展模式指南

大B：“不论使用何种方法扩展模式，都请遵循下列指南：1、查看是否可以使用缺省模式中已定义的对象类、属性或语法，而不是添加新的。2、不要定义多个属性来存储同种信息。相反，应该仅添加一个属性，然后在多个结构对象类使用的辅

助对象类中定义该属性。3、不要编辑现有的模式元素。例如，不要从现有的对象类中删除属性或将属性添加到其上。可以删除不再需要的定制对象类，前提是确信它没有被使用。4、尽可能用将属性定义为‘可选’而不是‘强制’的方法创建对象类，以使模式更灵活。5、扩展模式后，配置对新模式元素的访问权限。”

25.4 扩展现有对象类

小A：“如何将属性添加到缺省模式中的对象类？”

大B：“取决于该属性是否还要应用到另一个对象类。如果该属性仅应用于一个对象类，请将其添加到新的结构对象类中，并让该新的对象类继承需要扩展的对象类的属性。例如，要扩展缺省模式中的对象类A，请将属性添加到新结构对象类 B，然后定义对象类 B 继承 A。”

大B：“如果属性要应用于多个结构对象类，请将其添加到一个新的辅助对象类中，然后将该辅助对象类添加到要使用该属性的每个结构对象类中。”

小A：“假定需要将相同属性添加到缺省模式中的对象类 A 和 B 呢？”

大B：“请将该属性添加到一个新的辅助对象类 C 中，然后将 C 添加到 A 和 B 中。”

大B：“注意添加新的目录条目类型，通常应该创建继承顶级对象类的新结构对象类。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质

电子书下载！！！！

第二十六章 三明治——装饰器模式

26.1 三明治

时间：1月11日 地点：大B房间 人物：大B，小A

小A：“很多人都吃过三明治，都会知道三明治必不可少的是两块面包片，然后可以在夹层里加上蔬菜、沙拉、咸肉等等，外面可以涂上奶油之类的。”

大B：“假如现在你要为一个三明治小店构造一个程序，其中要设计各种三明治的对象。可能你已经创建了一个简单的Sandwich对象，现在要产生带蔬菜的就是继承原有的Sandwich添加一个蔬菜的成员变量，看起来很‘正点’的做法，以后我还要带咸肉的、带奶油的、带蔬菜的又分为带青菜的、带芹菜的、生菜的.....还是一个一个继承是吧！假如我们还需要即带蔬菜又带其它肉类，设置我们还要求这些添加成分的任意组合，那你就慢慢继承吧！”

小A：“呵呵。”

大B：“读过几年书的会下面这个算术，我们有n种成分，在做三明治的时候任意搭配，那么有多少种方案呢？！算算吧！你会有惊人的发现。N种成分，什么都不要是 C_n^0 种方案吧！要1种是 C_n^1 吧！.....要n种是 C_n^n 吧！”

小A：“加起来不就是吗？”

大B：“牛顿莱布尼兹公式记得吧！总共2的n次方案。有可能前面10天写了K个类，老板让你再加一种成分你就得再干10天，下一次再加一种你可得干20天哦！同时你可以发现你的类库急剧地膨胀！老板可能会说你：xxx前K天你加了n个成分，怎么现在这么不上进呢？后K天只加了1个成分啊？！！可能你会拿个比给老板算算，老板那么忙会睬你吗？！有可能你的老板会说：不管怎么样我就要你加，K天你还给我加n个成分！！”

小A：“呵呵，怎么办啊！跳槽啊！”

大B：“跳槽了也没人要你！！人家一看就知道你没学设计模式。”

26.2 装饰器模式

通过例子告诉大家一点：任何设计不是一成不变的、模式的应用是极其灵活的.....

大B：“装饰模式：Decorator常被翻译成‘装饰’，我觉得翻译成‘油漆工’更形象点，油漆工(decorator)是用来刷油漆的，那么被刷油漆的对象我们称decoratee。这两种实体在Decorator模式中是必须的。”

小A：“那我们应该如何去定义它？”

大B：“动态给一个对象添加一些额外的职责，就象在墙上刷油漆。使用Decorator模式相比用生成子类方式达到功能的扩充显得更为灵活。”

下面是以上各个类的意义：

1、Ingredient（成分）：所有类的父类，包括它们共有的方法，一般为抽象类且方法都有默认的实现，也可以为接口。它有Bread和Decorator两个子类。这种实际不存在的，系统需要的抽象类仅仅表示一个概念，

2、Bread（面包）：就是我们三明治中必须的两片面包。它是系统中最基本的元素，也是被装饰的元素，和IO中的媒质流（原始流）一个意义。在装饰器模式中属于一类角色，所以其颜色为紫色。

3、Decorator（装饰器）：所有其它成分的父类，这些成分可以是猪肉、羊肉、青菜、芹菜。这也是一个实际不存在的类，仅仅表示一个概念，即具有装饰功能的所有对象的父类。

4、Pork（猪肉）：具体的一个成分，不过它作为装饰成分和面包搭配。

5、Mutton（羊肉）：同上。

6、Celery（芹菜）：同上。

7、Greengrocery（青菜）：同上。

大B：“我们现在来总结一下装饰器模式中的四种角色：1、被装饰对象（Bread）；2、装饰对象（四种）；3、装饰器（Decorator）；4、公共接口或抽象类（Ingredient）。其中1和2是系统或者实际存在的，3和4是实现装饰功能需要的抽象类。写段代码体会其威力！程序很简单，但是实现的方法中可以假如如何你需要的方法，意境慢慢体会吧！”

```
//Ingredient.java

public abstract class Ingredient {
    public abstract String getDescription();
    public abstract double getCost();
    public void printDescription(){
        System.out.println("Name"+ this.getDescription());
        System.out.println("Price RMB"+ this.getCost());
    }
}
```

大B：“所有成分的父亲类，抽象类有一个描述自己的方法和一个得到价格的方法，以及一个打印自身描述和价格的方法。”

小A：“这个方法不就是与刚才的那两个方法构成模板方法吗？”

```
//Bread.java

public class Bread extends Ingredient {
    private String description ;
    public Bread(String desc){
        this.description=desc ;
    }
    public String getDescription(){
        return description ;
    }
    public double getCost(){
        return 2.48 ;
    }
}
```

大B：“面包类，因为它是一个具体的成分，因此实现父类的所有的抽象方法。

描述可以通过构造器传入，也可以通过set方法传入。同样价格也是一样的，我就很简单地返回了。”

```
//Decorator.java

public abstract class Decorator extends Ingredient {
    Ingredient ingredient ;
    public Decorator(Ingredient igd){
        this.ingredient = igd;
    }
    public abstract String getDescription();
    public abstract double getCost();
}
```

大B：“装饰器对象，所有具体装饰器对象父类。它最经典的特征就是：1、必须有一个它自己的父类为自己的成员变量；2、必须继承公共父类。这是因为装饰器也是一种成分，只不过是那些具体具有装饰功能的成分的公共抽象罢了。在我们的例子中就是有一个Ingredient作为其成员变量。Decorator继承了Ingredient类。”

```
//Pork.java

public class Pork extends Decorator{
    public Pork(Ingredient igd){
        super(igd);
    }
    public String getDescription(){
        String base = ingredient.getDescription();
        return base + "\n" + "Decroated with Pork !";
    }
}
```

```

public double getCost(){
    double basePrice = ingredient.getCost();
    double porkPrice = 1.8;
    return basePrice + porkPrice ;
}
}

```

大B：“具体的猪肉成分，同时也是一个具体的装饰器，因此它继承了Decorator类。猪肉装饰器装饰可以所有的其他对象，因此通过构造器传入一个Ingredient的实例，程序中调用了父类的构造方法，主要父类实现了这样的逻辑关系。同样因为方法是具体的成分，所以getDescription得到了实现，不过由于它是具有装饰功能的成分，因此它的描述包含了被装饰成分的描述和自身的描述。价格也是一样的。价格放回的格式被装饰成分与猪肉成分的种价格哦！”

大B：“从刚才的两个方法中我们可以看出，猪肉装饰器的功能得到了增强，它不仅仅有自己的描述和价格，还包含被装饰成分的描述和价格。主要是因为被装饰成分是它的成员变量，因此可以任意调用它们的方法，同时可以增加自己的额外的共同，这样就增强了原来成分的功能。”

```

//Mutton.java
public class Mutton extends Decorator{
    public Mutton(Ingredient igd){
        super(igd);
    }
    public String getDescription(){
        String base = ingredient.getDescription();
        return base + "\n" + "Decroated with Mutton !";
    }
}

```

```

public double getCost(){
double basePrice = ingredient.getCost();
double muttonPrice = 2.3;
return      basePrice + muttonPrice ;
}
}

```

羊肉的包装器。

```

//Celery.java
public class Celery extends Decorator{
public Celery(Ingredient igd){
super(igd);
}
public String getDescription(){
String base = ingredient.getDescription();
return base + "\n" + "Decorated with Celery !";
}
public double getCost(){
double basePrice = ingredient.getCost();
double celeryPrice = 0.6;
return      basePrice + celeryPrice ;
}
}

```

芹菜的包装器。

```

//GreenGrocery.java
public class GreenGrocery extends Decorator{

```

```

public GreenGrocery (Ingredient igd){
    super(igd);
}
public String getDescription(){
    String base = ingredient.getDescription();
    return base + "\n" + "Decorated with GreenGrocery  !";
}
public double getCost(){
    double basePrice = ingredient.getCost();
    double greenGroceryPrice = 0.4;
    return      basePrice + greenGroceryPrice ;
}
}

```

青菜的包装器。

大B：“我们来领略装饰器模式的神奇吧！我们有一个测试类，其中建立夹羊肉的三明治、全蔬菜的三明治、全荤的三明治。”

小A：“好像真的很香哦！”

```

public class DecoratorTest{
    public static void main(String[] args){
        Ingredient compound = new Mutton(new Celery(new Bread("Master24's Bread")));
        compound.printDescription();
        compound = new Celery(new GreenGrocery(new Bread("Bread with milk")));
        compound.printDescription();
        compound = new Mutton(new Pork(new Bread("Bread with cheese")));
        compound.printDescription();
    }
}

```

```
}
```

大B：“这就是一个简单的装饰器类！假如你对想中国式的吃法，可以将加入馒头、春卷皮、蛋皮.....夹菜可以为肉丝.....突然想到了京酱肉丝。”

26.3 装饰器模式的结构

大B：“在谈及软件中的结构，一般会用UML图表示。”

小A：“喔。”

大B：“UML和ANT、JUnit等都是软件设计中基本的工具，会了没有啊！”

小A：“是吗？”

大B：“我和你具体说说。1、Component就是装饰器模式中公共方法的类，在装饰器模式结构图的顶层。2、ConcreteComponent是转换器模式中具体的被装饰的类，IO包中的媒体流就是此种对象。3、Decorator装饰器模式中的核心对象，所有具体装饰器对象的父类，完成装饰器的部分职能。刚才的例子中Decorator类和这里的对应。该类可以只做一些简单的包裹被装饰的对象，也可以还包含对Component中方法的实现.....他有一个鲜明的特点：继承至Component，同时包含一个Component作为其成员变量。装饰器模式动机中的动态地增加功能是在这里实现的。4、ConcreteDecoratorA和ConcreteDecoratorB是两个具体的装饰器对象，他们完成具体的装饰功能。装饰功能的实现是通过调用被装饰对象对应的方法，加上装饰对象自身的方法。这是装饰器模式动机中的添加额外功能的关键。你可能还

会发现：ConcreteDecoratorA和ConcreteDecoratorB的方法不一样，这就是一般设计模式中谈及装饰器模式的‘透明装饰器’和‘不透明装饰器’。‘透明装饰器’就是整个Decorator的结构中所有的类都保持同样的‘接口’，这里是共同方法的意思，这是一种极其理想的状况，就像餐饮的例子一样。现实中绝大多数装饰器都是‘不透明装饰器’，他们的‘接口’在某些子类中得到增强，主要看这个类与顶层的抽象类或者接口是否有同样的公共方法。IO中的ByteArrayInputStream就比InputStream抽象类多一些方法，因此IO中的装饰器是一个‘不透明装饰器’。”

小A：“喔。”

大B：“从IO中输入字节流部分的装饰器我们可以知道：1、InputStream是装饰器的顶层类，一个抽象类！包括一些共有的方法，如：1、读方法——read（3个）；2、关闭流的方法——close；3、mark相关的方法——mark、reset和markSupport；4、跳跃方法——skip；5、查询是否还有元素方法——available。2、FileInputStream、PipedInputStream...五个紫色的，是具体的被装饰对象。从他们的‘接口’中可以看出他们一般都有额外的方法。3、FilterInputStream是装饰器中的核心，Decorator对象。4、DataInputStream、BufferedInputStream...四个是具体的装饰器，他们保持了和InputStream同样的接口。5、ObjectInputStream是IO字节输入流中特殊的装饰器，他不是FilterInputStream的子类，不知道Sun处于何种意图不作为FileterInputStream的子类，其中流中也有不少的例子。他和其他FilterInputStream的子类功能相似都可以装饰其他对象。IO包中不仅输入字节流是采用装饰器模式、输出字节流、输入字符流和输出字符流都是采用装饰器模式。”

26.4 装饰器模式的特征

大B：“装饰器角色持有一个构件对象的实例，并实现了抽象构件的接口。每一个接口的实现都是委派给所持有的构件对象，并增加新的功能。”

26.5 装饰器模式优点和缺点

小A：“装饰器模式有什么优点？”

大B：“装饰器与继承的目的都是扩展对象的功能，但装饰器提供了比继承更大的灵活性，可以动态的决定是‘粘上’还是‘去掉’一个装饰。通过使用不同的具体装饰类和这些类的排列组合，可以创建出很多不同行为的组合。”

小A：“那装饰器模式有什么缺点哩？”

大B：“装饰器比继承关系使用更少的类，但比继承关系使用更多的对象，更多的对象会使查错变得更困难，特别是这些对象看上去很像的时候。”

26.6 模式的简化

大B：“简化必须注意两点：a、一个装饰器类的接口必须与被装饰的类的接口相容。b、尽量保持Component作为一个‘轻’类。Component类的职责在于为各个

具体装饰器类提供共同的接口，而不是存储数据，所以不要把太多的逻辑和状态放在Component类里面。省略Component接口，只有一个具体的ConcreteComponent类，则Decorater经常作为ConcreteComponent的子类。”

如图26-1所示：

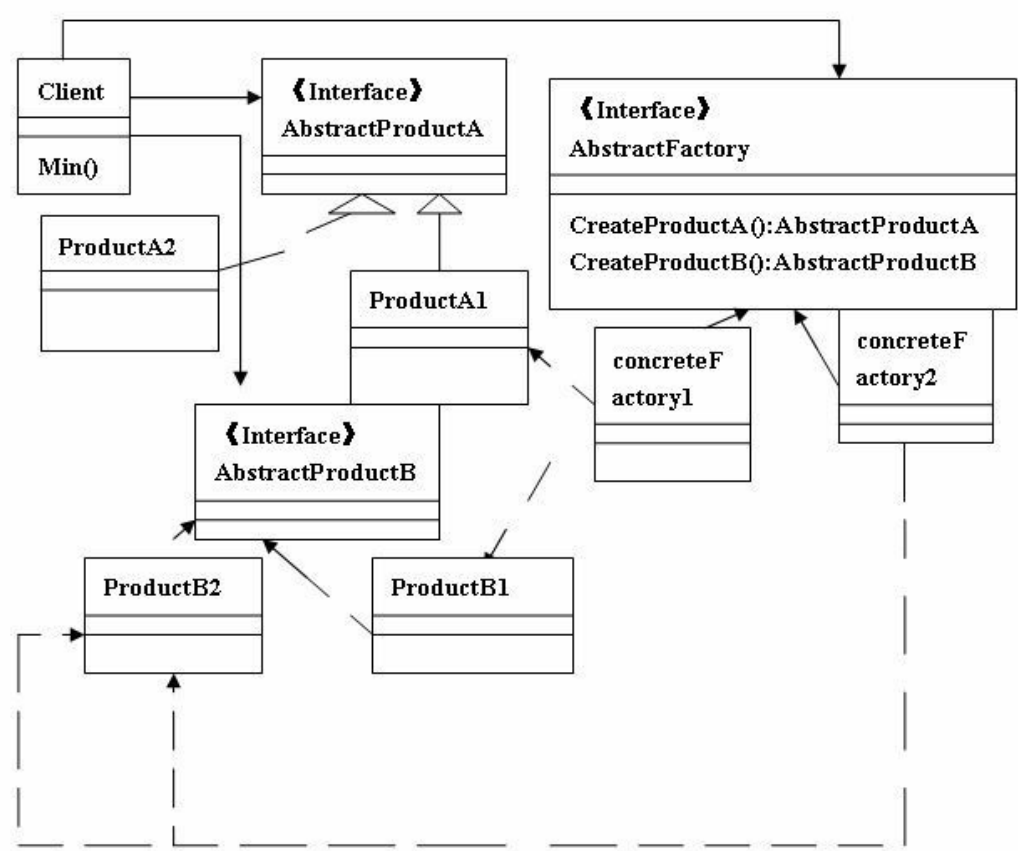


图26-1

省略Decorator类，如果只有一个具体的ConcreteDecorator类，那可以省略Decorator，将ConcreteDecorator和Decorator角色的职责合并在一起。如图26-2所示：

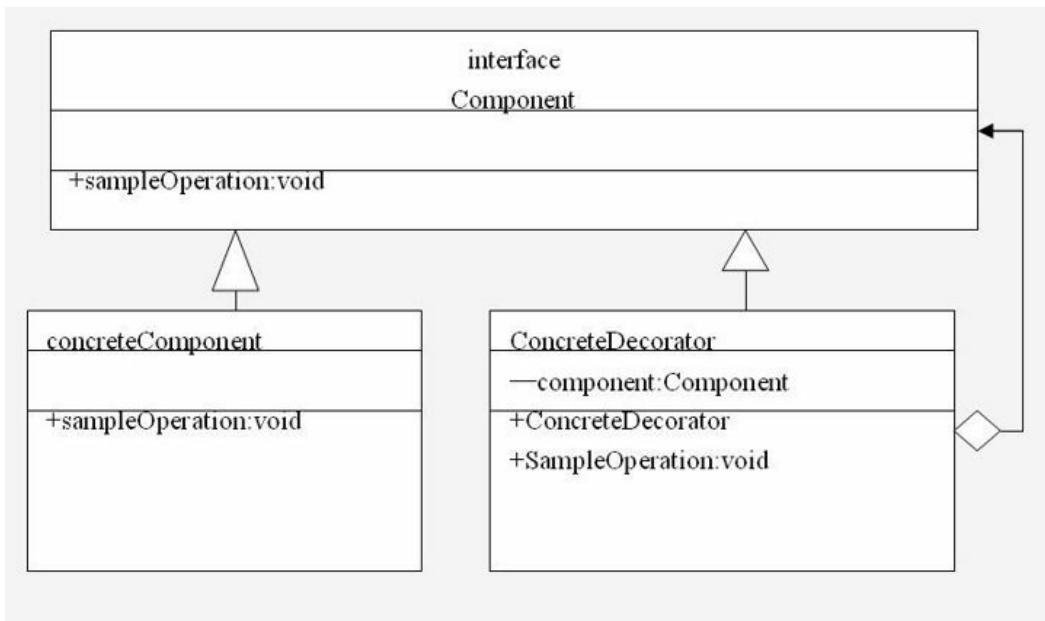


图26-2

// Component.java 构件类

```

public abstract class Component
{
    public abstract void doSomething();
}
  
```

// ConcreteComponent.java 具体构件类

```

public class ConcreteComponent extends Component
{
    public void doSomething()
    {
        // provide implementation here
    }
}
  
```

// Decorator.java 装饰器抽象类

```

public abstract class Decorator extends Component
{
    protected Component component;
    public Decorator(Component component)
    {
  
```

```
this.component = component;
}
public void doSomething()
{
    component.doSomething();
}
}
// ConcreteDecorator.java 具体装饰器类
public class ConcreteDecorator extends Decorator
{
    public ConcreteDecorator(Component component)
    {
        super(component);
    }
    private void addedBehavior()
    {
        // some extra functionality goes here
    }
    public void doSomething()
    {
        component.doSomething();
        addedBehavior();
    }
}
```

26.7 为什么使用Decorator?

小A：“为什么使用Decorator?”

大B：“我们通常可以使用继承来实现功能的拓展，如果这些需要拓展的功能的种类很繁多，那么势必生成很多子类，增加系统的复杂性，同时，使用继承实现功能拓展，我们必须可预见这些拓展功能，这些功能是编译时就确定了，是静态的。使用Decorator的理由是：这些功能需要由用户动态决定加入的方式和时机。Decorator提供了‘即插即用’的方法，在运行期间决定何时增加何种功能。”

26.8 如何使用装饰器模式？

小A：“那我们应该如何使用它哩？”

大B：“举Adapter中的打桩示例，在Adapter中有两种类：方形桩、圆形桩，Adapter模式展示如何综合使用这两个类，在Decorator模式中，我们是要在打桩时增加一些额外功能，比如，挖坑。在桩上钉木板等，不关心如何使用两个不相关的类。”

我们先建立一个接口：

```
public interface Work
{
    public void insert();
}
```

接口Work有一个具体实现：插入方形桩或圆形桩，这两个区别对Decorator是无所谓。我们以插入方形桩为例：

```
public class SquarePeg implements Work{
    public void insert(){
        System.out.println("方形桩插入");
    }
}
```

大B：“现在有一个应用：需要在桩打入前，挖坑，在打入后，在桩上钉木板，这些额外的功能是动态，可能随意增加调整修改，比如，可能又需要在打桩之后钉架子，只是比喻。那么我们使用Decorator模式，这里方形桩SquarePeg是decoratee被刷油漆者，我们需要在decoratee上刷些‘油漆’，这些油漆就是那些额外的功能。”

```
public class Decorator implements Work{
    private Work work;
    //额外增加的功能被打包在这个List中
    private ArrayList others = new ArrayList();
    //在构造器中使用组合new方式,引入Work对象;
    public Decorator(Work work)
    {
        this.work=work;
        others.add("挖坑");
        others.add("钉木板");
    }
    public void insert(){
        newMethod();
    }
    //在新方法中,我们在insert之前增加其他方法,这里次序先后是用户灵活指定的
    public void newMethod()
```

```

{
otherMethod();
work.insert();
}

public void otherMethod()
{
ListIterator listIterator = others.listIterator();
while (listIterator.hasNext())
{
System.out.println(((String)(listIterator.next())) + " 正在进行");
}
}
}

```

大B：“在刚才讲的例子中，我们把挖坑和钉木板都排在了打桩insert前面，这里只是举例说明额外功能次序可以任意安排。好了，Decorator模式出来了，我们看如何调用。”

```

Work squarePeg = new SquarePeg();
Work decorator = new Decorator(squarePeg);
decorator.insert();

```

Decorator模式至此完成.

大B：“如果你细心，会发现，上面调用类似我们读取文件时的调用。”

```

FileReader fr = new FileReader(filename);
BufferedReader br = new BufferedReader(fr);

```

大B：“实际上Java 的I/O API就是使用Decorator实现的，I/O变种很多，如果都采取继承方法，将会产生很多子类，显然相当繁琐。Jive中的Decorator实现，在论坛系统中，有些特别的字是不能出现在论坛中如‘打倒XXX’，我们需要过滤这些‘反动’的字体。不让他们出现或者高亮度显示。在IBM Java专栏中专门谈Jive的文章中，有谈及Jive中ForumMessageFilter.java使用了Decorator模式，其实，该程序并没有真正使用Decorator。我们在分辨是否真正是Decorator模式，以及会真正使用Decorator模式，一定要把握好Decorator模式的定义，以及其中参与的角色(Decoratee 和Decorator)。”

大B：“若你从事过面向对象的php开发，即使很短的时间或者仅仅通过本书了解了一些，你会知道，你可以通过继承改变或者增加一个类的功能，这是所有面向对象语言的一个基本特性。如果已经存在的一个php类缺少某些方法，或者须要给方法添加更多的功能（魅力），你也许会仅仅继承这个类来产生一个新类——这建立在额外的代码上。但是产生子类并不总是可能或是合适的。”

小A：“如果改变一个已经初始化的对象的行为，怎么办？或者，继承许多类的行为，又怎么办？”

大B：“前一个，只能在于运行时完成，后者显然时可能的，但是可能会导致产生大量的不同的类——可怕的事情。”

小A：“你如何组织你的代码使其可以容易的添加基本的或者一些很少用到的特性，而不是直接不额外的代码写在你的类的内部？”

大B：“装饰器模式提供了改变子类的灵活方案。装饰器模式允许你在不引起子类数量爆炸的情况下动态的修饰对象，添加特性。当用于一组子类时，装饰器模式

更加有用。”

小A：“如果你拥有一族子类（从一个父类派生而来），你需要在与子类独立使用情况下添加额外的特性，你可以使用装饰器模式，以避免代码重复和具体子类数量的增加。”

大B：“看看以下例子，你可以更好的理解这种观点。考虑一个建立在组件概念上的‘form’表单库，在那里你需要为每一个你想要表现的表单控制类型建立一个类。这种类图可以如图26-3所示：Select and TextInput类是组件类的子类。假如你想要增加一个‘labeled’带标签的组件——一个输入表单告诉你要输入的内容。因为任何一个表单都可能需要被标记，你可能会象这样继承每一个具体的组件。”

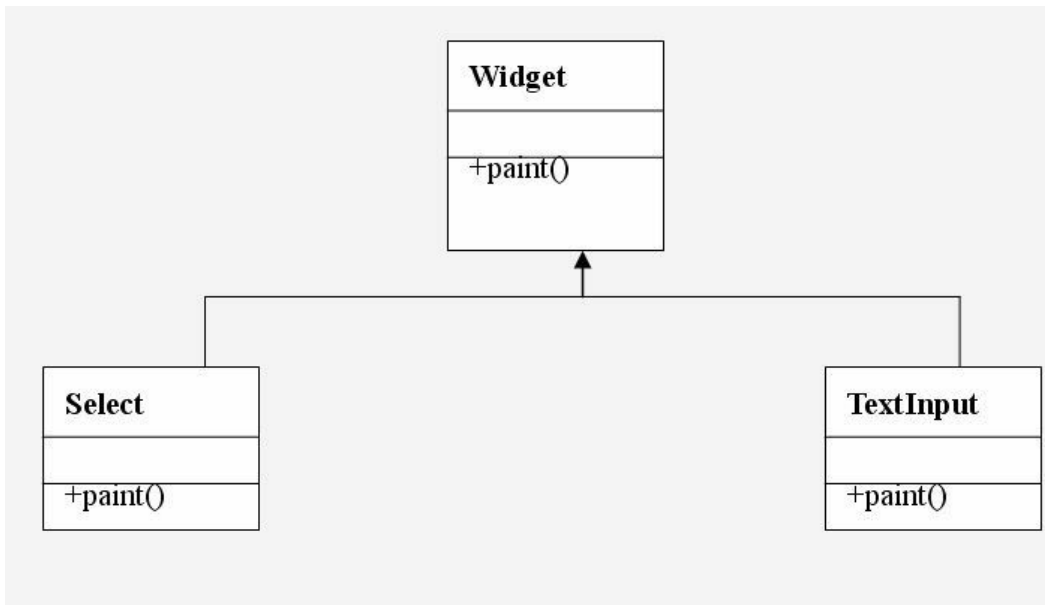


图26-3 “form” 表单库

大B：“图26-3看起来并不怎么坏，现在让我们再增加一些特性。表单验证阶段，你希望能够指出一个表单控制是否合法。你为非法控制使用的代码又一次继承其它组件，因此又需要产生大量的子类。”

如图26-4所示：

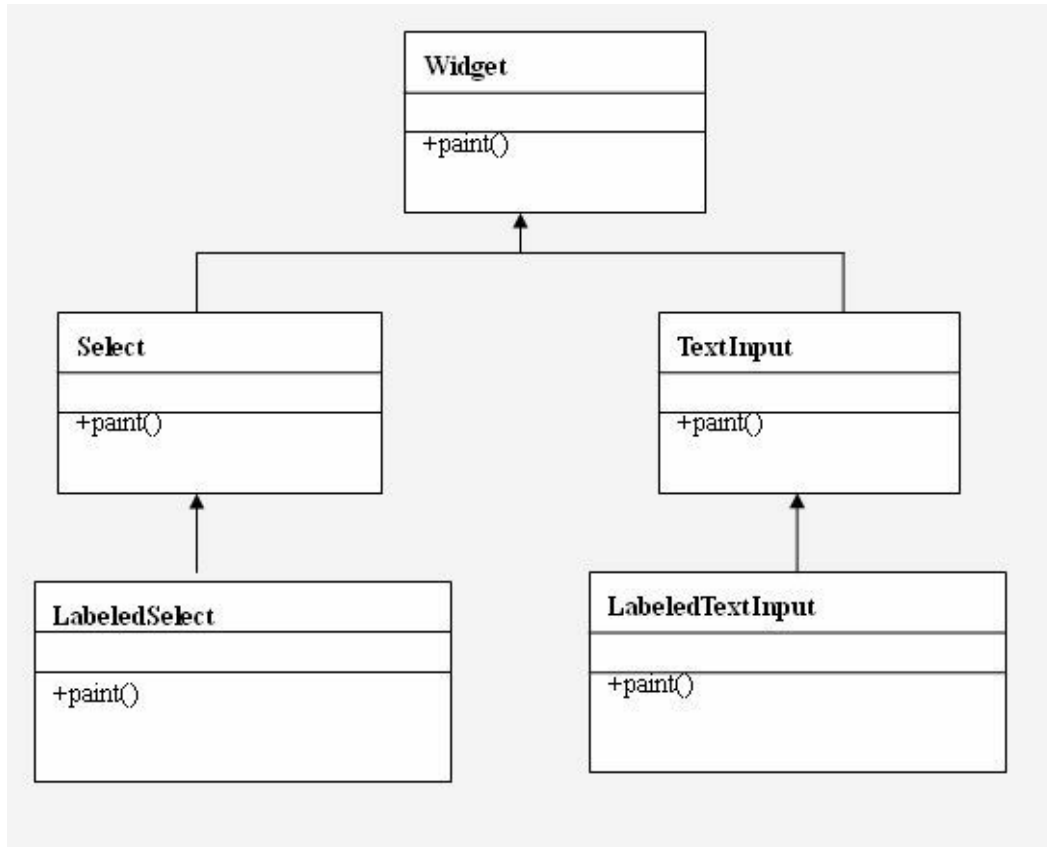


图26-4 表单验证

大B：“这个类看起来并不是太坏，所以让我们增加一些新的功能。在结构有效性确认中你需要指出结构是否是有效的。你需要让你检验有效性的代码也可以应用到其它部件，这样不用再更多的子类上进行有效性验证。”

如图26-5所示

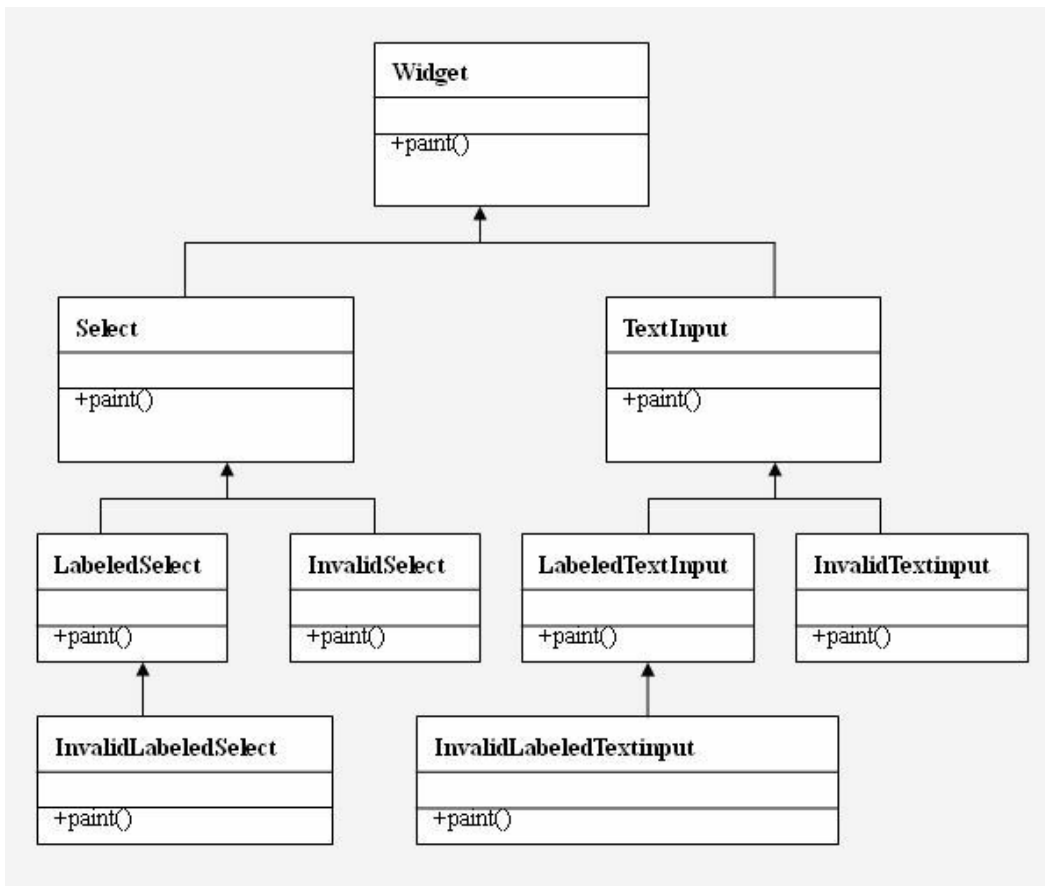


图26-5 表单有效性验证

大B：“这里子类溢出并不是唯一的问题。想一想那些重复的代码，你需要重新设计你的整个类层次。有没有更好的方法！”

小A：“确实，装饰器模式是避免这种情况的好方法。”

大B：“装饰器模式结构上类似与代理模式。一个装饰器对象保留有对对象的引用，而且忠实的重新建立被装饰对象的公共接口。装饰器也可以增加方法，扩展被装饰对象的接口，任意重载方法，甚至可以在脚本执行期间有条件的重载方法。”

小A：“那是为什么？”

大B：“为了探究装饰器模式，让我们以前面讨论过的表单组件库为例，并且用装饰器模式而不是继承，实现 ‘lable’ 和 ‘invalidation’ 两个特性。”

26.9 装饰器模式与适配器模式的区别

小A：“装饰器模式与适配器模式有什么区别？”

大B：“装饰器模式与适配器模式都叫做包装模式（Warpper），但装饰器与被装饰具有相同的接口（具体表现为都实现想同的Java Interface或装饰器是被装饰类的子类等）。但适配器与被适配的类具有不同的接口（虽然可能用部分重合的API）如BufferedReader是一个Decorator因为它接受一个Reader对象，但是InputStreamReader它接受一个InputStream对象，把InputStream的API转换成Reader的API。半装饰器（退化了的装饰器）如果一个Decorator除了提供被装饰类的接口外还提供了另外的方法，就变成了一个半透明的装饰器，客户如果要使用这个特殊的方法说要使用具体的装饰器类，这样就违背了装饰器模式的使用初衷，但实际应用往往无法避免。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第二十七章 老公，有钱不？——迭代器模式

27.1 老公，有钱不？

时间：1月12日 地点：大B房间 人物：大B，小A

大B：“现在的社会都是男人挣钱女人花，比如说有个人，他有很多钱，钱对来说就是一个聚合对象，他老婆是个会花钱的主。”

小A：“有钱嘛！怕什么？”

大B：“是啊。她老婆不断地从她老公那取钱出来花，每次要钱，总是先问：老公，有钱不？”

小A：“呵呵！那她老公怎么说？”

大B：“她老公就会说：有！”

小A：“那不错嘛！”

大B：“是啊。接着她老婆就会说‘拿N块钱出来我要买新衣服！’”

小A：“嘿嘿！那她老公会给她钱吗？”

大B：“会，她老公说：‘好，给！’”

小A：“嘿嘿！那还真不错喔！”

大B：“但是渐渐的，他老公觉得嫌烦，整天忙着赚钱(一个责任)，累得要死，现在连要钱也得烦他(另一个责任)！”

小A：“是喔！这样久了是挺烦人的。”

大B：“他老婆也觉得累，因为每次她老公的钱都不是放在同一地方，一会放钱包里，一会放上衣口袋，一会放裤子口袋，都得自己去翻，很麻烦的！”

小A：“那后来怎么办呢？”

大B：“遇到这个问题，她老公想到一个好方法，他把钱放进银行，办了一张银行卡，这样每次他老婆要钱就把银行卡给他，不要再回答她那么多问题了，她老婆也方便，不用操心钱放口袋还是钱包，只要到银行的自动取钱机取就OK了！”

小A：“嗯，这样就方便多了。”

27.2 迭代器模式

这个模式就是一个迭代器模式的生活例子！对于赚钱的老公，他就是一个聚合类，钱对他来说就是一个聚合对象，他老婆就是一个客户端应用程序，银行卡就是一个迭代器！将检查是否有钱和取钱的功能分离给银行卡完成！这样他可以安心去

挣钱了！银行卡完成了一个迭代器的功能，有检查是否有钱和取钱的功能！

大B：“好了，说这么多，我们对迭代器模式有了个大概了解！”

小A：“在面向对象的软件设计中，我们经常会遇到一类集合对象，这类集合对象的内部结构可能有着各种各样的实现。”

大B：“归结起来，无非有两点是需要我们去关心的：一是集合内部的数据存储结构，二是遍历集合内部的数据。面向对象设计原则中有一条是类的单一职责原则，所以我们要尽可能的去分解这些职责，用不同的类去承担不同的职责。Iterator模式就是分离了集合对象的遍历行为，抽象出一个迭代器类来负责，这样既可以做到不暴露集合的内部结构，又可让外部代码透明的访问集合内部的数据。”

27.3 迭代器模式的角色

大B：“迭代器模式有哪些角色？”

小A：“1、迭代器角色（Iterator）：迭代器角色负责定义访问和遍历元素的接口。2、具体迭代器角色（Concrete Iterator）：具体迭代器角色要实现迭代器接口，并要记录遍历中的当前位置。3、容器角色（Container）：容器角色负责提供创建具体迭代器角色的接口。4、具体容器角色（Concrete Container）：具体容器角色实现创建具体迭代器角色的接口——这个具体迭代器角色于该容器的结构相关。”

迭代器模式的类图27-1如下：

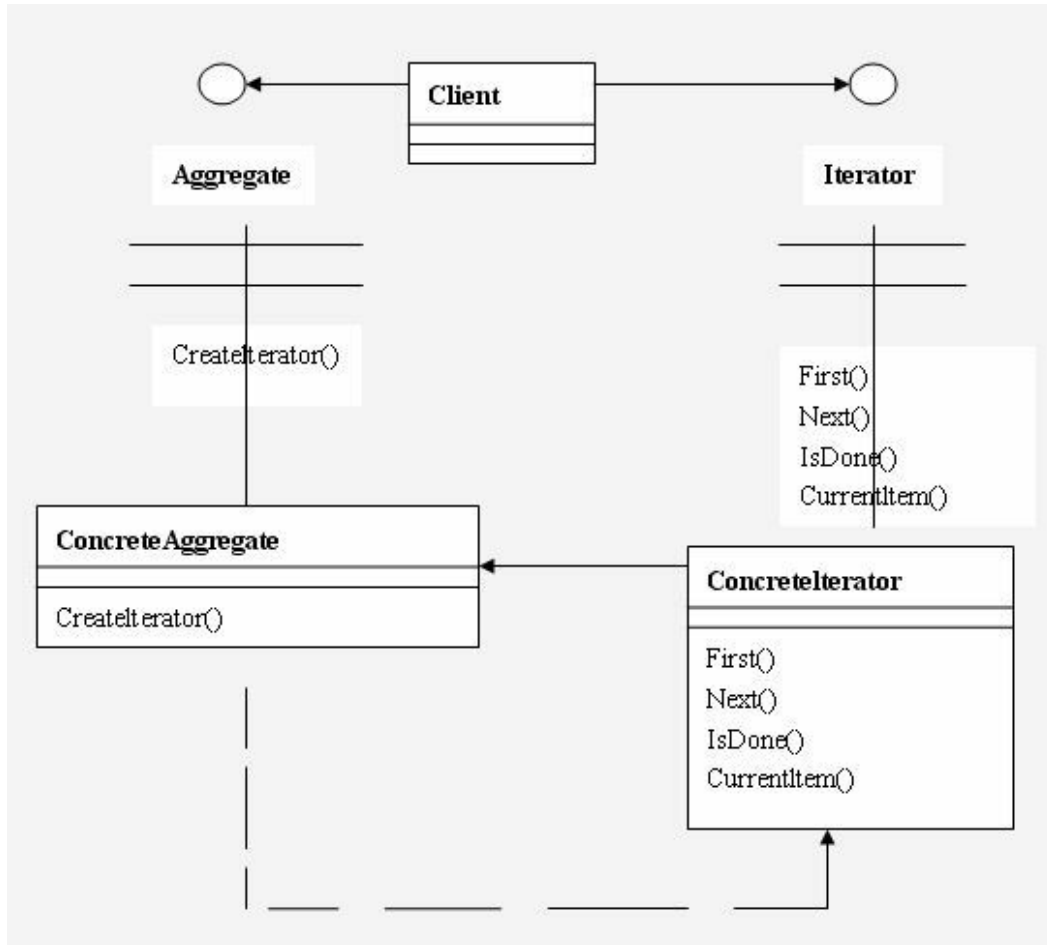


图27-1 迭代器模式的类图

小A：“从结构上，迭代器模式在客户与容器之间加入了迭代器角色。迭代器角色的加入，就可以很好的避免容器内部细节的暴露，而且也使得设计符号‘单一职责原则’。”

大B：“注意，在迭代器模式中，具体迭代器角色和具体容器角色是耦合在一起的——遍历算法是与容器的内部细节紧密相关的。为了使客户程序从与具体迭代器角色耦合的困境中脱离出来，避免具体迭代器角色的更换给客户程序带来的修改，迭代器模式抽象了具体迭代器角色，使得客户程序更具一般性和重用性。这被称为多态迭代。”

27.4 适用情况

小A：“迭代器模式给容器的应用会带来什么好处？”

大B：“1、支持以不同的方式遍历一个容器角色。根据实现方式的不同，效果上会有差别。2、简化了容器的接口。但是在java Collection中为了提高可扩展性，容器还是提供了遍历的接口。3、对同一个容器对象，可以同时进行多个遍历。因为遍历状态是保存在每一个迭代器对象中的。这样就得出迭代器模式的适用范围：1、访问一个容器对象的内容而无需暴露它的内部表示。2、支持对容器对象的多种遍历。3、为遍历不同的容器结构提供一个统一的接口（多态迭代）。”

27.5 实现自己的迭代器

大B：“在实现自己的迭代器的时候，一般要操作的容器有支持的接口才可以。”

小A：“喔。”

大B：“而且我们还要注意以下问题：在迭代器遍历的过程中，通过该迭代器进行容器元素的增减操作是否安全呢？在容器中存在复合对象的情况，迭代器怎样才能支持深层遍历和多种遍历呢？以上两个问题对于不同结构的容器角色，各不相同，值得考虑。”

27.6 用Iterator模式实现遍历集合

小A：“怎样用Iterator模式实现遍历集合？”

大B：“Iterator模式是用于遍历集合类的标准访问方法。它可以把访问逻辑从不同类型的集合类中抽象出来，从而避免向客户端暴露集合的内部结构。例如，如果没有使用Iterator，遍历一个数组的方法是使用索引：`for(int i=0; i`

小A：“那我们应该怎样去解决这此问题哩？”

大B：“为解决以上问题，Iterator模式总是用同一种逻辑来遍历集合：
`for(Iterator it = c.iterator(); it.hasNext();) { ... }`奥秘在于客户端自身不维护遍历集合的‘指针’，所有的内部状态（如当前元素位置，是否有下一个元素）都由Iterator来维护，而这个Iterator由集合类通过工厂方法生成，因此，它知道如何遍历整个集合。客户端从不直接和集合类打交道，它总是控制Iterator，向它发送‘向前’，‘向后’，‘取当前元素’的命令，就可以间接遍历整个集合。这样看来实现Iterator的目的是降低耦合以及实现统一的遍历模式吧。在JS里面，遍历数组和遍历Object是不一样的，一般数组是 `for(i=0; i`

27.7 迭代器模式的实现方式

大B：“由于迭代器模式本身的规定比较松散，所以具体实现也就五花八门。”

小A：“我们又应该用什么方法去实现迭代器模式？”

大B：“1、迭代器角色定义了遍历的接口，但是没有规定由谁来控制迭代。在Java collection的应用中，是由客户程序来控制遍历的进程，被称为外部迭代器；还有一种实现方式便是由迭代器自身来控制迭代，被称为内部迭代器。外部迭代器要比内部迭代器灵活、强大，而且内部迭代器在Java语言环境中，可用性很弱。2、在迭代器模式中没有规定谁来实现遍历算法。好像理所当然的要在迭代器角色中实现。因为既便于一个容器上使用不同的遍历算法，也便于将一种遍历算法应用于不同的容器。但是这样就破坏掉了容器的封装——容器角色就要公开自己的私有属性，在Java中便意味着向其他类公开了自己的私有属性。”

小A：“那我们把它放到容器角色里来实现好了。这样迭代器角色就被架空为仅仅存放一个遍历当前位置的功能。但是遍历算法便和特定的容器紧紧绑在一起了。”

大B：“而在Java Collection的应用中，提供的具体迭代器角色是定义在容器角色中的内部类。这样便保护了容器的封装。但是同时容器也提供了遍历算法接口，你可以扩展自己的迭代器。好了，我们来看下Java Collection中的迭代器是怎么实现的吧。”

```
//迭代器角色，仅仅定义了遍历接口
```

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove();  
}
```

```
//容器角色，这里以List为例。它也仅仅是一个接口，就不罗列出来了
```

```
//具体容器角色，便是实现了List接口的ArrayList等类。为了突出重点这里指罗列和迭代器相关的内容
```

```
//具体迭代器角色，它是以内部类的形式出来的。AbstractList是为了将各个具体容器角色的公共部分提取出来而存在的。
```

```

public abstract class AbstractList extends AbstractCollection implements List {
.....
//这个便是负责创建具体迭代器角色的工厂方法
public Iterator iterator() {
return new Itr();
}
//作为内部类的具体迭代器角色
private class Itr implements Iterator {
int cursor = 0;
int lastRet = -1;
int expectedModCount = modCount;
public boolean hasNext() {
return cursor != size();
}
public Object next() {
checkForComodification();
try {
Object next = get(cursor);
lastRet = cursor++;
return next;
} catch (IndexOutOfBoundsException e) {
checkForComodification();
throw new NoSuchElementException();
}
}
public void remove() {
if (lastRet == -1)
throw new IllegalStateException();
checkForComodification();
try {
AbstractList.this.remove(lastRet);
if (lastRet

```

大B：“至于迭代器模式的使用。如引言中所列那样，客户程序要先得到具体容器角色，然后再通过具体容器角色得到具体迭代器角色。这样便可以使用具体迭代器角色来遍历容器了。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第二十八章 指挥工人工作——访问者模式

28.1 指挥工人工作

时间：1月13日 地点：大B房间 人物：大B，小A

大B：“假如你有10个全能工人，10样相同工作。工作都要做完，而且大喊一声所有人去工作。”

小A：“嗯？”

大B：“当条件变了，工人不是全能，但是工作相同。”

小A：“ok，问题不大。”

大B：“条件再变，工作不是相同，但工人是全能。”

小A：“ok，问题不大。”

大B：“以上三种情况在现实生活中是很少发生的，最多的情况是这样：10个工人，每人会做一种工作，10样工作。你又一份名单写着谁做什么。但你不认识任何人。这个时候你怎么指挥呢？”

小A：“可以一个个的叫工人，然後问他们名字，认识他们，查名单，告诉他们做什么工作。”

大B：“这是个办法。还可以怎么指挥呢？”

小A：“你可以直接叫出他们名字，告诉他们干什么，不需要知道他是谁。”

大B：“看起来很简单。但如果你要指挥10万人呢？而且人员是流动的，每天的人不同，你每天拿到一张文档。”

小A：“这样的话就比较麻烦了。”

大B：“其实很简单，最常用的做法是，你把这份名单贴在墙上，然後大喊一声，所有人按照去看，按照自己的分配情况去做。”

小A：“嘿嘿！这个方法不错喔！”

大B：“这里利用的关键点是‘所有工人自己认识自己’，你不能苛求每个工人会做所有工作，不能苛求所有工作相同，但你能要求所有工人都认识自己。”

小A：“嗯。”

大B：“再想想我们开始的程序，每个工人对应着PA PB PC PD PE.....所有的工人都使工人P，每个工人会做的东西不一样runPA runPB runPC，你有一份名单Visitor（重载）记录着谁做什么工作。”

小A：“为什么不把这些方法的方法名做成一样的，那就可以解决了。例如，我们每个PA，PB，PC都加入一个run方法，然後run内部再调用自己对应的runPx()方法。”

大B：“有些时候从不同的角度考虑，或者因为实现的复杂度早成很难统一方法名。例如上边指挥人工作的例子的例子，其实run方法就是大叫一声去工作，因为每个工人只会做一种工作，所以能行，但我们不能要求所有人只能会做一种事情，这个要求很愚蠢。所以如果每个工人会干两种或者多种工作呢，也就是我PA有runPA() walkPA()等等方法，PB有runPB() climbPB()等等。这个时候按照名单做事才是最好的办法。”

28.2 访问者模式

小A：“怎样去定义访问者模式？”

大B：“它的通俗定义是：在每个自定义对象中预定义一个Accept(请求访问)方法，这个方法会以对象为参数，调用Visitor(访问者)对象的visit方法来操作这个对象。C#与Java运用多次重载来实现自动匹配接口，在JS中应该是内置了这种模式，所以真正的再仿效C#与Java去实现是多余的与笨拙的。在JS中，可以定义任意一个以this为目标替代符的函数，使所有的对象可以用call或者apply来临时以它们自身的名义运行，一旦代入后，就是函数为刀俎，对象为鱼肉，任其妄为了。”

28.3 访问者模式的角色

在计算计技术领域的很多技术上看起来很高深的东西，其实就是现有社会中人的生活方式的一种映射。而且这种方式是简单的不能再简单的方式。

大B：“访问者模式会涉及到一些角色。”

小A：“喔？它会涉及到哪些角色？”

大B：“抽象访问者：声明一个或者多个访问操作，形成所有的具体元素都要实现的接口。具体访问者：实现抽象访问者所声明的接口。抽象节点：声明一个接受操作，接受一个访问者对象作为参量。具体节点：实现了抽象元素所规定的接受操作。结构对象：遍历结构中的所有元素，类似List Set等。”

28.4 在什么情况下应当使用访问者模式

小A：“在什么情况下应当使用访问者模式？”

大B：“访问者模式应该用在被访问类结构比较稳定的时候，换言之系统很少出现增加新节点的情况。因为访问者模式对开 - 闭原则的支持并不好，访问者模式允许在节点中加入方法，是倾斜的开闭原则，类似抽象工厂。”

28.5 访问者模式的缺点

小A：“访问者模式有什么缺点？”

大B：“1、增加节点困难2、破坏了封装。因为访问者模式的缺点和复杂性，很多设计师反对使用访问者模式。个人感觉应该在了解的情况下考虑衡量选择。”

28.6 visitor模式准备

大B：“静态分派，动态分派，多分派，单分派是visitor模式准备。”

小A：“visitor模式准备？能不能详细讲讲，我不明白。”

大B：“可以。”

1、静态分派：

（1）定义：发生在编译时期，分派根据静态类型信息发生，重载就是静态分派

（2）什么是静态类型：变量被声明时的类型是静态类型

什么是动态类型：变量所引用的对象的真实类型

（3）有两个类,BlackCat ,WhiteCat都继承自Cat

如下调用

```
class Cat{}  
class WhiteCat extends Cat{}  
class BlackCat extends Cat{}  
public class Person {  
    public void feed(Cat cat){  
        System.out.println("feed cat");  
    }  
    public void feed(WhiteCat cat){  
        System.out.println("feed WhiteCat");  
    }  
}
```



```
public void feed(BlackCat cat){
    System.out.println("feed BlackCat");
}

public static void main(String[] args) {
    Cat wc = new WhiteCat();
    Cat bc = new BlackCat();
    Person p = new Person();
    p.feed(wc);
    p.feed(bc);
}
}
```

运行结果是：

feed cat

feed cat

这样的结果是因为重载是静态分派，在编译器执行的，取决于变量的声明类型，因为wc，bc都是Cat所以调用的都是feed(Cat cat)的函数。

2、动态分派

定义：发生在运行期，动态分派，动态的置换掉某个方法。

还是上边类似的例子：

```
class Cat{
    public void eat(){
        System.out.println("cat eat");
    }
}
```

```
}  
}  
public class BlackCat extends Cat{  
    public void eat(){  
        System.out.println("black cat eat");  
    }  
    public static void main(String[] args){  
        Cat cat = new BlackCat();  
        cat.eat();  
    }  
}
```

这个时候的结果是：

black cat eat

这样的结果是因为在执行期发生了向下转型，就是动态分派了。

3、单分派：

定义：根据一个宗量的类型进行方法的选择。

4、多分派：

(1) 定义：根据多于一个宗量的类型对方法的选择。

(2) 说明：多分派其实是一系列的单分派组成的，区别的地方就是这些但分派不能分割。

(3) C++，Java都是动态单分派，静态多分派语言。

小A：“访问同一类型的集合类是我们最常见的事情了，我们工作中这样的代码太常见了。”

```
Iterator ie = list.iterator();
while (ie.hasNext()) {
    Person
    p = (Person)ie.next();
    p.doWork();
}
```

这种访问的特点是集合类中的对象是同一类对象Person，他们拥有功能的方法run，我们调用的恰好是这个共同的方法。在大部份的情况下，这个是可以的，但在一些复杂的情况，如被访问者的继承结构复杂，被访问者的并不是同一类对象，也就是说不是继承自同一个根类。方法名也并不相同。例如Java GUI中的事件就是一个例子。

例如这样的问题，有如下类和方法：

```
类：PA ,方法：runPA();
类：PB ,方法：runPB();
类：PC ,方法：runPC();
类：PD ,方法：runPD();
类：PE ,方法：runPE();
```

有一个集合类List

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

```
list.add(new PA());
list.add(new PB());
list.add(new PC());
list.add(new PD());
list.add(new PE());
....
```

大B：“要求能访问到每个类的对应的方法。我们第一反应应该是这样的。”

```
Iterator ie = list.iterator();
while (ie.hasNext()) {
    Object obj = ie.next();
    if (obj instanceof PA) {
        ((PA)obj).runPA();
    } else if (obj instanceof PB) {
        ((PB)obj).runPB();
    } else if (obj instanceof PC) {
        ((PC)obj).runPC();
    } else if (obj instanceof PD) {
        ((PD)obj).runPD();
    } else if (obj instanceof PE) {
        ((PE)obj).runPE();
    }
}
```

大B：“当数目变多的时候，维护if else是个费力气的事情：仔细分析if,else做的工作，首先判断类型，然後根据类型执行相应的函数。”

小A：“如何才能解决这两个问题呢？”

大B：“首先想到的是Java的多态，多态就是根据参数执行相应的内容，能很容易的解决第二个问题，我们可以写这样一个类。”

```
public class visitor {  
    public void run(PA pa) {  
        pa.runPA();  
    }  
    public void run(PB pb) {  
        pb.runPB();  
    }  
    public void run(PC pc) {  
        pc.runPC();  
    }  
    public void run(PD pd) {  
        pd.runPD();  
    }  
    public void run(PE pe) {  
        pe.runPE();  
    }  
}
```

大B：“这样只要调用run方法，传入对应的参数就能执行了。”

小A：“还有一个问题就是判断类型。”

大B：“由于重载(overloading)是静态多分配。Java语言本身是支持‘静态多分配’的。所以造成重载只根据传入对象的定义类型，而不是实际的类型，所以必须在传入前就确定类型，这可是个难的问题，因为在容器中对象全是Object，出来后要是判断是什么类型必须用if (xx instanceof xxx)这种方法。”

小A：“如果用这种方法启不是又回到了原点，有没有什么更好的办法呢？我们知道Java还有另外一个特点，覆写(overriding)，而覆写是‘动态单分配’的，那如何利用这个来实现呢？”

大B：“看下边这个方法：我们让上边的一些类PA PB PC PD PE都实现一个接口P，加入一个方法，accept()。”

```
public void accept(visitor v) {  
    // 把自己传入1  
    v.run( this );  
}
```

然后在visitor中加入一个方法

```
public void run(P p) {  
    // 把自己传入2  
    p.accept( this );  
}
```

// 这样你在遍历中可以这样写

```
Visitor v = new Visitor();  
Iterator ie = list.iterator();  
while (ie.hasNext()) {  
    P p = (P)ie.next();  
    p.accept(v);  
}
```

```
}
```

大B：“首先执行的是‘把自己传入2’，在这里由于Java的特性，实际执行的是子类的accept()，也就是实际类的accept然後是‘把自己传入1’，在这里再次把this传入，就明确类型，ok我们巧妙的利用overriding解决了这个问题。其实归纳一下第二部分，一个关键点是‘自己认识自己’，是不是很可笑。”

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！

第二十九章 大学生毕业3条出路：学、仕、商——设计模式总结

29.1 大学生毕业3条出路：学、仕、商

时间：2月7日 地点：大B房间 人物：大B，小A

小A：“师兄，在就业形势日益激烈的今天，大学毕业生应该如何面对求职难的问题。”

大B：“按照现今社会的主流说法，大学生毕业后的走向大体有三条——学、仕、商。”

小A：“学、仕、商？为什么？”

大B：“上大学以前，我的最初想法是，本科毕业后，先从商，成也好，败也好，人生嘛，什么都应该历一下。等自己喜欢安定了，就去感受一下仕途。”

小A：“嗯！我现在也是这样认为的。”

大B：“但上了大学，了解了很多现实，很多跟理想差距很大的现实，现阶段我们要做的就是选好一条有利于自己的并且可行的路，然后按照要求和计划，努力提

高自己相关方面的能力。”

小A：“我也曾这么想过。”

大B：“但来到大学以后，比较充分了解了社会现状，而这个现状使得我不得不放弃原来的思路，而且一些新问题比我原来想象的要复杂得多。而现在，我首先想提出的问题是本科毕业后，是考研还是就业。”

小A：“嗯，对！这是我一直在想的问题。”

大B：“是啊！这个问题离我们比较近，所以考虑得比较多。就选择而言，两个都有各自的好处和风险。考研的好处在于，毕业后，可以有个更高的台阶去面对社会，个人的第一身份也比本科生高得多。但它的风险在于，选择考研，至少要投入三年以上的的时间，更遭的是有很多人成为了所谓的考研专业户。而如果把这笔时间投入到社会上，说不定已经有不小的成就。但提前走入社会的风险在于可能几年时间下来，还是一事无成，那还不如多读几年书，有个更高的学识和文凭。总的来说，就是把本科后的四年作为一个发展段，把考研与就业带来的收益和风险作权衡，其中有个零界点，现阶段迷茫的是，以个人自身的条件，应该站在选择的自我定位在界限的哪边，选择的自我定位有哪些标准，怎样努力才能最大限度有利于自己的定位。”

小A：“嗯！师兄的话真让我受益非浅。说的正是我现在困惑的问题。”

大B：“还有一个疑问是职业困惑，就是离开学校，无论是本科毕业还是硕士毕业，选择职业的切入点在哪，或者说以什么样的标准来区分到底是去考公务员，还是自己在政途以外的地方寻找落脚点才能最大限度地获取自己能创造的价值。”

小A：“嗯！”

大B：“说到这，很汗颜，因为这些选择都有很强的趋利性。最后，在找到自己先天所适合的路以后，怎样培养自己，使得自己更适合这条路，沿着适合自己的道路努力奋进，假以时日，收获成功就不是一句空话了！”

29.2 设计模式总结

大B：“刚毕业的大学生面临人生的这一大转折。要好好准备，去迎接机遇与挑战。现在我们就来聊聊设计模式吧。学了这么久的设计模式，你有什么感想吗？今天我们来总结一下设计模式吧。”

小A：“刚开始学习设计模式的时候，感到这些模式真的非常抽象。在设计过程中，我发现了很多设计模式的用处，也确实应用了很多设计模式，这让我越来越感到设计模式的重要性。”

大B：“设计模式是个好东西，它给出了很多设计中的技巧与思路，对于很多优秀的设计，它加以总结与提炼。设计模式并非是人拍脑瓜想出来的，而是他们搜集了其他人优秀的设计，加以整理出来的，他们不是这些模式的创造者，仅仅是整理者。”

小A：“应用设计模式还给我们带来了很多好处。”

大B：“是啊！软件将变得更加灵活，模块之间的耦合度将会降低，效率会提升，开销会减少。更重要的，设计模式就好像美声唱法中的花腔，让你的设计更加

漂亮。总的来说，设计模式似乎将软件设计提升到艺术的层次。”

小A：“设计模式已经被广泛的应用了，在现在很多的图形界面框架都使用了MVC模式，大量迭代器模式的应用，彻底改变了我们对集合的操作方式。不仅如此，应用了设计模式的设计，往往被看成为优秀的设计。这是因为，这些设计模式都是久经考验的。”

大B：“在学习和使用设计模式的时候，往往出现一个非常严重的误区，那就是设计模式必须严格地遵守，不能修改。但是设计模式不是设计模型，并非一成不变。正相反，设计模式中最核心的要素并非设计的结构，而是设计的思想。只有掌握住设计模式的核心思想，才能正确、灵活的应用设计模式，否则再怎么使用设计模式，也不过是生搬硬套。”

小A：“当然，掌握设计模式的思想，关键是要仔细研究模式的意图和结构。一个模式的意图，就是使用这个设计模式的目的，体现了为什么要使用这个模式，也就是需求问题。”

大B：“是啊！这个模式的结构，就是如何去解决这个问题，是一种手段、一种经典的解决方法，这种解决方法只是一种建议。两个方面结合起来，明白为什么需要设计模式，同时明白了如何实现这个模式，就容易抓住模式的本质思想。”

小A：“在抓住意图和结构的基础上，实践也是掌握设计模式的必要方法。”

大B：“当然，设计模式必须在某个场景下得到应用才有意义，这也是为什么要提供大量的例子用来说明模式的应用场景，这实际上是提供了一种上下文环境。学外语不是要强调‘语言环境’么，学习设计模式也是这样。”

小A：“嗯！是的。”

大B：“看到网上很多人在讨论设计模式，他们确实很有创意，满嘴都是模式的名字，恨不得写个Hello World都要应用到设计模式。设计模式确实是好东西，但是，中国有句古话叫作物极必反，即便是按照辩证法，事物总要一分为二的看。”

小A：“是啊！我们说设计模式的目的是为了让软件更加灵活，重用度更高。”

大B：“但是，某种意义上，设计模式增加了软件维护的难度，特别是它增加了对象之间关联的复杂度。”

小A：“嗯。对！”

大B：“我们总说，重用可以提高软件开发的效率。如果你是大牛，你自然希望你的设计可以被反复使用10000年，那就是：当世界毁灭的时候，你的设计依然存在。然而，现实是一个系统的设计往往在5年之内就会被抛弃，这是因为：1、软件技术产生了新的变化，使用新的技术进行的设计，无论如何都比你的设计好；2、硬件环境发生了很大变化，你的设计里对开销或者效率的追求已经没有意义了；3、新的大牛出现了，并且取代了你的位置。”

小A：“应用设计模式会导致设计周期的加长，因为更复杂了，但是很多项目还在设计阶段就已经胎死腹中，再好的设计也没有发挥的余地。”

大B：“当我们向设计模式顶礼膜拜的时候，我们还必须清醒地看到软件生产中非技术层面上的东西往往具有决定性作用。理想固然崇高，但现实总是残酷的。如何看清理想与现实的界限，恐怕是需要我们在实践中不断磨砺而体会出来的。在看完设计模式后，不妨反问以下自己，这些模式究竟能给你带来什么？”

29.3 常见的23个设计模式概念

小A：“所有结构良好的面向对象软件体系结构中都包含了许多模式。”

大B：“实际上，当我们评估一个面向对象系统的质量时，所使用的方法之一就是要判断系统的设计者是否强调了对象之间的公共协同关系。在系统开发阶段强调这种机制的优势在于，它能使所生成的系统体系结构更加精巧、简洁和易于理解，其程度远远超过了未使用模式的体系结构。”

小A：“喔。”

大B：“这23个设计模式便是总结了面向对象设计中最有价值的经验，并且用简洁可复用的形式表达出来。”

1、Abstract Factory 抽象工厂模式——提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

2、Adapter 适配器模式——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一起工作的那些类可以一起工作。

3、Bridge 桥接模式——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。

4、Builder 生成器模式——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。

5、Chain of Responsibility 职责链模式——为解除请求的发送者和接收

者之间耦合，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。

6、Command 命令模式——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，以及支持可取消的操作。

7、Composite 组合模式——将对象组合成树形结构以表示“部分-整体”的层次结构。Composite使得客户对单个对象和复合对象的使用具有一致性。

8、Decorator 装饰模式——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。

9、Facade 外观模式——为子系统的一组接口提供一个一致的界面，Facade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。

10、Factory Method 工厂方法模式——定义一个用于创建对象的接口，让子类决定将哪一个类实例化。Factory Method使一个类的实例化延迟到其子类。

11、Flyweight 享元模式——运用共享技术有效地支持大量细粒度的对象。

12、Interpreter 解释器模式——给定一个语言，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。

13、Iterator 迭代器模式——提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。

14、Mediator 中介者模式——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。

15、Memento 备忘模式——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。

16、Observer 观察者模式：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，所有依赖于它的对象都得到通知并自动刷新。

17、Prototype 原型模式——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。

18、Proxy 代理模式：为其他对象提供一个代理以控制对这个对象的访问。

19、Singleton 单态模式——保证一个类仅有一个实例，并提供一个访问它的全局访问点。

20、State 状态模式：允许一个对象在其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。

21、Strategy 策略模式——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。

22、Template Method 模板方法模式——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。

23、Visitor 访问者模式——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。

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质
电子书下载！！！！

附录：面向对象基础

小A：“为什么要‘面向对象’？”

大B：“面向对象方法使构建系统更容易，因为：解决正确的问题，正常工作，易维护，易扩充，易重用。大家发现面向对象更易理解，实现可以更简单。把数据和功能组合在一起简单而自然，分析和实现之间的概念跨度更小，设计良好的一组对象能弹性地适应重用和变化，可视化模型提供更有效的沟通，建模过程有助于创建通用词汇以及在开发者和用户/客户之间达成共识。非计算机编程人员也能理解对象模型 这些好处可以使用面向对象方法获得，但面向对象方法不能保证这一点。”

小A：“怎样才能变成优秀的面向对象设计者？”

大B：“只有靠经验和聪明的头脑才能做到。”

■ 过程化方法 (The Procedural Approach)

小A：“怎样过程化方法？”

大B：“系统由过程(procedures)组成，过程之间互相发送数据，过程和数据各自独立，集中于数据结构、算法和运算步骤的先后顺序，过程经常难以重用，缺乏具有较强表现力的可视化建模技术，分析与实现之间需要进行概念转换，本质上

是机器/汇编语言的抽象，从设计模型到代码实现跨度很大。”

■ 面向对象方法

大B：“系统由对象组成，对象互相发送消息（过程调用）相关的数据和行为紧密地绑定在对象中，把问题领域建模成对象，要解决的问题自然的映射为代码的实现，可视模型表现力强，相对容易理解 集中于实现之前所确定的职责（responsibilities）和接口。强有力的概念：接口，抽象，封装，继承，委托（delegation）和多态。问题的可视模型逐渐进化成解决方案模型，设计模型与代码实现之间跨度较小努力缩减软件的复杂度。”

■ 温度换算

大B：“下面我就以温度换算为例。”

过程/函数化方法

```
float c = getTemperature(); // 假定为摄氏度 Celsius
```

```
float f = toFahrenheitFromCelcius( c );
```

```
float k = toKelvinFromCelcius( c );
```

```
float x = toKelvinFromFahrenheit( f );
```

```
float y = toFahrenheitFromKelvin( k );
```

面向对象方法

```
Temp temp = getTemperature();
```

```
float c = temp.toCelcius();
```

```
float f = temp.toFarenheit();
```

```
float k = temp.toKelvin();
```

包含有数据的Temp的内部单元是什么？

■ 建模 (Modeling)

小A：“成功的程序能解决真实世界的问题。”

大B：“嗯，是的。它们紧密对应于需要解决的问题。对问题领域和用户活动进行建模。”

小A：“建模促进与用户更好的可视化交流。”

大B：“成功的面向对象设计总是一开始就由领域专家和软件设计者建立一个反映问题领域的可视化的‘对象模型’。”

小A：“嗯。是的。”

大B：“你愿意让承包人在没有设计蓝图的情况下建造你的新房子吗？”

小A：“那当然不行啦。”

■ 对象

大B：“你知道怎样去理解什么是对象吗？”

小A：“对象代表真实或抽象的事物，有一个名字，有明确的职责（well-defined responsibilities），展示良好的行为（well-defined behavior），接口清晰，并且尽可能简单、自相容，内聚，完备（self-consistent,coherent,and complete）。”

大B：“嗯，对。（通常）不是很复杂或很大，只需要理解自己 and 一小部分其他对象的接口，与一小部分其它对象协同工作(team players)，尽可能地与其它对象松散耦合（loosely coupled），很好地文档化，以便他人使用或重用，对象是类的实例，每一个对象都有唯一的标识，类定义一组对象的接口和实现，即定义了这些对象的行为，抽象类不能拥有实例，只要有抽象类（如宠物），通常就会有能够实例化的具体类（如猫，狗等），一些面向对象语言（如Smalltalk）支持元类（metaclass）的概念，程序员可以随时（on-the-fly）定义一个类，然后实例化。这种情况下，类也是一个对象，即元类。对象一旦实例化，就不能更改它的类。”

■ 对象的特征

大B：“那你知道对象有什么特征吗？”

小A：“有唯一标识，可以分成许多种类（即类），可以继承或聚合。行为、职责明确，接口与实现分离，隐藏内部结构，有不同的状态，可以提供服务，可以给

其它对象发送消息，从其它对象接收消息，并做出相应响应，可以把职责委托给其它对象。”

大B：“对，说得非常全面。”

■ 类

小A：“怎么样才叫类呢？”

大B：“有公共的属性和行为的一组对象可以抽象成为类，对象通常根据你所感兴趣的属性而分类。”

小A：“喔。”

大B：“例如：街道，马路，高速公路... 不同的程序对它们分类也不同。交通模拟器程序，单行道，双通道，有分车道的，住宅区的，限制通行的维护调度程序，路面材料，重型卡车运输类本身也可以有属性和行为。例如：养老金管理程序中的“雇员”类雇员总数，雇员编制多少，不同语言对类的支持略有不同：

Smalltalk 把类当作对象（很有好处），C++提供最小限度的支持（有时会带来很多烦恼），Java位于上述两者之间，类也是对象，类可以有属性“雇员”类可以有一个包含其所有实例的列表（list）“彩票”类可以有一个种子（seed）用于产生随机票号，该种子被所有实例共享，类可以有行为，雇员”类可以有 `getEmployeeBySerialNum` 行为。“彩票”类可以有 `generateRandomNumber` 行为。”

■ 封装

大B：“只暴露相关的细节，即公有接口（public interface）。”

小A：“封装什么？如何封装？”

大B：“隐藏“齿轮和控制杆”只暴露客户需要的职责，防止对象受到外界干扰，防止其它对象依赖可能变化的细节，信息隐藏有助于对象和模块之间的松散耦合，使得设计更加灵活，更易于重用，减少代码之间的依赖，“有好篱笆才有好邻居”。例如：汽车的气动踏板。”

小A：“怎样才能更好地实践？”

大B：“最佳实践：对象之间只通过方法（函数）互相访问。切忌直接访问属性。”

```
class Person {  
    public int age;  
}  
  
class BetterPerson {  
    private int age; // change to dateOfBirth  
    public int getAge() { return age; }  
}
```

更完善的 Person 类可能是：private dateOfBirth

■ 抽象

小A：“什么是抽象？”

大B：“抽象使得泛化（generalizations）成为可能，简化问题-忽略复杂的细

节，关注共性，并且允许变更，人类经常使用泛化。 当你看见约翰和简家里的那头灰德国牧羊犬时，你有没有.....想到“狗”这个词？抽象同样能简化计算机程序。例如，软件中有两个重要抽象：客户端和服务器（clients and servers）。

小A：“喔。”

大B：“在图形用户界面中，系统可能会询问用户各种问题：是或不是多选一？输入数字，任意文本问题统一处理这些问题会显得很简单，每一个问题都作为Question类的特例（specialization）；程序只需维护这些问题的实例列表，分别调用各自的askTheUser()方法。”

■ 继承

小A：“什么是继承？”

大B：“继承用于描述一个类与其它类的不同之处。例如：类Y像类X，但有下列不同...”

小A：“为什么使用继承？”

大B：“你有两种类型，其中一种是另一种的扩展。有时（不是所有时候）你想忽略对象之间的不同，而只关注它们的共同之处（基类）。这就是泛化。假如某系统需要对不同的形状进行操作（经典例子）：有时你并不关心你正在操作的形状的种类（例如，移动形状时）有时你必须知道形状的种类（在显示器上绘制形状）”

小A：“怎样去理解派生类？”

大B：“派生类继承自基类；派生类扩展了基类；派生类是基类的特殊化（specialization）。派生类能够提供额外的状态（数据成员），或额外的行为（成员函数/方法），或覆盖所继承的方法。基类是所有它的派生类的泛化。如：通常所有宠物都有名字。基类（Base Class）=父类（parent class）=超类（superclass）派生类（Derived Class）=子类（child class）=子类（subclass）”

小A：“喔。”

大B：“继承含有（有些，不是全部）是一个（is-a）或是一种（is-a-kind-of）的关系，正方形是一种矩形（使用继承），Leroy 是一种狗（不使用继承），传统的过程分析和设计中不能很好地模拟这种关系。继承是一种强有力的机制，使我们关注共性，而不是特定的细节。使得代码可以重用且富有弹性（能适应变化）。”

小A：“怎样去实现继承？”

大B：“实现继承（Implementation inheritance）：派生类继承基类的属性和行为。”

小A：“又应该怎样去接口继承？”

大B：“接口继承（Interface inheritance）：类实现抽象接口的方法，保留既定语义（intended semantics） C++允许多重实现继承。Java规定派生类只能有一个基类，但可以继承自多个接口。”

■ 多态

小A：“什么是多态？”

大B：“多态是一种允许多个类针对同一消息有不同的反应的能力。对于任何实现了给定接口的对象，在不明确指定类名的情况下，就可以使用。例如：
question.askTheUser(); 当然，这些不同反应都有类似的本质 尽可能使用接口
继承和动态（运行期）绑定 Liskov 替换原则：如果Y是X的子类，那么在任何使用
X实例的地方都可以用Y的实例来替换。”

演示多态的Java代码

```
// File: question/QuestionTest.java
// 下面的代码将输出什么？
// Refer to the Beginning Java link on the course web site.
package question;
abstract class Question { // Full class name is question. QuestionTest
public Question( String _text ) { // Constructor
theText = _text;
}
public abstract void askTheUser();
protected String theText;
}
class YesNoQuestion extends Question {
public YesNoQuestion( String _text ) { super( _text ); }
public void askTheUser() {
System.out.println( theText );
System.out.println( "YES or NO ...?" );
}
}
class FreeTextQuestion extends Question {
public FreeTextQuestion( String _text ) { super( _text ); }
public void askTheUser() {
```

```
System.out.println( theText );
System.out.println( "Well...? What' s the answer...?" );
}
}
public class QuestionTest {
public static void main(String[] args) {
Question[] questions = getQuestions();
for (int i = 0; i
```

输出：

Do you understand polymorphism?

YES or NO ...?

Why is polymorphism good?

Well...? What's the answer...?

更多的Java例子

```
// File: Derived.java
// What will the following Java code output to the screen?
class Base {
final void foo() {
System.out.println("Base foo");
}
void bar() {
System.out.println("Base bar");
}
}
public class Derived extends Base {
```

```
void bar() {  
    System.out.println("Derived bar");  
}  
  
public static void main(String[] args) {  
    Derived d = new Derived();  
    d.foo();  
    d.bar();  
    Base b = d;  
    b.bar();  
}  
}
```

输出：

Base foo

Derived bar

Derived bar

■ 为什么面向对象有效

小A：“为什么面向对象有效？”

大B：“首先是减小复杂度。”

小A：“嗯。”

大B：“我们从封装、多态、继承、委托来具体说。”

小A：“喔？”

大B：“封装：只暴露公有接口，隐藏了复杂的实现细节，避免代码之间复杂的相互依赖。多态：允许有相同接口的类互相替换，由此减小代码的复杂度。继承：使用抽象类或接口实现泛化来减小复杂度。委托：通过从更小、封装更好的服务来构建更完整或更高层次的服务来减小复杂度。委托还增加了运行时的灵活性。”

小A：“面向对象有效我们是不是还可以从语言学和辨识角度来说？”

大B：“是的。我们主要使用名词，然后对它进行修饰和增加属性，最后，把它和动词联合在一起。面向对象设计遵循这个模式，过程化设计不遵循这个模式，这就是为什么人们经常发现对象更容易理解。我们从对象模型中能形成构造良好的主谓宾（Subject-verb-object）句子：人们拥有宠物。Paula 拥有Leroy。试试用功能分解来形成主谓宾句子！而且，人们广泛使用抽象和泛化...”

■ 面向对象是编程进化一个自然阶段

小A：“为什么说面向对象是编程进化一个自然阶段？”

大B：“首先出现机器语言。在此基础上发展出汇编语言，提供了符号。高级语言出现：Fortran, Pascal, C等。它们提供了程序语句之间的“结构”关系。“goto”的使用日渐稀少，这有助于简化程序结构。数据结构和算法提供了程序结构的可重用模式，促进更高层次上的抽象。面向对象的抽象是为了关注于解决问题，而不是机器。通过更高层次的抽象，程序语句之间的关系转化成为相对简单的对象协作关系，设计模式提供可重用的对象结构...”

■ 面向对象更多的好处

大B：“面向对象还有更多的好处。”

小A：“是吗？都还些什么好处哩？”

大B：“组件非常有用，代码重用。设计模式很好，设计重用。接口不错，灵活健壮的代码。底层结构（`infrastructure`）和可重用服务同样很好。接口能完美分离个人和团队的职责，增加团队效率，松散耦合和模块化提高了扩展性、灵活性、可量测性和重用性。逻辑变化很自然的被隔离起来，这多亏了对象的模块化和信息隐藏（封装）。这意味着实现更快，维护更容易。面向对象中间件使我们无须关注位置、平台和语言。组设计良好的对象是我们可以增加新功能而不用更改设计。”

■ 好的面向对象设计

小A：“什么样的好的面向对象设计？”

大B：“艺术多于科学。可解决问题的模型本质上当然没有问题，但是一些模型就是比其它的好，这是因为它们更实用、更灵活、更容易扩展、更方便理解、更简单...第一个设计几乎不可能是最好的设计。找到最好的抽象来对问题建模始终不是一件容易的事情，经验很重要。”

小A：“经常需要尝试多次，来确定如何划分系统各部分之间的边界才是最好？每一部分应该为其它部分提供什么接口？”

大B：“以体系结构为中心，而不是功能为中心。首先关注全面的大体的，其次

才是具体的特定的。设计时首先做到这一点，就成功了一大半。设计中要考虑可能发生的扩展，使得以后扩展是递增式的，不用更改设计。不同的设计可以有完全相同的功能，但是在这方面可能完全不同。尽量推广可重用的面向服务的底层结构，这样，在需求不可避免的变化时，代码也能更快、更容易的更改。”

■ 职责

小A：“什么是职责？”

大B：“是面向对象分析中采用的最普遍的方法。基于“客户端/服务器”关系，对“客户端/服务器”有两种通用的解释：用于分布式体系中，服务器提供对共享资源（如数据库）的访问，客户端提供用户界面。用于面向对象术语中，服务器是一个提供服务的对象；在这里我们使用这个含义，客户端与服务器协作（发送消息）。一个对象可能在一个协作中是客户端，而在另一个协作中是服务器。服务器负责提供某种服务，一般来说，对象应该以某种定义良好的方式工作。”

■ 设计过程概述

大B：“我们讲了这么多，你知道设计过程是什么吗？”

小A：“查看领域，识别对象、类。通常首先识别出对象，通过对象分组找到类，确定对象之间和类之间的关系，结构关系，协作关系，赋予职责，基于协作关系，迭代,迭代,迭代,迭代,迭代,迭代...以领域建模作为开始，而不是以建模解决方案作为开始。”

■ CRC卡片

小A：“什么是CRC卡片？”

大B：“CRC方法使用3×5（英寸）索引卡片，一个类就是一张卡片，卡片上写有该类的职责以及为了完成这些职责必须与哪些类协作。类的简要描述写在卡片背面。下面的例子中，类Foo必须与类X和类Y协作（给它们发送消息），以完成“do something”责任。”

■ 例子：“棍子”游戏

游戏设计两个玩家使用一台计算机来一起玩。游戏中许多棍子按行排列，当游戏开始时，它们如下排列：

1: |

2: | |

3: | | |

4: | | | |

■ 游戏规则

玩家轮流参加，每人可以从任何一个非空行中移走一根或多根棍子。移走最后一根棍子的人为输家。

游戏开始时，程序将显示游戏的状态：轮到谁了，还有几行，还有多少棍子。

操作不符合规则，程序将给出提示。（如所移走的棍子数目超过该行的棍子总数）

找到对象和类...

用CRC卡片

附加问题：哪个类负责记录轮到那个玩家了？

■ 词汇 (Vocabulary)

类 (Class)

– 抽象 (Abstract) / 具体 (Concrete) / 元 (Meta)

对象 (Object)

– 实例 (Instance)

– 标识 (Identity)

属性 (Attribute)

– 成员 (Member)

– 域 (Field)

– 状态 (State)

行为 (Behavior)

- 方法 (Method)
- 成员函数 (Member Function)
- 操作 (Operation)
- 职责 (Responsibility)
- 消息 (Message)
- 调用方法 (Method Call)

接口 (Interface)

抽象 (Abstraction)

- 泛化 (Generalization)
- 特殊化 (Specialization)

继承 (Inheritance)

- 接口 (Interface) / 实现 (Implementation)
- 基类 (Base) / 派生类 (Derived) , 父类 (Parent) / 子类 (Child) , 超类 (Super) / 子类 (Sub)

委托 (Delegation)

协作 (Collaboration)

多态 (Polymorphism)

- Liskov 替换原则 (Liskov Substitution Principle)

- 动态绑定 (Dynamic (run-time) Binding)

聚合 (Aggregation)

底层结构 (Infrastructure)

- 服务 (Services) / 中间件 (Middleware) / 框架 (Frameworks)

统一建模语言 (Unified Modeling Language (UML))

分析 (Analysis) / 设计 (Design) / 实现 (Implementation) / 架构
(Architecture) / 过程 (Process)

松散耦合 (Loose Coupling & Flexibility)

封装 (Encapsulation)

- 信息隐藏 (Information Hiding)

模块性 (Modularity)

透明 (Transparency)

Java

– 构造函数 (Constructor) / 包 (Package) / 静态 (Static)

模式 (Patterns)

职责驱动设计 (Responsibility driven design)

■ 设计模式与面向对象

小A：“面向对象设计模式主要是解决什么问题哩？”

大B：“面向对象设计模式解决的是“类与相互通信的对象之间的组织关系，包括它们的角色、职责、协作方式几个方面。面向对象设计模式是‘好的面向对象设计’。

小A：“什么是‘好的面向对象设计’？”

大B：“所谓‘好的面向对象设计’是那些可以满足‘应对变化，提高复用’的设计。”

小A：“这么说来，面向对象设计模式主要都是讲些什么哩？”

大B：“面向对象设计模式描述的是软件设计，因此它是独立于编程语言的，但是面向对象设计模式的最终实现仍然要使用面向对象编程语言来表达，基于C#语言，但实际上它适用于支持.NET框架的所有.NET语言，如Visual Basic.NET、C++/CLI等。面向对象设计模式不像算法技巧，可以照搬照用，它是建立在对“面向对象”纯熟、深入的理解的基础上的经验性认识。掌握面向对象设计模式的前提是首先掌握“面向对象”！从编程语言直观了解面向对象，各种面向对象编程语言

相互有别，但都能看到它们对面向对象三大机制的支持，即：“封装、继承、多态”

- 封装，隐藏内部实现
- 继承，复用现有代码
- 多态，改写对象行为

使用面向对象编程语言，可以推动程序员以面向对象的思维来思考软件设计结构，从而强化面向对象的编程范式。C#是一门支持面向对象编程的优秀语言，包括：各种级别的封装支持；单实现继承+多接口实现；抽象方法与虚方法重写。但OOPL并非面向对象的全部。通过面向对象编程语言（OOPL）认识到的面向对象，并不是面向对象的全部，甚至只是浅陋的面向对象。OOPL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（例如C#），就实现了面向对象的设计与开发！因此我们不能依赖编程语言的面向对象机制，来掌握面向对象。”

小A：“OOPL没有回答面向对象的根本性问题——我们为什么要使用面向对象？我们应该怎样使用三大机制来实现“好的面向对象”？我们应该遵循什么样的面向对象原则？”

大B：“任何一个严肃的面向对象程序员（例如C#程序员），都需要系统地学习面向对象的知识，单纯从编程语言上获得的面向对象知识，不能够胜任面向对象设计与开发。”

从一个示例谈起

示例场景：

我们需要设计一个人事管理系统，其中的一个功能是对各种不同类型的员工，计算其当月的工资——不同类型的员工，拥有不同的薪金计算制度。

结构化做法

1. 获得人事系统中所有可能的员工类型

2. 根据不同的员工类型所对应的不同的薪金制度，计算其工资

```
enum EmployeeType
{
    Engineer;
    Sales;
    Manager;
    ...
}
// 计算工资程序
if ( type == EmployeeType.Engineer)
{
    .....
}
else if (type == EmployeeType.Sales)
{
    .....
}
```

面向对象设计

1.根据不同的员工类型设计不同的类，并使这些类继承自一个Employee抽象类，其中有一个抽象方法GetSalary。

2.在各个不同的员工类中，根据自己的薪金制度，重写（override）GetSalary方法。

```
abstract class Employee
{
...
public abstract int GetSalary();
}
class Engineer: Employee
{
...
public override int GetSalary()
{
...
}
}
class Sales: Employee
{
...
public override int GetSalary()
{
...
}
}
// 显示工资程序
Employee e = emFactory.GetEmployee(id);
MessageBox.Show( e.GetSalary());
```

示例场景：

现在需求改变了.....随着客户公司业务规模的拓展，又出现了更多类型的员工，比如钟点工、计件工.....等等，这对人事管理系统提出了挑战——原有的程序必须改变。

结构化做法，几乎所有涉及到员工类型的地方（当然包括“计算工资程序”）都需要做改变.....这些代码都需要重新编译，重新部署.....面向对象做法，只需要在新的文件里增添新的员工类，让其继承自Employee抽象类，并重写GetSalary()方法，然后在EmployeeFactory.GetEmployee方法中根据相关条件，产生新的员工类型就可以了。其他地方（显示工资程序、Engineer类、Sales类等）则不需要做任何改变。重新认识面向对象，对于前面的例子，从宏观层面来看，面向对象的构建方式更能适应软件的变化，能将变化所带来的影响减为最小。从微观层面来看，面向对象的方式更强调各个类的“责任”，新增员工类型不会影响原来员工类型的实现代码——这更符合真实的世界，也更能控制变化所影响的范围，毕竟Engineer类不应该为新增的“钟点工”来买单.....

小A：“对象是什么？”

大B：“从概念层面讲，对象是某种拥有责任的抽象。从规格层面讲，对象是一系列可以被其他对象使用的公共接口。从语言实现层面来看，对象封装了代码和数据。”

小A：“有了这些认识之后，怎样才能设计“好的面向对象”？”

大B：“遵循一定的面向对象设计原则熟悉一些典型的面向对象设计模式。从设计原则到设计模式，针对接口编程，而不是针对实现编程。客户无需知道所使用对

象的特定类型，只需要知道对象拥有客户所期望的接口。优先使用对象组合，而不是类继承。类继承通常为“白箱复用”，对象组合通常为“黑箱复用”。继承在某种程度上破坏了封装性，子类父类耦合度高；而对象组合则只要求被组合的对象具有良好定义的接口，耦合度低。封装变化点，使用封装来创建对象之间的分界层，让设计者可以在分界层的一侧进行修改，而不会对另一侧产生不良的影响，从而实现层次间的松耦合。使用重构得到模式——设计模式的应用不宜先入为主，一上来就使用设计模式是对设计模式的最大误用。没有一步到位的设计模式。敏捷软件开发实践提倡的“Refactoring to Patterns”是目前普遍公认的最好的使用设计模式的方法。”

小A：“有没有更具体的原则？”

大B：“有啊，我和你说几条更具体的设计原则。单一职责原则（SRP）：一个类应该仅有一个引起它变化的原因。开放封闭原则（OCP）：类模块应该是可扩展的，但是不可修改（对扩展开放，对更改封闭）。

Liskov 替换原则（LSP）：子类必须能够替换它们的基类。依赖倒置原则（DIP）：高层模块不应该依赖于低层模块，二者都应该依赖于抽象。抽象不应该依赖于实现细节，实现细节应该依赖于抽象。接口隔离原则（ISP）：不应该强迫客户程序依赖于它们不用的方法。”

小A：“这样就好记多了。”

大B：“设计模式描述了软件设计过程中某一类常见问题的一般性的解决方案。面向对象设计模式描述了面向对象设计过程中、特定场景下、类与相互通信的对象之间常见的组织关系。深刻理解面向对象是学好设计模式的基础，掌握一定的面向

对象设计原则才能把握面向对象设计模式的精髓，从而实现灵活运用设计模式。”

小A：“嗯，我记住了。”

大B：“我再给你最后讲讲三大基本面向对象设计原则：1、针对接口编程，而不是针对实现编程。2、优先使用对象组合，而不是类继承。3、封装变化点，使用重构得到模式。敏捷软件开发实践提倡的“Refactoring to Patterns”是目前普遍公认的最好的使用设计模式的方法。”

■ Java常识

大B：“师弟，我来给你介绍一些JAVA的常识，这样对你以后学习JAVA有帮助。”

小A：“嘿嘿！好啊！”

大B：“你最好就好好地记住它。”

1、java的方法中的所有变量都必须初始化之后才能使用，否则无法通过编译，提示no initialize。而在方法外的变量则会被自动初始化，可以在该“{}”中使用。所有的变量都仅在自己声明的“{}”中起作用。

2、java的包分类：lang（构成语言的核心包）、awt（抽象图形工具包）、applet（已封装的applet小程序类）、io（基本的输入输出类）、net（与网络编程相关的类）、util（实用程序包，包括随机生成数字等）。

3、java中摒弃了C/C++中的指针与存储管理等应用，从而提高了程序的健壮

性，防止内存漏洞与存储器漏洞。

4、用加号 “+” 进行字符串连接。

5、java中的boolean类型不能与int类型进行转换。

6、java的条件控制语句 (if()) 中，括号中使用的是布尔表达式，而不是C/C++使用的数字值。因为java中布尔类型不能与数字类型转换，因而 “if(x)” 这种写法是错误的，应改为 “if(x!=0)” 。

7、switch()语句中的条件必须是与int类型是扶植兼容的，byte、short、char类型可以被升级，不允许使用浮点和long表达式。

8、注意label与break和continue等跳出语句的使用。break label/continue label，跳转到label出继续执行。

9、java中数组是一组同种数据类型的集合，是一种对象，声明是不分配内存空间，只创建了该对象的一个引用，数组元素的实际内存空间是通过new()方法进行初始化而动态分配的。数组声明的两种方法：char [] s 或者 char s []。

10、声明可以不指出数组的大小。

11、java支持多维数组，不但支持矩阵型数组，而且支持非矩阵型数组。

12、java中具有数组拷贝函数 (System.arraycopy()) 。

13、子类从超类 (父类) 继承所有方法和变量；但子类不从超类继承构造函数；包含构造函数的两个办法是a、使用缺省构造函数，b、写一个或多个显式构造函数。

14、多态性是个运行时问题，与重载相反，重载是一个编译时问题。

15、关键字`super` 可被用来引用该类中的超类。它被用来引用超类的成员变量或方法，可以使用`super.method()`的格式来调用。

16、`instanceof`的使用，在对象强制类型转换时常常使用。

17、在一个类中可以通过参数个数/参数类型不同，从而构造重载函数。在子类与父类之间，可以在子类中定义与父类具有一样函数名称、参数个数、参数类型、返回类型的函数，从而达到函数覆盖的作用。

18、通过子类的数据初始化父类的成员变量，可以在子类的构造函数中使用`super()`来调用父类的构造函数初始化父类中的成员变量。

19、`import`语句必须先于所有类的声明。`import` 语句被用来将其它包中的类带到当前名空间。当前包，不管是显式的还是隐含的，总是当前名空间的一部分。

20、类中的`static`变量可以被该类的所有实例共享，如果声明为`private`，只有该类的实例才能访问，如果声明为`public`，可以不通过该类的实例，直接在类体外通过类名就可以访问。

21、类中的`static`方法可以不通过该类的实例，直接在类体外通过类名就可以访问，`static` 方法不能访问与它本身的参数以及`static` 变量分离的任何变量。访问非静态变量的尝试会引起编译错误。没有`this` 值。

22、静态方法不能被覆盖成非静态。

23、`static block`。 “`static {}`” 。

24、final 类不能被分成子类；final 方法不能被覆盖；final 变量是常数。被标记为static 或private 的方法被自动地final，因为动态联编在上述两种情况下都不能应用。如果变量被标记为final，其结果是使它成为常数。想改变final 变量的值会导致一个编译错误。

25、声明方法的存在而不去实现它的类被叫做抽象类。不能有抽象构造函数或抽象静态方法。Abstract 类的子类为它们父类中的所有抽象方法提供实现，否则它们也是抽象类。

26、接口是抽象类的变体。在接口中，所有方法都是抽象的。多继承性可通过实现这样的接口而获得。接口中的所有方法都是抽象的，没有一个有程序体。接口只可以定义static final成员变量。

27、内部类不能声明任何static 成员；只有顶层类可以声明static 成员。

28、finally 语句定义一个总是执行的代码块，而不考虑异常是否被捕获。如果终止程序的System.exit()方法在保护码内被执行，那么，这是finally 语句不被执行的唯一情况。这就暗示，控制流程能偏离正常执行顺序，比如，如果一个return 语句被嵌入try 块内的代码中，那么，finally 块中的代码应在return 前执行。

29、java中类的成员变量是在对象实例化之后才分配内存空间，类变量则在类加载的时候分配空间，该类以及该类的实例对象都共享类变量。类的成员方法是在类的第一个对象实例化的时候才分配入口地址的，当再创建对象时，不再分配入口地址，就是说，所有对象共享一个入口地址，而类方法则是在类被加载到内存的时候分配入口地址的，因而可以被类以及类的实例所调用。

30、this指向本类，可以通过成员访问运算符"."访问类的成员变量或方法，但this不能出现在类方法中，因为类方法是可以直接通过类名调用。

31、a、public b、protected c、友好的(无修饰符) d、private，private在子类中无法继承父类的成员变量与成员方法，在同包中，子类可继承父类的a b c类型的变量与方法，不同包中，子类只能继承a b类型的变量与方法。

32、当子类与父类之间如果存在同名的成员变量时，则父类的成员变量被隐藏，当子类与父类存在返回类型、参数类型以及个数都相同的函数时，父类的成员方法被隐藏，因此可以通过重写子类的成员函数而将父类的状态和行为改变为自身的状态与行为。但子类重写方法时，访问权限不能低于父类的修饰符。

33、如果一个方法被修饰为final方法，则这个方法不能被重写，如果一个成员变量被修饰为final的，就是常量。

34、对象的上转型对象，A是B的父类，A a=new B(),则a是b的上转型对象，上转型对象不能操作子类新增的成员变量，但可以操作子类继承或重写的成员变量，也可以使用子类继承的或重写的方法，不可以将父类创建的对象引用赋值给子类声明的对象。

35、接口用interface声明，用于java多继承，接口体中包含常量定义和方法定义两部分，接口体中只进行方法的声明，不许提供方法的实现，所以方法的定义没有方法体，且用分号。

36、如果一个类使用了某个接口，那么这个类必须实现该接口的所有方法。如果一个类声明实现一个接口，但没有实现接口中的所有方法，那么这个类必须是abstract类。

37、接口回调:可以把实现某一接口的类创建的对象引用赋值给该接口声明的接口变量中，那么该接口变量就可以调用被类实现的接口中的方法。当接口变量调用被类实现的接口的方法时，就是通知相应的对象调用接口的方法。

38、string类型变基本类型，可以用public int Integer.parseInt(string s) 其他基本类型相似，基本类型变string类型，可以用string.valueOf(int/char/float/double),对象类型用date.toString() 转换。

39、stringtokenizer类可以实现字符串的分析，stringtokenizer objectname=new stringtokenizer(s," 分隔符列表 用空格间隔"),重要方法：counttoken()、nexttoken()、hasmoretokens()。

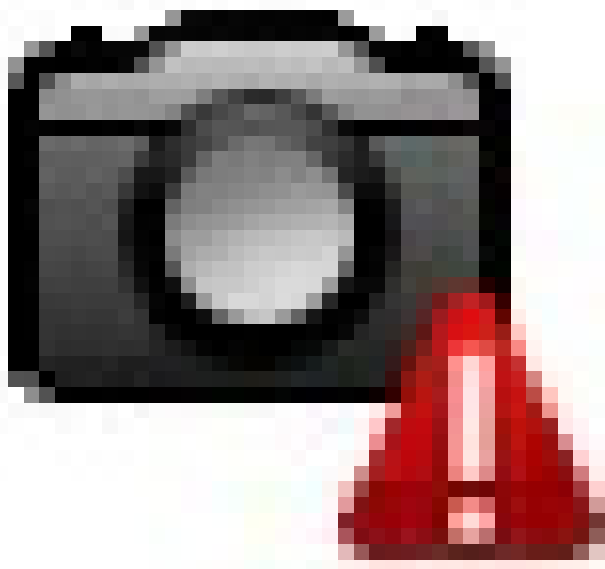
40、charactor类重要方法：isDight(char) isLetter(char) isLetterOrDight(char) isLowerCase(char) isUpperCase(char) toLowerCase(char) toUpperCase(char) isSpacechar(char) 。

41、将字符串转变成字符数组，可以使用string类的String.toCharArray() 方法，char a []=String.toCharArra()

42、string(char [],int offset,int length) 返回一个string对象,getChars(int start,int end,char c [],int offset) 功能：字符串变字符数组，由string对象使用。

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质

电子书下载！！！！



本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质
电子书下载！！！！

Table of Contents

版权信息

前言

第一部分 设计模式概述

第一章 大学毕业了怎么办？——设计模式概述

第二部分 接口型模式

第二章 学校招聘会——接口型模式介绍

第三章 我们班来了位新同学——适配器模式

第四章 金融危机股票还挣钱？——外观模式

第五章 生日礼物——组合模式

第六章 蜡笔与毛笔——桥接模式

第三部分 责任型模式

第七章 击鼓传花——责任型模式

第八章 购物车——单体模式

第九章 放风者与偷窃者——观察者模式

第十章 中介公司——中介者模式

第十一章 高老庄的故事——代理模式

第十二章 包子——享元模式

第四部分 构造型模式

第十三章 可恶的皇帝——构造型模式

第十四章 汽车组装——生成器模式

第十五章 运动协会——工厂方法模式

[第十六章 麦当劳的鸡腿——抽象工厂模式](#)

[第十七章 兰州拉面馆——原型模式](#)

[第十八章 月光宝盒——备忘录模式](#)

[第五部分 操作型模式](#)

[第十九章 儿子的功课——操作型模式](#)

[第二十章 订单处理——模板方法模式](#)

[第二十一章 金融危机何时休——状态模式](#)

[第二十二章 还钱——策略模式](#)

[第二十三章 饭店点菜——命令模式](#)

[第二十四章 苹果汁——解释器模式](#)

[第六部分 扩展型模式](#)

[第二十五章 多功能的手机——扩展型模式](#)

[第二十六章 三明治——装饰器模式](#)

[第二十七章 老公，有钱不？——迭代器模式](#)

[第二十八章 指挥工人工作——访问者模式](#)

[第二十九章 大学生毕业3条出路：学、仕、商——设计模式总结](#)

[附录：面向对象基础](#)

本书由「ePUBw.COM」整理，ePUBw.COM 提供最新最全的优质电子书下载！！！！