C#设计模式

C# Design Patterns: A Tutorial

[美] James W. Cooper 著
张志华 刘云鹏 等译

電子工業出版社
Publishing House of Electronics Industry
http://www.phei.com.cn

C#设计模式

C# Design Patterns: A Tutorial

设计模式是针对日常软件开发问题的巧妙、通用和可复用的解决方案。程序员可以通过设计模式组织程序中的对象，使其易于编写和修改，以提高代码效率。本书正是一本指导读者采用最常用的模式来编写C#程序的实用手册。

该书首先简明扼要地介绍了C#、面向对象程序设计和UML图；然后分别说明了23种设计模式以及何时使用这些模式，并阐明了模式在大型设计中的作用。每种模式的应用都用简单的示例程序加以论证，这些示例程序包含在随书附带的光盘中，可以直接运行、测试、编辑和使用。

采用下列方法后，设计模式将对你的工作产生非常有益的影响。此外，设计模式不仅可以提高工作效率，而且会成为C#程序设计中不可或缺的组成部分：

- 将设计模式有效地应用于日常程序设计中
- 用模式创建复杂、强壮的C#程序
- 采用UML图说明类之间的关系
- 使用设计模式提高程序设计能力

James W. Cooper

IBM Thomas J. Watson研究中心信息管理和检索领域的研究人员，拥有多项专利，发表过大量科技论文，并编著了日本书。先后著有"Visual Basic Design Patterns"（Addison-Wesley，2002）和"Java Design Patterns"（Addison-Wesley，2000），同时也是"JavaPro"杂志的固定专栏作家。

ISBN 7-5053-8979-3

9 787505 389793 >

责任编辑：熊淑韵
封面设计：毛惠庚

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书

ISBN 7-5053-8979-3/TP・5212　定价：33.00元（含光盘）

C# 设计模式

C# Design Patterns：A Tutorial

[美] James W. Cooper 著

张志华 刘云鹏 等译

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

内容简介

本书主要介绍如何用最常见的设计模式编写C#程序。全书分为四个部分，首先介绍了C#语言和面向对象程序设计的一般原则，可作为C#程序设计的快速入门教程；然后分别讲述了创建型模式、结构型模式和行为型模式。每一类设计模式又包括若干种具体模式，共有23种。在介绍每种模式时，给出了一个或多个应用该模式的示例，以便于理解，且这些示例都是能完全运行的程序，包含在随书附带的光盘中。此外，每一章还提供了UML图，用以说明类之间的关系。

本书适用于计算机及相关专业的本科生和研究生，对于软件开发人员也是一本很好的参考书。

版权贸易合同登记号：图字：01-2002-6038

图书在版编目（CIP）数据

C#设计模式/（美）库珀（Cooper, J. W.）著；张志华等译. - 北京：电子工业出版社，2003.8
书名原文：C# Design Patterns: A Tutorial
ISBN 7-5053-8979-3

Ⅰ.C… Ⅱ.①库… ②张… Ⅲ.C语言－程序设计 Ⅳ.TP312

中国版本图书馆CIP数据核字（2003）第067673号

责任编辑：陶淑毅
印 刷 者：北京兴华印刷厂
出版发行：电子工业出版社 http://www.phei.com.cn
北京市海淀区万寿路173信箱 邮编：100036
经 销：各地新华书店
开 本：787 × 1092 1/16 印张：16.75 字数：482千字 附光盘一张
版 次：2003年8月第1版 2003年8月第1次印刷
定 价：33.00元（含光盘）

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系电话：（010）68279077

译 者 序

本书介绍了C#设计模式，其中C#是Microsoft公司推出的新一代软件开发环境Microsoft.NET的核心语言，它的出现代表了程序设计语言下一步的发展方向；而设计模式是系统地获取一些软件开发专家的成功经验，提供“在一定的环境中解决某一问题的方案”。本书主要讲述的是，如何使用最常见的设计模式编写C#程序，以提高程序的可读性、可重用性和可维护性。

本书分为四个部分：第一部分介绍了C#语言和面向对象程序设计的一般原则；第二部分讲解了创建型模式（Creational Pattern），这一模式规定了创建对象的方式；第三部分介绍了结构型模式（Structural Pattern），该模式规定了如何组织类和对象；第四部分讲述了行为型模式（Behavioral Pattern），这一模式规定了对象之间交互的方式。全书一共介绍了23种设计模式，从第8章到第31章每章讨论一种模式，并给出一个或多个应用该模式的示例，这些示例都是能完全运行的程序。另外，每一章还包括一个UML图，用以说明类之间的关系。

本书的第一部分可作为C#程序设计的快速入门教程，另外在这一部分还介绍了UML，可供不熟悉UML的读者参考。对于第二部分至第四部分介绍的各种类型的设计模式，读者可以根据需要有选择地阅读。学习设计模式时，不要急于求成，对于某些难以理解的模式，多研究一下示例程序会很有帮助。

本书用C#示例程序讲解抽象的设计模式，使读者理解起来更容易。无论是有经验的软件开发人员，还是初学者，这都是一本理想的教材和参考用书。

全书主要由张志华、刘云鹏翻译，其他参与本书翻译工作的同志还有施运梅、管萍、李文虎、沈志刚、李树玲、刘广、张驰等。在此，也由衷地感谢所有支持我们、给我们提供帮助的人。

由于译者水平有限，难免有不当之处，恳请广大读者批评指正。

前　言

这是一本很实用的书，告诉读者如何用最常见的设计模式编写C#程序，同时本书也可以作为C#程序设计的快速入门教程。设计模式分成若干较短的章节来讨论，每章讨论一种设计模式，并给出一个或多个应用该模式的示例，这些示例都是能完全运行的可视化程序。每一章还包括了用以说明类之间关系的UML图。

本书并非“Gang of Four”所著的很有影响的“Design Patterns”（《设计模式》）一书的姊妹篇。而对于那些想要了解设计模式是什么以及如何把设计模式应用于工作中的人来说，此书却是一本不错的指南。读者没有必要先阅读“Design Patterns”，再阅读本书，但是，如果你事先读过那本书，也完全可以再读一读本书以获得更多的知识。

在本书中，读者会发现设计模式经常用在程序中对象的组织方式上，以使程序更加容易编写和修改。另外，通过熟悉设计模式，也会掌握一些用于讨论程序如何构建的有用词汇。

人们从不同方面开始欣赏设计模式，从高深的理论研究到日常的实践操作。最终，人们看到了这些模式的巨大力量，体会到了使用模式带来的诸多便利，于是设计模式融入到了工作中的方方面面。

全书以尽可能多的方式来描述模式，帮助读者理解抽象的概念。该书分成如下几个主要部分：概述，C#介绍和模式的讨论（这一部分又分创建型模式、结构型模式和行为型模式三部分内容）。

对于每一种模式，先给出简洁的描述，然后构建几个简单的程序示例。每一个示例都是可视化的程序，读者可以运行、检验程序，以使该模式的概念尽可能地具体化。所有的程序示例及其变化形式都在随书附带的光盘中，读者可以运行、修改这些程序，并查看改动所产生的效果。

示例中用到的每一个类都放在C#文件中，由于一个示例由多个C#文件组成，因此我们为每个示例提供一个C#项目文件，并把每个示例都放在一个单独的子目录中，从而避免了混乱。本书假设读者具备Visual Studio.NET环境，这个软件有各种版本，这里的示例代码是使用专业版开发的。

书中还穿插了许多用来说明设计模式的程序片段，帮助读者进一步理解这些模式。另外，还附加了这些程序的UML图，以说明类之间的关系。UML图说明了类及其继承结构，由框和箭头组成，其中箭头指向父类，后接虚线的箭头指向接口。如果读者还不熟悉UML，可以参看第6章提供的UML的简单介绍。书中所有的图表都是用WithClass 2000制作的，该软件的演示版包含在随书附带的光盘中。

学习完本书后，读者会熟练掌握设计模式的基础知识，并能将其运用于日常的C#程序设计工作中。

目　录

第一部分　C# 面向对象程序设计

第二部分 创建型模式

第三部分 结构型模式

第四部分 行为型模式

第一部分

C#面向对象程序设计

本书的第一部分介绍了设计模式的概念、C#语言和面向对象程序设计的一般原则。具体讲解了C#语法和如何创建简单的程序，并讨论了类、对象、继承、接口及UML图的基础知识。学习完这一部分后，可以熟悉C#和面向对象程序设计，接下来就可以学习如何将设计模式运用于C#程序设计中了。

第1章　设计模式概述

也许读者正坐在电脑桌旁，凝视着天花板，勾画着如何编写一个新程序的功能部件。凭直觉，你也许知道必须做什么，要用到哪些数据和对象，但是潜意识里总觉得，会有一种更好的、更通用的方式来编写这个程序。

实际上，直到读者在脑子里建立了一个代码要完成什么操作和代码段之间如何相互作用的框架后，才有可能进行编码。框架建立得越完善，对问题的解决方案会越满意。如果没有掌握这种正确的方法，那么即使问题的解决方案很明显，恐怕也还要茫然一段时间。

我们凭感觉就知道，越是一流的解决方案就越容易复用，也越容易维护。即使读者是惟一可靠的程序员，一旦设计完一个相对优雅、没有暴露太多内部异常的解决方案后，也会觉得很安心。

计算机科学的研究人员开始欣赏设计模式的主要原因之一是，设计模式能够使解决方案既优雅简单，又可复用。术语“设计模式”对缺乏专业知识和经验的人来说有点儿正式，第一次遇到时会有些无所适从。事实上，设计模式仅仅是一些在项目之间和程序员之间复用面向对象代码的简便方法。设计模式背后的思想很简单：对通用的对象间的相互作用方式进行记录和编目（程序员经常发现这些对象间的相互作用方式很有用）。

一个常被引用的模式是Smalltalk（Krasner and Pope 1988）的模型－视图－控制器（Model-View-Controller）框架，它出自关于程序设计框架的早期文献。这个模式把用户接口问题分成三个部分，如图1.1所示。图中“数据模型”包括程序的计算部分，“视图”表示用户界面，“控制器”定义用户和视图的交互方式。

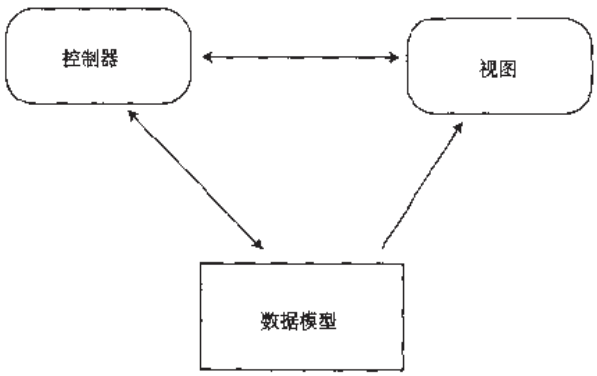


图1.1　模型－视图－控制器框架

其中的每个部分都是一个独立的对象，每个对象用自己的规则处理数据。用户、GUI（图形用户界面）和数据间的通信应该认真控制，而这种功能的分离恰恰能很好地满足这一点。三个对象使用这样一组约束的连接相互通信，这就是一个功能强大的设计模式示例。

换言之，设计模式描述了对象如何进行通信才能不牵涉相互的数据模型和方法。保持这种独立性一直是一个好的面向对象（Object Oriented, OO）程序设计的目标，如果读者一直在努力使对象只涉及各自的业务，那么你很可能已经在使用某些通用的设计模式了。

设计模式开始变得越来越流行是在20世纪90年代初期，当时Eric Gamma（1992）把模式合并在了GUI应用程序框架ET++中。各种讨论和学术会议发展的巅峰是一本喻为“设计模式鼻祖”的

书的出版，即由 Gamma 等人（1995）创作的“Design Patterns：Elements of Reusable Software”。这本书通常被称为“Gang of Four”或“GoF”，它对想弄清楚如何使用设计模式的人产生了深远的影响，并成为销量空前的畅销书。书中描述了经常遇到又通用的23种模式，并对如何使用和何时使用这些模式加以论述。我们在本书中把这本具有开创性的书称为“Design Patterns”（《设计模式》）。

自从“Design Patterns”一书出版以来，相继又有许多其他有价值的书出版。其中一本关系较密切的书是“The Design Patterns Smalltalk Companion”（Alpert，Brown，and Woolf 1998），它以Smalltalk的观点介绍了同样的23种模式。我们在本书中称之为“Smalltalk Companion”。最近又出版了“Java Design Patterns：A Tutorial”和“Visual Basic Design Patterns：VB6 and VB.NET”，这两本书使用各自的语言阐明了同样的23种模式（Cooper 2000，2002）。

1.1　定义设计模式

我们总在谈论做各种事情的方法：工作、爱好和家庭生活等各个方面，而且在不断地重复一些模式。

- 发粘的小圆面包与晚餐上的小圆面包看起来很像，但我在上面加了红糖和坚果。
- 她门前的花园和我的很像，但我在花园里种了玫瑰。
- 这张茶几和那张样式上很像，但这张茶几用门代替了抽屉。

在程序设计中也会有同样的事情：告诉一个同事我们怎样完成了一个包含相当多技巧的程序，这样他就不用从头开始再做一次。通过维持对象的独立性，我们找到了对象通信的有效方式。

随着该领域文献的增加，出现了一些有用的设计模式定义。

- 设计模式是对读者经常遇到的设计问题的可再现的解决方案（The Smalltalk Companion）。
- 设计模式建立了一系列描述如何完成软件开发领域中特定任务的规则（Pree 1995）。
- 设计模式更关注于复用可重复出现的结构设计方案，而框架注重于具体设计和实现（Coplien and Schmidt 1995）。
- 模式提出了一个发生在特定设计环境中的可重复出现的设计问题，并提供了解决方案（Buschmann et al. 1996）。
- 模式识别并确定类和实例层次上或组件层次上的抽象关系（Gamma，Johnson，and Vlissides，1993）。

尽管找到与建筑学、家具制造和逻辑学的相似之处是很有帮助的，但设计模式不仅仅局限于对象的设计，还涉及到对象之间的相互作用。有一种观点就是将某些模式看做通信模式。

另外一些模式不但涉及到对象通信，还涉及到针对对象继承和包含的策略。正是简单而优雅的设计及相互作用的方式使众多的设计模式变得如此重要。

设计模式可存在于多个层次中，从最底层的专门解决方案到很普遍的系统问题。目前已提出了数百个模式，在文献中和各级会议上都有讨论。某些模式应用很广，也有些作者将模式行为归成几类，每类适用一种问题（Kurata 1998）。

很明显，读者不需要记住这些模式。实际上，大部分模式是被发现而不是被设计出来的。寻找这些模式的过程称为“模式挖掘”，它本身就值得写一本书。

“Design Patterns”一书中包含的23种设计模式都具有几个已知的应用，并且有一定的普遍性，很容易跨越应用领域解决不同的问题。

本书作者把这些模式分成三类：创建型、结构型和行为型。

- 创建型模式是创建对象而不是直接实例化对象，这会使程序在判断给定情况下创建哪一个对象时更为灵活。
- 结构型模式可以将一组对象组合成更大的结构，例如复杂的用户界面或报表数据。
- 行为型模式定义系统内对象间的通信，以及复杂程序中的流程控制。

在后面的章节里，会看到用C#描述的这些模式，对23种模式中的每一种，我们都至少提供一个完整的C#程序。读者可以分析提供的代码片段，也可以运行、编辑和修改所附光盘上的完整程序。在每一种模式描述完之后，都会给出光盘中相应程序的列表。

1.2 学习过程

我们发现，不管使用哪种语言，学习设计模式都需要三个阶段：

1. 接受。
2. 认可。
3. 领会。

首先，你要接受设计模式对自己的工作很重要这一前提；接下来，会认识到需要学习设计模式，这样才能知道什么时候需要使用设计模式；最后，要充分消化有关模式的细节，这样才能知道哪一种模式能帮助自己解决给定的问题。

对某些幸运的人来说，设计模式是显而易见的工具，这些人只需阅读模式的概要部分就能抓住它们的本质。而对于大多数像我们这样的人来说，了解一种模式后会有一个较慢的感悟阶段，然后当看到能将这些模式用在工作中时，才会恍然大悟。本书通过提供一系列完整的、可运行且读者可以进行试验的程序，带你走到最后一步的领会阶段。

“Design Patterns”中的示例很简洁，多采用C++描述，有时候是用Smalltalk描述。如果选用另外一种语言工作，那么采用选择的语言来描述模式的示例也很有帮助。本书力图满足C#程序员的这一需求。

1.3 学习设计模式

熟悉设计模式有几种不同的方法。在每种方法中，都应该按顺序阅读本书和“Design Patterns”一书，或者以相反顺序阅读。为了完整性，我们还强烈推荐读者阅读“Smalltalk Companion”一书，因为它对每种模式都有不同的描述。并且，还有许多学习和讨论设计模式的网站，读者可以去浏览学习。

1.4 关于面向对象方法

使用设计模式的根本原因是为了保持类之间的隔离，防止相互之间了解太多的内容；另一个重要的原因是，使用这些模式可以帮助读者避免重复开发，能用其他程序员容易理解的术语描述你的程序设计方法。

有很多策略能让OO程序员达到分离类的目的，其中包括封装和继承。几乎所有具备OO能力的语言都支持继承。一个继承父类的类能访问父类中的所有方法和所有非私有变量，然而，如

果以一个功能完整的类开始继承层次结构，可能会过多地限制住自己，也会给特定方法的实现带去累赘。因此，“Design Patterns”建议应遵循下列设计原则：针对接口编程，而不是针对实现编程。

用更简洁的形式来表示这一原则，就是在任何类层次结构的顶端，定义的是一个抽象类或一个接口，它没有实现方法，但定义了该类需要支持的方法。这样在所有的派生类中，就会有更大的自由度去实现这些方法，能最大限度地满足你的要求。

另外一个需要考虑的重要概念是对象组合，这是一种可包含其他对象的对象结构，即把几个对象封装在另一个对象中。许多初级OO程序员使用继承去解决每个问题，而随着你开始编写比较复杂的程序，就会体会到对象组合的优点。新的对象拥有一个最适合于要完成的目标的接口，而不是拥有父类中的所有方法，这就导出了“Design Patterns”一书提出的第二个重要原则：优先使用对象组合，而不是继承。

1.5　C# 设计模式

本书讨论了“Design Patterns”一书提出的23种模式，每种模式至少给出了一个可运行的程序示例。所有的程序都有可视化界面，以使程序理解起来更容易，并且都使用了类、接口和对象组合，但程序本身保持了必要的简单性，这样编码就不会掩盖所描述模式的优雅特性。

尽管C#是我们的目标语言，但这并非一本专门讲解C#语言的书。有许多的C#特性书中都没有涉及，但本书覆盖了C#的大部分重要内容。读者会发现，对于采用C#进行面向对象程序设计的人来说，这是一本相当有用的指导书，书中对如何使用C#进行编程也给出了很好的概述。

1.6　本书的组织结构

第1章到第7章介绍了设计模式的概念、C#语言的基础知识，以及面向对象程序设计和继承的思想，还讲解了如何用UML类图表示对象。

然后研究了23种设计模式，将它们分类进行编目：创建型模式、结构型模式和行为型模式。许多模式都是在某种程度上独立存在的，但我们也会不时地用到已经讨论过的模式。例如，在介绍完工厂和命令模式后，会广泛地使用这两种模式；在介绍了中介者模式后也会多次使用该模式。我们还将备忘录模式用到了状态模式中；将职责链模式用到了解释器模式的讨论中；将单件模式用到了享元模式的讲解中，但我们在正式介绍某种模式之前决不会去使用该模式。

本书利用当前讨论的模式来介绍C#的新特性。例如，我们在适配器模式和桥接模式中介绍了ListBox，DataGrid和TreeView；在抽象工厂中讲解了如何绘制图形对象；在解释器模式和组合模式里介绍了Enumeration界面，并研究了格式转换；在单件模式中用到了异常；在外观模式中讨论了ADO.NET数据库连接；在代理模式中，教授读者如何使用C#计时器。

整个.NET系统的设计目标是，使基于Web的客户－服务器交互操作更加灵活，而本书则关注于一般的面向对象程序设计问题，而不是如何编写基于Web的系统。我们涵盖了C#程序设计的核心内容，并给出了一些简单的示例，说明如何在设计模式帮助下编写更好的程序。

第2章　C#语言的语法

C#具有功能强大的现代语言的所有特性。如果读者熟悉Java，C或C++，就会发现多数C#的语法都是非常熟悉的。如果使用过Visual Basic或类似语言，那么应该阅读本章来看看C#与VB的区别，你会很快发现在VB.NET上执行的主要操作都和C#上的相似。

C#和VB的两个主要区别是：(1) C#是区别大小写的（大多数语法使用小写）；(2) C#中的每一条语句都以分号（;）结束，这样，C#语句就不必限制在一行内，而且不需要续行符。

在VB中，我们可以这样写：

```
y = m * x + b       'compute y for given x
```

也可以这样写：

```
Y = M *X + b       'compute y for given x
```

可以将它们看成是一样的。不管大写还是小写，对于变量Y，M和X都是一样的。然而，在C#中，大小写至关重要，如果我们这样写：

```
y = m * x + b;         // all  lowercase
```

或

```
Y = m * x + b;        // Y differs from y
```

则意味着Y和y是两个不同的变量。尽管刚开始看来这似乎有些笨拙，但以后会发现区别大小写是很有用的。例如，程序员经常用大写的符号表示常数：

```
Const PI = 3.1416  As Single in VB
const float PI = 3.1416;     //in C#
```

C#中的修饰符const的含义是：被命名的值是一个常数，不能更改。

程序员有时也用大小写混合的形式定义数据类型，用小写形式定义这种数据类型的变量。

```
class Temperature {     // begin definition of
                        // new data type
Temperature temp;       // temp is of this new type
```

我们在后面的章节中会更具体地介绍类的知识。

2.1　数据类型

表2.1给出了C#的主要数据类型。注意，基本类型的长度与计算机或操作系统的类型无关。C#中的字符是16位的宽度，可以表示非拉丁语言中的所有字符。它使用一种叫双字节码的字符编码系统，其中定义了绝大多数可书写语言的数以千计的字符。可以使用通常的方式在不同的变量类型之间转换。

表2.1 C#中的数据类型

bool	true或false
byte	8位无符号整数
short	16位整数
int	32位整数
long	64位整数
float	32位浮点数
double	64位浮点数
char	16位字符
string	多个16位字符

- 宽度窄(字节数少)的数据类型可以直接赋给较宽的数据类型，并能自动转换为新类型。如果y是浮点类型而j是整数类型，则可以用下列方式把一个整数类型转换成浮点类型。

```
float y = 7.0f;          //y is of type float
int j = 5;               //j is of type int
y = j;                   //convert int to float
```

- 可以通过强制转换把较宽的类型(字节较多)缩减成较窄的类型。这时，需要把数据类型的名称放在圆括号内，并把它放在要转换的数据前面。

```
j = (int)y;          //convert float to integer
```

也可以写出这样的合法语句，它包含的强制转换可能会失败。

```
float x = 1.0E45;
int k = (int) x;
```

如果强制转换失败，程序执行时会出现一个异常错误。

布尔型变量只能接受保留字true和false所表示的值。布尔型变量通常接受比较操作和其他逻辑操作的结果作为它的值。

```
int k;
boolean gtnum;

gtnum = (k > 6 );          //true if k is greater than 6
```

与C或C++不同，不能将数值型数据赋值给布尔型变量，也不能在布尔类型和其他数据类型之间转换。

2.2 数值与字符串间的转换

可以使用Convert方法将数值转换成字符串或将字符串转换成数值。在开发环境中只要键入“Convert”和一个圆点，系统就会提供一系列可用的方法，在其中能找到正确的那一个。

```
string s = Convert.ToString (x);
float y = Convert.ToSingle (s);
```

注意，“Single”代表一个单精度浮点数。

数值型对象也提供了各种格式化方法来指定小数位数。

```
float x = 12.341514325f;
string s = x.ToString ("###.###");        //gives 12.342
```

2.3 声明多个变量

注意，C# 允许在一条语句里声明同一类型的多个变量。

```
int i, j;
float x, y, z;
```

这一点与 VB6 不同，VB6 在声明变量时，必须指定每一个变量的类型。

```
Dim i As Integer, j As Integer
Dim x As Single, y As Single, z As Single
```

然而，很多程序员发现，一行语句声明一个变量会更清楚，这样才能逐一地注释每个变量。

```
int i;          // used as an outer loop index
int j;          // the index for the y variables
float x;        // the ordinate variable
```

2.4 数值型常量

程序中键入的任何数如果没有小数部分的话，就自动为整数类型，如果有小数部分，就自动为 double 类型。如果想指定成不同的类型，可以使用各种前缀和后缀字符。

```
float loan = 1.23f;          // float
long pig   = 45L;            // long
int color  = 0x12345;        // hexadecimal
```

C# 有三个保留字常量：true，false 和 null，其中 null 表示一个对象变量还没有指向任何对象。我们在后面的章节中会学到关于变量的更多知识。

2.5 字符常量

用单引号把字符括起来表示字符常量。

```
char c = 'q';
```

C# 遵循 C 的约定，空白字符（引起打印位置改变的非打印字符）用特殊字符前面加一个反斜杠表示，如表 2.2 所示。由于这种情况下反斜杠本身是一个特殊字符，所以用两个反斜杠来表示它。

表 2.2 空白字符和特殊字符的表示

\n	换行
\r	回车
\t	制表符
\b	退格
\f	换页
\0	空符号
\"	双引号
\'	单引号
\\	反斜杠

2.6 变量

C# 的变量名长度是任意的，可以是数字和大小写字母的混合形式，但是同 VB 一样，第一个字符必须是字母。因为 C# 是区分大小写的，所以下列变量名代表不同的变量。

```
temperature
Temperature
TEMPERATURE
```

C#程序中的任何变量在使用前，必须进行声明。

```
int j;
float temperature;
boolean quit;
```

2.6.1 用到变量时再声明

C#允许在需要使用变量的时候再进行声明，而不必在过程的顶端声明变量。

```
int k = 5;
float x = k + 3 * y;
```

这在面向对象的程序设计中非常普遍，比如我们可以在程序的循环内声明一个变量，在作用域之外该变量就不再存在了。

2.7 用多个等号初始化

C#像C一样，可以在一条语句里将多个变量初始化为一个值。

```
i = j = k = 0;
```

这可能会有点混乱，所以不要过度使用这一特性。

```
i = 0; j = 0; k = 0;
```

不管语句在一行里（如上所示）还是连续的几行里，编译程序产生的代码都是一样的。

2.8 一个简单的C#程序

现在我们来看一个非常简单的C#程序，它把两个数加在了一起。这是一个独立的程序。

```
using System;
class add2
{
        static void Main(string[] args)
        {
            double a, b, c;  //declare variables
            a = 1.75;        //assign values
            b = 3.46;
            c = a + b;       //add together
            //print out sum
            Console.writeLine("sum = " + c);
        }
}
```

正如它所表示的，这是一个完整的程序，若用C#编译器编译并运行，将会打印出如下结果：

```
sum = 5.21
```

接下来要看看由这个简单的程序能得出什么结论：

1. 必须使用using语句定义程序中要用到的C#代码库，这一点和VB中的imports语句及Java中的import语句类似。它也类似于C及C++中的#include指令。

2. 程序以一个称为 Main 的函数开始，它必须完全符合下列形式：

```
static void Main(string[] args)
```

3. 每一个程序模块都要包含一个或多个类。
4. 类和类中的函数必须放在花括号（{}）内。
5. 每个变量在使用之前必须声明类型，也可以按照下列方式进行声明：

```
double a = 1.75;
double b = 3.46;
double c = a + b;
```

6. 每条语句必须以分号结束。语句可以持续几行，但必须以分号结束。
7. 注释以 // 开头，在行尾结束。
8. 同其他大多数语言（除 Pascal 外）一样，等号表示数据赋值。
9. 用加号(+)连接两个字符串，串“sum = ”和由双精度变量 c 自动转换成的串连接在一起。
10. writeLine 函数是 System 名字空间中的 Console 类的一个成员，用于向屏幕输出数据。

这个简单程序的名字是 add2.cs，可以在开发环境中按 F5 键编译执行它。

2.9 算术运算符

C# 中的基本运算符非常类似于其他大多数现代语言。表 2.3 列出了 C# 中的基本运算符。

表 2.3 C# 的算术运算符

+	加
-	减、一元减
*	乘
/	除
%	求余（整除后的余数）

位运算符和逻辑运算符来源于C语言（见表2.4）。位运算符对两个字的单个位进行操作，根据 AND，OR 或 NOT 操作产生结果。位运算符和布尔运算符不同，因为布尔运算符对逻辑条件进行操作，求得结果为 true 或者 false。

表 2.4 C# 的逻辑运算符

&	按位与
|	按位或
^	按位异或
~	取反
>> n	右移 n 位
<< n	左移 n 位

2.10 递增运算符和递减运算符

同 Java 和 C/C++ 一样，C# 可以用 ++ 和 -- 运算符表示整型变量的加 1 和减 1 操作。可以将这些运算符应用在变量的使用前或使用后。

```
i = 5;
j = 10;
x = i++;        //x = 5,then i = 6
y = --j;        //y = 9 and j = 9
z = ++i;        //z = 7 and i =7
```

2.11 将算术语句和赋值语句合并

C#可以将加法、减法、乘法和除法与计算结果的变量赋值合并在一起。

```
x = x + 3;          // can also be written as:
x += 3;             // add 3 to x; store result in x
// also with the other basic operations;
temp *= 1.80;       // mult temp by 1.80
z -= 7;             // subtract 7 from z
y /= 1.3;           // divide y by 1.3
```

这样做的主要目的是为了减少输入，而不是为了生成不同的代码。当然，这些复合运算符（包括++和--）之间不能有空格。

2.12 C#中的判断

C#中也有类似于Visual Basic，Pascal和FORTRAN的if-then-else语句，但在C#中不使用then关键字。

```
if ( y > 0 )
  z = x / y;
```

C#要求条件外面要有圆括号，这种格式会引起些麻烦。正如语句所描述的那样，if语句只对它后面的一条语句起作用，如果想让多条语句都作为条件的一部分，必须把它们放在花括号里。

```
if  ( y > 0 )
   {
   z = x / y;
   Console.writeLine("z = " + z);
   }
```

相反，如果写成：

```
if ( y > 0 )
   z = x / y;
   Console.writeLine("z = " + z);
```

因为if语句只对紧跟在后面的一条语句起作用，所以C#程序会始终打印输出“z=”及某个值。正如读者所看到的，缩进对程序没有影响，它只表明你要说的内容，而不能表示你的实际本意。

由于这一原因，我们通常建议把所有的if语句都放在花括号里，即使只有一行。

```
if ( y > 0 )
{
   z = x / y;
}
```

如果想根据条件结果来执行一组语句或另一组语句，应该在if语句后面使用else语句。

```
if( y > 0 )
{
   z = x / y;
}
else
{
 z = 0;
}
```

正如前面的代码所表明的那样，如果else语句包含多条语句，必须把它们放在花括号里。

C#程序有两种花括号的缩进方式，这是一种方式：

```
if ( y > 0 )
      {
   z = x / y;
      }
```

另外一种缩进方式在C程序员中比较流行，即开始花括号放在if语句末尾，结束花括号则直接放在if下面。

```
if ( y > 0 ) {
   z = x / y;
   Console.writeLine("z= " + z);
}
```

这两种方式都被广泛使用，当然它们编译的结果是一样的。

可以在Visual Studio.NET中设置选项来选择使用某种格式。在Tools | Options菜单，选择Text editor | C#文件夹，然后选择格式，如果想选择第二种方式，选中标有“Leave open braces”的复选框，就可以将开始花括号放在if的同一行上，而如果想选择第一种方式，则让此复选框处于未选中状态。

2.13 比较运算符

我们在前面使用 > 运算符表示“大于”。C#中的大多数比较运算符同C语言及其他语言都一样，在表2.5中，要特别注意的是，“等于”需要两个等号，“不等于”与FORTRAN或VB中的表示方法不同。

表2.5 C#中的比较运算符

>	大于
<	小于
==	等于
!=	不等于
>=	大于等于
<=	小于等于

2.14 条件合并

若需要在if语句或其他逻辑语句中合并两个或更多个条件，可以使用逻辑与、逻辑或和逻辑非等运算符（见表2.6）。除C/C++语言外，这些运算符与其他语言完全不同，并且表2.6中的运算符和位运算符很相似，容易造成混乱。

表2.6 C#中的布尔运算符

&&	逻辑与
||	逻辑或
~	逻辑非

因此，若我们在VB.NET中写下列语句：

```
If  ( 0 < x ) And ( x <= 24 ) Then
  Console.writeLine ("Time is up")
```

在C#中应该写成：

```
if ( (0 < x ) && ( x <= 24 ) )
   Console.writeLine("Time is up");
```

2.15 最常见的错误

由于等于运算符是==，赋值运算符是=，它们很容易被错误使用。如果写成：

```
if ( x = 0 )
  Console.writeLine("x is zero");
```

而不是

```
if ( x == 0 )
  Console.writeLine("x is zero");
```

会得到一个令人迷惑的编译错误“Cannot implicitly convert double to bool”（不能将双精度数隐式地转换成布尔值），原因在于

```
( x = 0)
```

的结果是双精度数0，而不是布尔值true或false。而

```
(x = = 0)
```

的结果确实是布尔量，编译器不会给出任何错误信息。

2.16 switch语句

switch语句能为一个变量提供多个可能的值，如果其中一个值匹配，则执行相应的代码。C#中的switch语句里，被比较的变量必须是整型、字符型或字符串类型，变量必须放在圆括号中。

```
switch (j){
  case 12:
    System.out.println("Noon");
    break;
  case 13:
    System.out.println("1 PM");
    break;
  default:
    System.out.println("some other time...");
}
```

特别注意，switch语句的每个case语句后面必须跟一个break语句，这一点很重要，因为break语句的含义是“转移到switch语句末尾”。如果漏掉了break语句，会产生一个编译错误。

如果没有一个值能匹配switch中的变量，控制会转移到switch语句末尾，此时也会产生一个错误，所以应当为switch语句提供一个default语句。

2.17 C#注释

正如在前面所看到的那样，C#中的注释以两个反斜杠开始，到当前行的末尾结束。C#也能识别C语言的注释，这种注释以/*开头，中间可以持续若干行，直到发现*/为止。

```
// C#  single-line comment
/* other C# comment style */
/* also can go on
for any number of lines */
```

C# 注释不能嵌套。一个注释以某种方式开始后，就要一直使用这种方式直到注释结束。

最初学习一种新语言时，可能会忽略注释。而如果在写程序时未做注释，程序就不易于理解，反之，若加入了注释，在以后要重复使用该代码时，就会欣喜地发现注释能帮你解释当初为什么这样做。

2.18 有争议的三元运算符

C# 继承了 C/C++ 和 Java 的一个相当不明智的部分：三元运算符。下面这条语句：

```
if (a > b)
   z = a;
else
   z = b;
```

可以写成很简单的形式：

```
z = ( a > b) ? a : b;
```

类似于后置的递增运算符，该语句引入 C 语言的最初目的是提示编译器生成效率较高的代码，并可以在终端速度较慢时减少输入。今天，现代编译器对上述两种形式的语句生成的代码是一样的，这样，三元运算符的最初作用已不复存在。某些由 C 转到 C# 的程序员认为这是一种“雅致”的缩写，而我们不这样认为，本书中也不使用三元运算符。

2.19 C# 的循环语句

C# 有四种循环语句：while，do-while，for 和 foreach。每种循环语句都提供一种方式来指定重复执行一组语句直到满足某个条件。

2.20 while 循环

while 循环很容易理解。只要条件为真，花括号中的所有语句会一直执行。

```
i = 0;
while (i<100)
   {
  x = x + i++;
   }
```

因为只要条件为真，循环就执行下去，因此，可能有的循环一次也不执行。当然，如果你不小心，有的 while 循环也会永远不停止。

2.21 do-while 语句

C# 的 do-while 语句和 while 语句相当类似，区别是这种情况下循环体至少执行一次，原因在于测试放在循环的底部。

```
i = 0;
do {
   x += i++;
}
while (i < 100);
```

2.22 for循环

for循环的结构性相当好。它由三部分组成：初始化、条件和每循环一次要发生的操作。每一部分都由分号分隔。

```
for (i = 0; i < 100; i++){
   x += i;
 }
```

现在我们来剖析这条语句。

```
for (i = 0;           // initialize i to 0
    i < 100;          // continue as long as  i < 100
    i++)              // increment i after every pass
```

在前面的循环里，i在第一次循环开始的时候设置为0，通过检测来确定i小于100，然后执行循环体。执行一遍循环体后，程序返回顶部，i加1，再检测看它是否小于100，如果是，再执行循环体。

注意这个for循环执行的操作与前面的while循环一样。for循环可以永远也不执行，也可以写一个循环体不存在的for循环。

2.23 在for循环中用到变量时再声明

在for循环中，一个很常见的声明变量的场合是需要一个计数器变量的时候，可以在for语句里直接声明变量，如下所示：

```
for ( int i = 0; i < 100; i++)
```

该循环变量只在循环内存在，一旦循环结束后就消失了。这一点很重要，因为一旦循环结束，试图引用该变量会产生编译错误信息。下面的代码是不正确的。

```
for ( int i = 0; i < 5; i++ )  {
    x[ i] = i;
}

// the following statement is in error
// because i is now out of scope
System.out.println("i=" + i);
```

2.24 for循环语句中的逗号

可以在C#的for语句的初始化部分初始化多个变量，也可以在语句的操作部分执行多个操作，这些语句要用逗号分隔。

```
for ( x = 0, y = 0, i = 0; i < 100; i++, y += 2 )
{
```

```
 x = i + y;
 }
```

这对循环的效率没有影响，但下面的写法更清楚。

```
x = 0;
y = 0;
for (i = 0; i< 100; i++)
  {
  x = i + y;
  y += 2;
  }
```

用逗号运算符把整个程序写在一个特别拥挤的 for 语句里也是可能的，但这会混乱程序意图。

2.25 C# 与 C 的区别

如果读者以前接触过C语言，或者读者是一个比较熟练的C程序员，对C#和C的主要区别可能会感兴趣。

1. C# 通常不使用指针。
2. 可以在一个方法里的任何地方声明变量，不必把声明语句放在方法的顶端。
3. 使用一个对象前，不一定要声明它，可以在用到的时候再定义。
4. C# 对结构体类型的定义有些不同，它根本不支持联合类型。
5. C# 有枚举类型，允许一系列被命名的量（如颜色或一周里的各天）赋值为连续的数值，但语法有些不同。
6. C# 没有位域，也就是说，变量至少要占用一个字节的存储空间。
7. C# 不支持变长参数列表。必须针对参数值和类型定义一个方法。然而，C# 语句允许函数的最后一个参数为可变参数数组。
8. C# 引入了代理和索引器的思想，这些在其他流行的语言中是没有的。

2.26 C# 与 Java 的区别

C# 与 Java 关系密切，由于 C# 是在 Java 之后设计出来的，它吸收了 Java 的大部分精华。但两者还是有一些细微差别。

1. 许多系统对象方法都有相同的方法名，只是在大小写形式上有区别。
2. C# 不提供 throws 关键字，该关键字使编译器检查你是否捕获了一个方法抛出的异常。
3. C# 对于布局管理器有更多的限制。因为它是以 Windows 系统为中心的，大多数时候采取的是图形元素的绝对位置。
4. C# 允许运算符重载。
5. C# 引进了代理和索引器。
6. C# 有枚举类型。
7. C# 有不安全模式，在这种模式下可以使用指针。
8. 必须专门声明一个方法能被覆盖及一个方法能覆盖另一个方法。
9. 不能通过声明来区别继承和接口实现，它们的声明方式是一样的。

10. switch语句允许使用字符串变量。如果变量没有被匹配，必须有一个默认情况，否则会出现错误。break语句是必需的。
11. 布尔变量类型在C#中拼为“bool”，在Java中拼为“boolean”。

2.27 小结

在这简短的一章里，我们学习了C#语言的基本语法元素。了解了工具，还需要知道如何使用这些工具。在接下来的章节里，将开始学习对象以及如何使用对象，并体会对象的强大功能。

第3章　用C#编写Windows程序

C#语言来源于C++，Visual Basic和Java。C#和VB.NET使用相同的类库，编译成同样的底层代码；二者都是受管语言，对不用的变量空间能自动进行垃圾收集，可以交替使用这两种语言；二者使用的类和方法名非常类似于Java，所以，如果读者熟悉Java的话，使用C#应该不会有问题。

3.1　C#中的对象

C#中的任何事物都看做是对象。对象包含数据并具有操作数据的方法。例如，串就是对象，拥有下列方法：

```
Substring
ToLowerCase
ToUpperCase
IndexOf
Insert
```

整型变量、浮点型变量和双精度型变量也是对象，它们也具有方法。

```
string s;
float x;
x = 12.3;
s = x.ToString();
```

注意，数值类型的转换是用方法而不是外部函数完成的。在把一个数值格式化成一个特定的字符串时，每种数值类型都提供了一个格式化方法。

3.2　受管语言和垃圾收集

C#和VB.NET都是受管语言。这有两个含义。第一，两者都被编译成中间的低层语言，都使用通用语言运行时（Common Language Runtime，CLR）执行编译后的代码，或是进一步编译这些代码。所以，C#和VB.NET不仅共享同样的运行时类库，而且在很大程度上是同一语言系统的两个方面。区别是：VB.NET更像Visual Basic，对VB程序员来说更容易学习和使用；而C#更像C++和Java，对于熟练使用这些语言的程序员来说更具吸引力。

另一个主要含义是，受管语言都是垃圾自动收集的。垃圾自动收集语言负责释放不用的内存（你根本不用考虑这些事情）。垃圾收集系统一检测到变量、数组或对象不再被引用，就把相应的内存空间释放回系统。当然，还是有可能写出消耗内存的代码，但大多数情况下，根本不用考虑内存的分配与释放问题。

3.3　C#中的类和名字空间

所有的C#程序完全都是由类组成的。可视化窗口就是一种类。读者以后也会看到我们写的所有程序都由类组成。因为任何事物都是类，类对象名的数量会很庞大，因此，系统将它们分到各种函数库中，要使用这些库的函数，必须特别指定。

库实际上是相互独立的 DLL。只需要使用 using 语句，通过基类名字来引用库，就可以使用该库中的函数。

```
using System;
using System.Drawing;
using System.Collections;
```

逻辑上，每个库代表一个不同的名字空间。每个名字空间是独立的一组类和方法名，声明了名字空间后，编译器就能识别所用到的类和方法名。可以使用包含相同类名和方法名的空间，但是如果使用了出现在多个名字空间的类和方法，会被告知发生了冲突。

最常用的名字空间是 System 名字空间，它在默认情况下就被导入，不需要声明。该名字空间包含了许多最基础的类和方法，C# 用它们来访问基类，如 Application，Array，Console，Exceptions，Objects，以及标准对象，如 byte，bool 和 string。在一个最简单的 C# 程序中，我们可以只向控制台输出一个信息，而不用生成窗口。

```
class Hello {
        static void Main (string [ ] args )   {
          Console.WriteLine ("Hello C# World");
          }
}
```

该程序只是向命令窗口（DOS）输出文本 “Hello C# World”。任何程序的入口点必须是 Main 函数，而且该函数必须声明为静态的。

3.4　构建一个 C# 应用程序

我们先创建一个简单的控制台程序，即一个没有任何窗体、只能从命令行运行的程序。启动 Visual Studio.NET 程序，选择 File | New Project，在选择窗口中选择 Console Application（控制台应用程序），如图 3.1 所示。

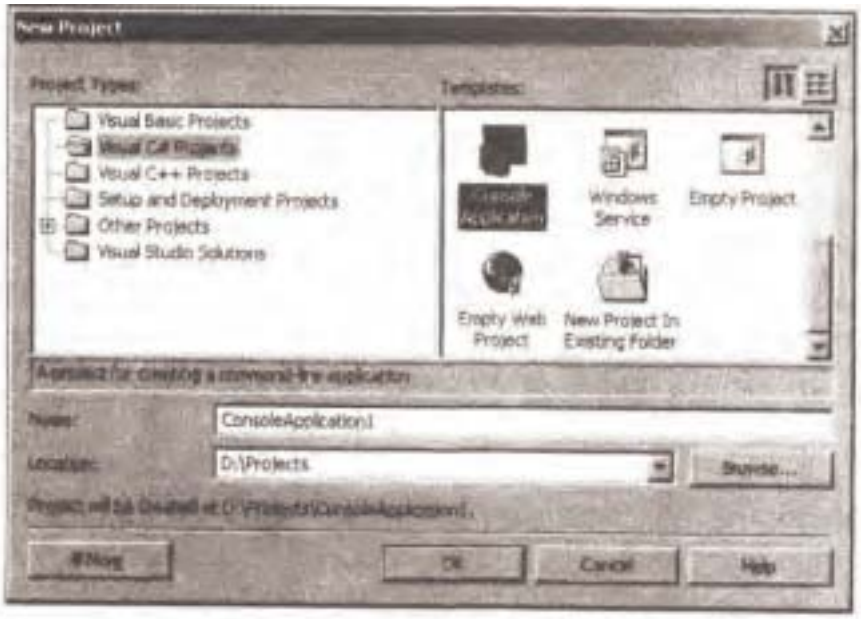


图 3.1　New Project 选择窗口：选择 Console Application

系统会自动生成一个包含 Main 的模块，读者可以参照下面键入其余的代码。

```
Console.writeLine ("Hello C# World");
```

按 F5 键编译运行该程序。

当按 F5 键编译运行该程序时，会出现一个 DOS 窗口，输出信息“Hello C# World”后退出。

3.5 用 C# 编写一个最简单的 Windows 程序

使用 C# 很容易创建 Windows GUI 程序，实际上，用窗口设计器（Windows Designer）可以生成程序的绝大部分。要做到这一点，先启动 Visual Studio.NET，选择 File | New Project，然后选择 C# Windows Application，应用程序的默认名（及文件名）是 WindowsApplication1，在关闭 New 对话框之前可以改变名字。上述操作会生成一个窗体项目，称做 Form1.cs，接下来像在 Visual Basic 里所做的那样，可以用工具箱（Toolbox）插入控件。

图 3.2 所示的是窗口设计器生成的一个简单窗体，带有一个文本域和一个按钮。

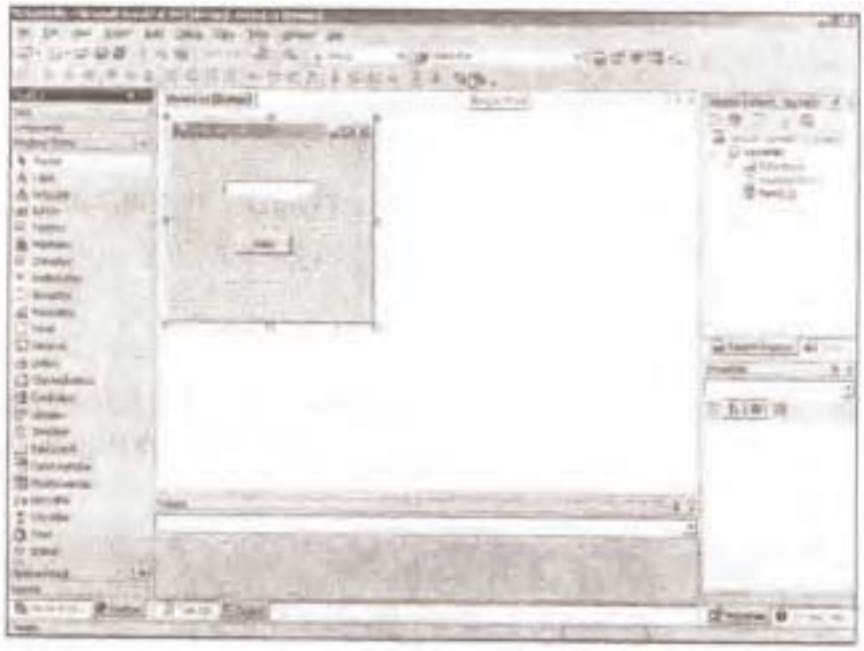


图 3.2 Visual Studio.NET 中的窗口设计器

从工具箱中选择文本框（TextBox），把控件拖到窗体中，就可以在窗体上画出控件，再以同样的方法画出按钮。然后，为了生成程序代码，只需要双击控件就可以了。在这个简单的窗体中，我们希望点击 Hello 按钮后，能将文本从文本域中复制到称为 txHi 的文本框中，并清除文本域。在窗口设计器中，双击按钮自动生成代码。

```
private void btHello_Click (object sender , EventArgs e )  {
txHi.Text = "Hello there ";
}
```

注意，Click 例程传递进来一个发送者对象和一个事件对象，可以通过查询它们获得更多的信息。实际上，是把事件和该方法联系起来。该程序的运行结果如图 3.3 所示。

尽管我们在前面的子例程里只需要写一行代码，但看一下代码的其余部分对该程序有何不同是有意义的。我们首先要导入几个类库，这样程序才能使用它们。

```
using System;
using System.Drawing;
```

```
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

其中最重要的是 Windows.Forms 类库，它对所有的.NET 语言都是通用的。

图 3.3　单击 Hello 按钮后的 SimpleHello 窗体

下面说明窗口设计器为控件生成的代码，这些代码是公开的，需要时可以修改。每个控件基本上都被声明为一个变量，并被添加到一个容器里。下面是控件的声明，注意事件处理程序加在了 btHello.Click 事件里。

```
private System.Windows.Forms.TextBox txHi;
private System.Windows.Forms.Button btHello;

private void InitializeComponent()     {
      this.btHello = new System.Windows.Forms.Button();
      this.txHi=new System.Windows.Forms.TextBox();
      this.SuspendLayout();
      //
      // btHello
      //
      this.btHello.Location=
                  new System.Drawing.Point(80,112);
      this.btHello.Name="btHello";
      this.btHello.size=new System.Drawing.Size(64,24);
      this.btHello.TabIndex=1;
      this.btHello.Text="Hello";
      this.btHello.Click +=
            new EventHandler(this.btHello_click);
      //
      //txHi
      //
      this.txHi.Location=
            new System.Drawing.Point(64,48);
      this.txHi.Name="txHi";
      this.txHi.Size=new System.Drawing.Size(104,20);
      this.txHi.TabIndex=0;
      this.txHi.Text=" ";
      //
      //Form1
      //
```

```
this.AutoScaleBaseSize=
      new System.Drawing.Size(5,13);
this.ClientSize=new System.Drawing.Size(240,213);
this.Controls.AddRange(
      new System.Windows.Forms.Control[]{
        this.btHello,
        this.txHi});
        this.Name="Form1";
        this.Text="Hello window";
        this.ResumeLayout(false);
      }
```

如果是手工修改代码，而不是使用属性页，窗口设计器可能无法再正常工作。我们在下一章讨论完对象和类之后，会进一步了解该系统的强大功能。

3.6 Windows 控件

所有基本的Windows控件的工作方式都和我们前面用到的文本框和按钮一样。图3.4所示的是一个 Windows controls 程序的输出界面，其中包含了很多常用的控件。

每个控件都有诸如Name，Text，Font，Forecolor 和 Borderstyle 等属性，用图3.2 右边的属性窗口可以很方便地修改它们，也可以在程序中修改这些属性。就像前面的程序那样，窗口设计器生成的Windows Form类总是创建一个Form1 构造函数，由它调用 InitializeComponent 方法。该方法一旦被调用，就创建了其余的控件，可以在代码中修改控件的属性。通常，我们会创建一个私有方法 init( )，它在 InitializeComponent 方法之后调用，在 init( )中可以添加额外的初始化代码。

图 3.4 Windows controls 程序的输出界面

3.6.1 标签

标签是窗体上的一个域，只能显示文本。程序员通常用标签标识紧挨着它的文本框的用途。既不能单击标签也不能对它使用 tab 键获得焦点。然而，如果愿意，可以在窗口设计器或运行时修改表 3.1 中的主要属性。

表 3.1　标签控件的属性

属性	说明
Name	只能在设计时使用
BackColor	一个 Color 对象
BorderStyle	None，FixedSingle 或 Fixed3D
Enabled	true 或 false，如果为 false，该控件外观变为灰色
Font	设置成一个新的 Font 对象
ForeColor	一个 Color 对象
Image	在标签里显示的一幅图像
ImageAlign	图象在标签中放置的位置
Text	标签的文本
Visible	true 或 false

3.6.2　文本框

文本框是可编辑一行或多行文本的控件，可以用文本框的Text属性来设置或获得文本框的内容。

```
TextBox tbox = new TextBox ();
tbox.Text = "Hello there ";
```

除了表 3.1 中的属性外，文本框也支持表 3.2 中的属性。

表 3.2　文本框的属性

属性	说明
Lines	一个字符串数组，每行文本都有一个
Locked	若为 true，不能向文本框键入文本
Multiline	true 或 false
ReadOnly	类似于 Locked，若为 true，只能选择文本、拷贝文本，否则可在程序中设置文本
WordWrap	true 或 false

3.6.3　复选框

根据Checked属性值，复选框可以是选中或未选中状态，可以在程序中或窗口设计器中设置或查询这一属性。在设计模式下，通过双击复选框创建一个事件处理程序，捕获复选框被选中或未选中的事件。

复选框有一个Appearance属性，可以被设置为Appearance.Normal或Appearance.Button。当复选框的外观属性设置为Button值时，该控件就像一个触发按钮一样，单击控件时处于按下状态，再次单击时，控件处于弹起状态。表 3.1 中的所有属性也同样适用于复选框。

3.6.4　按钮

按钮通常用来向程序发送命令。单击按钮时，产生一个事件，通常需要用事件处理程序去捕获。像复选框一样，在窗口设计器中，双击按钮会创建一个事件处理程序。表 3.1 中的所有属性同样适用于按钮。

按钮通常的显示方式是上面带有图片，可以在窗口设计器中或运行时设置按钮图片，图片可以是 bmp，gif，jpeg 或 icon 格式的文件。

3.6.5　单选按钮

单选按钮也称为选项按钮，是通过单击来选择的圆形按钮，一次只能选择一组单选按钮中的一个。如果在一个窗体里有多组单选按钮，就要像图 3.4 中的程序那样，将每组按钮放在一个 Group

框中。同复选框和按钮一样，在窗口设计器里通过双击可以把事件和按钮联系起来。单选按钮并不总是需要有与之联系的事件，而是在其他事件发生时（如单击 OK 按钮），程序员去检查单选按钮的 Checked 属性。

3.6.6 列表框与组合框

列表框与组合框都是在列表中含有一组元素项目。组合框是单行下拉式的，当选项很少改动时，程序员可以用它来节省空间。列表框可以通过设置属性而允许多项选择，但组合框不行。表3.3给出了它们的一部分属性。

表 3.3 列表框与组合框的属性

属性	说明
Items	列表中项目的一个集合
MultiColumn	如果为 true，ColumnWidth 属性给出了每列的宽度（不能用于组合框）
SelectionMode	One，MultiSimple 或 MultiExtended。若设置成 MultiSimple，可单击鼠标选择或取消选择多个项目；若设置成 MultiExtended，可用鼠标选择一组相邻项目（不能用于组合框）
SelectedIndex	选中项目的索引
SelectedIndices	当列表框的选择模式为多个项目时，返回选中项目的集合
SelectedItem	返回选中的项目

3.6.7 项目集合

可以用列表框和组合框里的项目集合去添加或删除显示列表的元素。它本质上就是一个 ArrayList（数组列表），我们在第 7 章会讨论它。项目集合的基本方法如表 3.4 所示。

表 3.4 项目集合的方法

方法	作用
Add	向列表添加对象
Count	列表中的项目数
Item[i]	集合中的第 i 个元素
RemoveAt(i)	删除元素 i

如果把列表框设置成多选项模式，可以用下列代码获得选中的项目集或选中的索引。

```
ListBox.SelectedIndexCollection it =
       new ListBox.SelectedIndexCollection (lsCommands );
ListBox.SelectedObjectCollection so =
       new ListBox.SelectedObjectCollection (lsCommands );
```

其中 lsCommands 是列表框名。

3.6.8 菜单

可以通过向窗体添加 MainMenu 控件来添加一个菜单。然后，选择菜单控件，编辑它的下拉项名字，添加新菜单项，如图 3.5 所示。

同其他可单击的控件一样，在窗口设计器里双击菜单会创建一个事件处理程序，读者可以在其中添加代码。

图 3.5 向窗体添加一个菜单

3.6.9 工具提示

工具提示（ToolTip）是把鼠标指针移到窗口中的控件时出现的方框。通过向窗体添加一个

ToolTip 控件（不可见的），然后向该控件添加指定的 ToolTip 控件和文本，这一属性才能激活。在图 3.4 的例子中，我们用加到窗口中的 tips 控件给按钮和列表框加上了工具提示文本。

```
tips.SetToolTip (btPush, "Press to add text to list box");
tips.SetToolTip (lsCommands, "Click to copy to text box");
```

图 3.6 举例说明了工具提示。

我们将在第 14 章和第 15 章讨论如何使用 DataGrid 和 TreeList，在第 28 章和第 29 章讨论 Toolbar。

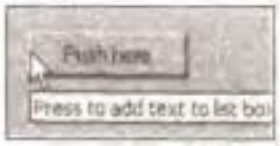


图 3.6　一个按钮的工具提示

3.7　Windows controls 程序

Windows controls 程序（输出界面如图 3.4 所示）控制着标签中文本的改变。

- 字体大小由组合框设置。
- 字体颜色由单选按钮设置。
- 粗体字由复选框设置。

对于复选框来说，是根据它的状态生成一种新字体，或者是粗体，或者是细体。

```
private void ckBold_CheckedChanged(object sender, EventArgs e) {
            if (ckBold.Checked ) {
                  lbText.Font =new Font ("Arial",
                              fontSize,FontStyle.Bold );
            }
            else {
                  lbText.Font = new Font ("Arial", fontSize);
            }
```

我们在创建窗体时，向组合框添加了字体尺寸的列表。

```
private void init() {
      fontSize = 12;
      cbFont.Items.Add ("8");
      cbFont.Items.Add ("10");
      cbFont.Items.Add ("12");
      cbFont.Items.Add ("14");
      cbFont.Items.Add ("18");
      lbText.Text ="Greetings";
      tips.SetToolTip (btPush, "Press to add text to list box");
      tips.SetToolTip (lsCommands, "Click to copy to text box");
}
```

当有人单击组合框中的一种字体尺寸时，就把相应的文本转换成数字，然后生成该尺寸的字体。注意，我们只是调用了对复选框做改动的代码，这样就不必重复做一些事情。

```
private void cbFont_SelectedIndexChanged (
             object sender, EventArgs e)  {
      fontSize = Convert.ToInt16 (cbFont.SelectedItem);
      ckBold_CheckedChanged(null, null);
}
```

对每个单选按钮，双击它并插入修改颜色的代码。

```
private void opGreen_CheckedChanged(object sender, EventArgs e) {
            lbText.ForeColor =Color.Green;
```

```
        }

        private void opRed_CheckedChanged(object sender, EventArgs e) {
              lbText.ForeColor =Color.Red;
        }

        private void opBlack_CheckedChanged(object sender, EventArgs e) {
              lbText.ForeColor =Color.Black;
        }
```

单击列表框时，它把获得的选项作为对象，然后将该对象转换成字符串，再拷贝到文本框中。

```
private void lsCommands_SelectedIndexChanged (
              object sender, EventArgs e )  {
      txBox.Text = lsCommands.SelectedItem.ToString();
}
```

最后，单击File | Exit菜单项，应该能关闭窗体以及程序。

```
private void menuItem2_Click(object sender, EventArgs e )  {
      this.Close ();
}
```

3.8 小结

我们现在已经学完了用C#编写程序的基本知识，在接下来的章节里准备详细介绍对象和面向对象程序设计。

3.9 随书附带光盘中的程序

Console Hello	\IntroCSharp\Hello
Windows Hello	\IntroCSharp\SayHello
Windows Controls	\IntroCSharp\WinControls

第4章　在C#中使用类和对象

4.1　关于类

所有的C#程序都由类组成。前面看到的Windows窗体就是类，它派生于基本的Form类，我们即将要编写的其他所有程序都是完全由类组成的。C#没有全局数据块或共享数据的概念，它们不是类的组成部分。

简单地说，类就是将一系列公有和私有方法以及私有数据组合在一个命名的逻辑单元中。通常将一个类放在一个单独文件里，尽管这不是硬性的规定。我们已经知道，Windows窗体就是类，在本章里，会进一步了解如何创建其他有用的类。

一个类创建后，它就不仅仅是一个单一实体了，而是一个“模板”，可以用new关键字创建它的副本或实例。创建实例时，可以用类的构造函数把初始化数据传到类中。构造函数是与类名相同的方法，没有返回类型，可以有0个或多个传递给类实例的参数。我们把实例叫做对象。在接下来的部分里，会创建一些简单的程序，并用类的实例简化程序。

4.2　一个简单的温度转换程序

假设我们要编写一个可视化程序，将温度值在摄氏度和华氏度之间转换。可能还会记得：水的冰点在0 °C或32°F，沸点是100°C或212°F。由这些数据可以很快推导出可能已经遗忘了的转换公式。

一种温标下冰点和沸点的差值是100，另一种温标下是180，即100/180或5/9。因为华氏温标下冰点是32°F，所以它的“偏移量”是32。因此有，

$$C = (F - 32) * 5/9$$

和

$$F = 9/5 * C + 32$$

在我们这个可视化程序里，允许用户输入一个温度值，并选择要转换的温标，如图4.1所示。

图4.1　将35°C转换成95°F的可视化界面

使用Visual Studio.NET提供的可视化构造器，几秒钟就可以画出用户界面，并且很容易实现两个按钮按下时要调用的例程。如果双击Compute按钮，程序会生成btConvert_Click方法，可以在其中添加代码，使之完成温标之间的转换工作。

```
private void btCompute_Click(object sender,
            System.EventArgs e) {
        float temp, newTemp;
        //convert string to input value
        temp = Convert.ToSingle (txEntry.Text );
        //see which scale to convert to
        if(opFahr.Checked)
              newTemp = 9*temp/5 + 32;
        else
              newTemp = 5*(temp-32)/9;
        //put result in label text
        lbResult.Text =newTemp.ToString ();
        txEntry.Text ="";    //clear entry field
 }
```

上述这个程序非常直接，很容易理解，它是某些简单的C#程序如何工作的一个典型例子。但是，该程序还是有些缺陷需要改进。

最明显的问题是，用户界面和数据处理都集中在一个程序模块里，而不是分别进行处理。通常，把数据处理和界面处理分开是一种好的想法，这样，修改界面的逻辑不会影响计算逻辑，反之亦然。

4.3 构造一个Temperature类

C#中的类是一个模块，它包含公有的和私有的函数与子例程，也包含私有数据。通常将类中的函数和子例程统称为方法。

类模块把一系列数据放在单独的命名空间中，用取出（get）函数和设置（set）函数存取数据，我们将这两种函数称为访问方法。

在C#集成开发环境（IDE）里，可以使用菜单项Project | Add Class module创建一个类模块。给每个新类指定一个文件名时，IDE也把这个名字作为类的名字，并生成一个带有空构造函数的空类。例如，如果创建一个Temperature类，IDE会生成如下代码。

```
namespace CalcTemp
{
        ///<summary>
        ///Summary description for Temperature
        ///</summary>
        public class Temperature
        {
             public Temperature()
             {
                    //
                    //TODO: Add constructor logic here
             }
        }
}
```

如果在“Summary description”里加上了专门的注释，在鼠标移到该类的实例上时，就会出现相应的注释文本。注意，系统生成了类和一个空构造函数。如果你的类需要一个带参数的构造函数，可以编辑代码来实现。

接下来我们要把所有的计算和温标之间的转换都移到新的Temperature类里。设计该类的一种方式是：先重写使用该类模块的调用程序。我们在下面的代码示例中，创建一个Temperature类的实例，用它完成转换工作。

```
private void btCompute_Click(object sender, System.EventArgs e) {
       string newTemp;
       //use input value to create instance of class
       Temperature temp = new Temperature (txEntry.Text );
       //use radio button to decide which conversion
       newTemp = temp.getConvTemp (opCels.Checked );

       //get result and put in label text
       lbResult.Text =newTemp.ToString ();
       txEntry.Text ="";     //clear entry field
}
```

实际的类如下所示。注意，这里把输入的温度字符串放到类的构造函数里，它将字符串转换成浮点值。不必知道数据的内部表示，也能在任何时候改变它。

```
public class Temperature    {
       private float temp, newTemp;
       //-------------
       //constructor for class
       public Temperature(string thisTemp)       {
            temp = Convert.ToSingle(thisTemp);
       }
       //-------------
       public string getConvTemp(bool celsius){
       if (celsius)
              return getCels();
       else
              return getFahr();
       }
       //-------------
       private string getCels() {
              newTemp= 5*(temp-32)/9;
              return newTemp.ToString();
       }
       //-------------
       private string getFahr() {
              newTemp = 9*temp/5 + 32;
              return Convert.ToString(newTemp);
       }
}
```

注意，温度变量temp声明为私有的，这样类的外部就不能访问它。只能用构造函数和getConvTemp方法存取数据。重新安排该代码的要点是，外部调用程序不必知道数据如何存储、如何取回，这是类内部需要知道的事情。

类的另外一个重要特性是它包含了数据。可以把数据放到类中，也可以在以后任何时候取回数据。我们这个类只包含了一个温度值，实际上类可以包含相当复杂的数据，这就是所谓的封装。

很容易修改这个类，来获得其他温标下的温度值，而且还不需要类的用户去了解如何存储数据或如何执行转换等事情。

4.3.1 转换成热力学温度

摄氏温标的绝对零点定义为-273.16°C，这可能是最冷温度，因为在该温度下所有的分子运动都停止了。热力学温标基于绝对零度，其刻度大小同摄氏度一样。我们可以加入下面的函数：

```
public string getKelvin()  {
       newTemp = Convert.ToString (getCels() + 273.16 )
}
```

那么，setKelvin方法应该是什么样的？

4.4 将判断放在Temperature类里

这次，我们仍然把使用Temperature类的哪个方法的判断放在用户接口里。如果能把复杂操作都隐藏在Temperature类里，可能会更好一些。但是，仅仅把单击Conversion按钮的方法写成下列形式，就已经不错了。

```
private void btCompute_Click(object sender, System.EventArgs e) {
      Temperature temper =
            new Temperature (txEntry.Text, opCels.Checked );
      //put result in label text
      lbResult.Text =temper.getConvTemp();
      txEntry.Text ="";     //clear entry field
}
```

这段代码把判断过程移到Temperature类里，将调用接口程序减少到只有两行代码。

完成这些处理的类Temperature变得更复杂了，但是它能够跟踪传递进来的数据和所做的转换。我们把数据和单选按钮的状态传递给它的构造函数。

```
public Temperature ( string sTemp, bool toCels )   {
       temp = Convert.ToSingle(sTemp);
       celsius = toCels;
}
```

现在，布尔型变量Celsius会告诉类，是否转换以及在取出温度值时要做哪一种转换。输出例程简化为：

```
public string getConvTemp(){
       if (celsius)
             return getCels();
       else
             return getFahr();
}
//-------------
private string getCels() {
      newTemp= 5*(temp-32)/9;
      return newTemp.ToString();
}
//-------------
private string getFahr() {
      newTemp = 9*temp/5 + 32;
      return Convert.ToString(newTemp);
}
```

在这个类中，有公有和私有两种方法。公有方法可以被其他模块调用，如用户界面窗体模块。私有方法（getCels和getFahr）在内部使用，对温度变量进行操作。

注意，现在能把要输出的温度值返回成字符串形式或浮点值形式，这样就可以根据需要来改变输出格式。

4.5　使用类完成格式化和数值转换工作

很多情况下，用一个方法来完成数据格式和数据表示之间的转换是很方便的。可以使用类处理这类转换并隐藏具体的细节。例如，可以设计一个程序，在程序里输入以分和秒表示的时间，分和秒之间可以有冒号也可以没有。

```
315.20
3:15.20
315.2
```

因为可能会有各种各样的输入格式，所以最好设计一个类来分析所有合法的情况，并把数据以标准格式存到类中。图 4.2 表示输入项为 112 和 102.3 时的分析情况。

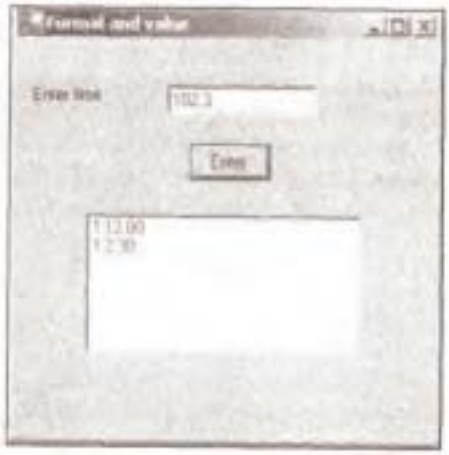


图 4.2　一个使用 Times 类的简单分析程序的界面

大部分的分析工作都发生在类的构造函数里。分析工作的主要依据是查找冒号，如果没有冒号，大于 99 的数当做分钟。

```
public FormatTime(string entry)        {
errflag = false;
if (! testCharVals(entry)) {
       int i = entry.IndexOf (":");
       if (i >= 0 ) {
             mins = Convert.ToInt32 (entry.Substring (0, i));
             secs = Convert.ToSingle (entry.Substring (i+1));
             if(secs >= 60.0F ) {
                    errflag = true;
                    t = NT;
             }
             t = mins *100 + secs;
       }
       else {
             float fmins = Convert.ToSingle (entry) / 100;
             mins = (int)fmins;
             secs = Convert.ToSingle (entry) - 100 * mins;
             if (secs >= 60) {
                    errflag = true;
                    t = NT;
             }
             else
```

```
                t = Convert.ToSingle(entry);
            }
        }
    }
```

由于可能会输入非法的时间值，所以要检测像89.22这种情况，并设置一个错误标志（记住1分钟只有60秒）。

根据所表示的时间种类，可能会有一些非数值的输入项，如NT代表没有时间，如果是体育时间，SC表示起跑时间，或者DQ表示无效时间。这一切都在类中得到了最好的处理，你不必知道这些值在内部使用什么样的数据来表示。

```
static public int NT = 10000;
static public int DQ = 20000;
```

图4.3给出了程序处理这类情况的一些结果。

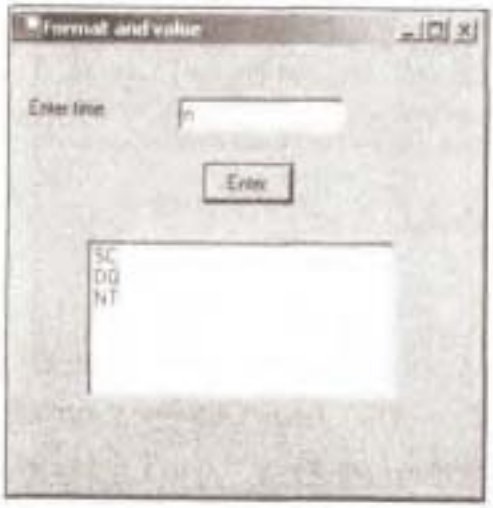


图4.3 时间输入界面显示了对表示起跑时间、无效时间和没有时间的符号的分析结果

4.5.1 处理不合理的数值

类也是封装错误处理的好地方。例如，超过某些极限值的时间是不可能的，这可能是输入时没有小数点的时间。如果不可能有特别大的时间值，那么像123473这样的数应该被看成12:34.73。

```
public void setSingle(float tm )  {
        t = tm;
        if((tm > minVal ) && (tm < NT))  {
                t = tm / 100.0f;
        }
}
```

下限值minVal随着考虑的时间范围的不同而不同，因而应该是一个变量。也可以用类的构造函数给变量设置默认值。

```
public class FormatTime {
  public FormatTime(string entry )     {
        errflag = false;
        minVal = 1000;
        t = 0;
```

4.6　分析字符串的类

许多语言都提供一个简单的方法把字符串分成若干个单词，各个单词用一个指定的字符分隔开。C#恰好没有为这一功能提供相应的类，而用String类中的Split方法可以很容易地构造这样的类。Tokenizer（单词分析器）类的作用是，传进来一个字符串，再一次返回一个单词。例如，如果我们有一个简单的字符串：

```
Now is the time
```

则单词分析器会返回下面四个单词：

```
Now
is
the
time
```

该类的关键部分是，要保存原始字符串并记住下一个要返回的单词。

这里用到了Split函数，它类似于Tokenizer，但返回的是一组子串而不是对象接口。要编写的类有一个nextToken方法，它返回单词，或者在达到单词串末尾时返回零长度的字符串。下面给出的是完整的类。

```
//String Tokenizer class
public class StringTokenizer     {
  private string data, delimiter;
  private string[] tokens;  //token array
  private int index;        //index to next token
//----------
public StringTokenizer(string dataLine)          {
       init(dataLine, " ");
}
//----------
//sets up initial values and splits string
private void init(String dataLine, string delim) {
       delimiter = delim;
       data = dataLine;
       tokens = data.Split (delimiter.ToCharArray() );
       index = 0;
}
//----------
public StringTokenizer(string dataLine, string delim) {
       init(dataLine, delim);
}
//----------
public bool hasMoreElements() {
       return (index < (tokens.Length));
}
//----------
public string nextElement() {
       //get the next token
       if( index > tokens.Length )
              return tokens[ index++] ;
       else
              return "";     //or none
       }
}
```

图4.4说明了类的运行情况。

使用Tokenizer类的程序代码如下：

```
//call tokenizer when button is clicked
private void btToken_Click(object sender,
            System.EventArgs e) {
      StringTokenizer tok =
            new StringTokenizer (txEntry.Text );
      while(tok.hasMoreElements () ) {
            lsTokens.Items.Add (tok.nextElement());
      }
}
```

图4.4 运行中的Tokenizer类

4.7 类与对象

面向过程的程序设计和面向对象（OO）的程序设计的主要差别在于类的出现。类只是一个模块，就像我们前面给出的那样，它既包含公有的和私有的方法，也包含数据。类是惟一的，一个类可以有多个实例，每个实例包含不同的数据。通常把这些实例称为对象，后面我们会看到单个实例和多个实例的例子。

假设把一场游泳比赛的结果存储在一个文本数据文件里。文件类似于下列形式，各个列分别代表位置、名字、年龄、俱乐部和时间（成绩）。

```
1  Emily Fenn          17   WRAT    4:59.54
2  Kathryn Miller      16   WYW     5:01.35
3  Melissa Sokolnik    17   WYW     5:01.58
4  Sarah Bowman        16   CDEV    5:02.44
5  Caitlin Klick       17   MBM     5:02.59
6  Caitlin Healey      16   MBM     5:03.62
```

如果我们编写一个程序来显示运动员和他们的成绩，需要读入文件并进行分析。对每个运动员都要保存姓名、年龄、俱乐部和成绩。一种组织运动员数据的有效方式是设计一个Swimmer类，为每个运动员创建一个实例。

下面是我们读文件和创建实例的代码。每个实例创建好之后，都加到ArrayList对象中。

```
private void init() {
      ar = new ArrayList ();    //create array list
      csFile fl = new csFile ("500free.txt");
```

```
        //read in lines
        string s =  fl.readLine ();
        while (s != null) {
                //convert to tokens in swimmer object
                Swimmer swm = new Swimmer(s);
                ar.Add (swm);
                s= fl.readLine ();
        }
        fl.close();
        //add names to list box
        for(int i=0; i < ar.Count; i++) {
                Swimmer swm = (Swimmer)ar[ i];
                lsSwimmers.Items.Add (swm.getName ());
        }
}
```

Swimmer类分析文件中的每一行数据，并将数据存储起来，用getXxx访问函数可以访问这些数据。

```
public class Swimmer {
        private string frName, lName;
        private string club;
        private int Age;
        private int place;
        private FormatTime tms;
//-----------
public Swimmer(String dataLine)       {
        StringTokenizer tok = new StringTokenizer (dataLine);
        place =  Convert.ToInt32 (tok.nextElement());
        frName = tok.nextElement ();
        lName = tok.nextElement ();
        string s = tok.nextElement ();
        Age = Convert.ToInt32 (s);
        club = tok.nextElement ();
        tms = new FormatTime (tok.nextElement ());

 }
//-----------
 public string getName() {
        return frName+" "+lName;
 }
//-----------
public string getTime() {
        return tms.getTime();
 }
}
```

4.8　类包含

Swimmer类的每个实例都包含了一个StringTokenizer类实例，用于分析输入的字符串，还包含了一个我们前面写的Times类的实例，用于分析时间并将其以格式化形式返回给调用程序。让一个类包含其他类在面向对象的程序设计中很常见，也是由简单程序构造复杂程序的一种主要方式。显示游泳选手成绩的程序的运行情况如图4.5所示。

图 4.5 游泳选手及其成绩的列表，程序中用到了包含

点击任何一个游泳选手，他的成绩就会显示在右边的文本框中。由于所有的数据都在Swimmer类中，显示时间的代码很容易编写。

```
private void lsSwimmers_SelectedIndexChanged(
            object sender, System.EventArgs e) {
  //get index of selected swimmer
      int i = lsSwimmers.SelectedIndex;
      //get that swimmer
      Swimmer swm = (Swimmer)ar[ i];
      //display her time
      txTime.Text =swm.getTime();
}
```

4.9 初始化

注意，在前面的Swimmer类里，它的构造函数接下来调用了StringTokenizer类的构造函数。

```
public Swimmer(String dataLine )      {

      StringTokenizer tok =
            new StringTokenizer (dataLine);
```

4.10 类和属性

C#的类拥有属性(Property)方法，还拥有公有、私有函数及子程序。这些属性方法相当于Form的各种属性，但它们能存储、取出要使用的各种数据。例如，下面的代码没有getAge方法和setAge方法，只有一个Age属性，它相当于get方法和set方法。

```
private int Age;
//Age property
public int Age{
      get {
         return Age;
      }
      set{
            Age=value;
      }
}
```

注意，属性声明在属性名后没有圆括号，专门的关键字value用来获得存储的数据。

要使用这些属性，可以把Age属性放在等号左边来设置值，把Age属性放在等号右边取回值。

```
age = sw.Age;      //Get this swimmer's age
sw.Age = 12;       //Set a new age for this swimmer
```

属性最初更多地用于VB中的窗体，但许多程序员发现它们相当有用。使用get方法和set方法与使用属性提供的特性相同，且二者产生的代码效率相同。

在SwimmerTimes显示程序的修订版本里，把所有的get方法和set方法都转换成属性，允许用户输入新的时间来修改每个游泳选手的成绩。下面是Swimmer类。

```
public class Swimmer
{
        private string frName, lName;
        private string club;
        private int Age;
        private int place;
        private FormatTime tms;
//-----------
public Swimmer(String dataLine)       {
        StringTokenizer tok = new StringTokenizer (dataLine);
        place =  Convert.ToInt32 (tok.nextElement());
        frName = tok.nextElement ();
        lName = tok.nextElement ();
        string s = tok.nextElement ();
        Age = Convert.ToInt32 (s);
        club = tok.nextElement ();
        tms = new FormatTime (tok.nextElement ());

}
//-----------
public string name {
        get{
        return frName+" "+lName;
        }
}
//-----------
public string time {
        get {
          return tms.getTime();
        }
        set {
          tms = new FormatTime (value);
        }
}
//------------------
//age property
public int age {
        get {
           return Age;

        set {
           Age = value;
        }
}
}
```

接下来可以为任何一个游泳选手输入新的成绩，当txTime文本域失去焦点时，可以按如下方式存储新成绩。

```
private void txTime_onLostFocus(
        object sender, System.EventArgs e){
        //get index of selected swimmer
        int i = lsSwimmers.SelectedIndex;
        //get that swimmer
        Swimmer swm = (Swimmer)ar[ i] ;
        swm.time =txTime.Text;
}
```

属性的一个显著优点是能够按照下列方式书写语句。

```
sw.Age += 10;
```

它比下面的方式易于阅读，也更友好。

```
sw.setAge (sw.getAge() + 10 );
```

4.11 C# 的程序设计风格

可以为C#开发出许多种容易阅读的程序设计风格。我们这里使用的风格一部分受微软的匈牙利注释法（以它的发起人命名，Charles Simonyi）的影响，一部分是Java的设计风格。

建议在命名C#控件时（比如按钮、列表框等）使用前缀，这样可以使名字的意图更清楚，当一个窗体上有多个控件时，就使用这种命名法。

如果在代码中没有直接引用标签、框架和窗体，通常不给它们起新名字。让类的名字以大写字母开头，类的实例以小写字母开头。并且在类和实例的名字里使用大小写混合形式，使其用途更加清楚，例如

```
swimmerTime
```

控件名	前缀	示例
Buttons	bt	btCompute
ListBoxes	ls	lsSwimmers
Radio（单选按钮）	op	opFSex
ComboBoxes	cb	cbCountry
Menus	mnu	mnuFile
TextBoxes	tx	txTime

4.12 代理

C#引入了类C语言中的一个独特属性，叫做代理。代理本质上是对另一个类中的函数的引用，可以分派和使用函数而无需知道它来自哪一个类，只要函数满足同样的接口就可以。

考虑图4.6中的简单程序。单击Process按钮时，输入域中的文本以全大写或全小写的形式拷贝到列表框中。尽管有许多编写该程序的简单方法，而我们这里使用它的目的是为了说明代理。

代理是一个类方法的原型，以后会赋给它一个实际的“身份”。方法既可以是静态的也可以来自某个类实例。可以将代理声明为一种类型声明。

```
private delegate string fTxDelegate (string s);
```

接下来可以声明该类型的一个或多个实例，并给它们赋值。下面是一个代理变量。

图4.6　说明代理的一个简单程序

```
fTxDelegate ftx;          //instance of delegate
```

注意变量ftx表示某个类中一个特定方法的实例，它接受一个输入串，返回一个输出串。

确定选择了哪一个单选按钮后，选择放到该代理变量中的函数。创建一个Capital类，它有一个fixText方法，如下所示。

```
public class Capital
{
      public string fixText ( string s )  {
            return s.ToUpper ();
      }
}
```

再创建一个Lower类，它同样拥有一个fixText方法，只是在这里是一个静态方法。

```
public class Lower
{
      public static string fixText ( string s )  {
            return s.ToLower ();
      }
}
```

现在单击两个单选按钮中的一个，把其中的一个fixText方法赋给该代理变量。

```
private void opCap_CheckedChanged(object sender, EventArgs e) {
       btProcess.Enabled =true;
       //assign an instance method to the delegate
       //create an instance of the Capital class
       ftx = new fTxDelegate (new Capital().fixText);
}
//-----
private void opLower_CheckedChanged(object sender, EventArgs e) {
       btProcess.Enabled =true;
       //assign a static method to the delegate
       //the Lower class has a static method fixText
       ftx = new fTxDelegate (Lower.fixText);
}
```

注意，由类实例创建代理的语法与由静态方法创建代理的语法稍有不同。

```
ftx =new fTxDelegate (new Capital().fixText );  //instance
ftx =new fTxDelegate (Lower.fixText );          //static
```

接下来，在单击 Process 按钮时，只要用 ftx 代理去执行相应的方法就可以了。

```
private void btProcess_Click (object sender, EventArgs e )   {
        string s = ftx( txName.Text );
        lsBox.Items.Add ( s );
}
```

ftx 方法去解决调用哪一个 fixText 方法的问题。

代理方法可以在编辑时提供更多的灵活性，但它确实不是一个全新的方法，因为可以使用接口并直接调用 fixText 方法实现同样的功能。

4.13 索引器

C# 引入了一种特殊的类方法，叫做索引器。它可以用某种方法访问类中的数据元素，使该数据看起来像数组元素一样。在下面的 BitList 类中，索引器返回类中数据的第 i 位值。

```
public class BitList
{
        private int number;
        public BitList(string snum) {
                number = Convert.ToInt32 (snum);
        }
        //------
        // here is an indexer that
        // returns the i-th bit of a value
        public int this[ int index] {
              get{
                        int val = number >> index;
                        return val & 1;
              }
        }
}
```

在图 4.7 中，使用下面的简单代码，在每次单击数值型垂直滚动控件时，用得到的索引值去获取 BitList 类中数据的相应位的值。

图 4.7 用索引器取出一个数字各位上值的演示程序

```
private void numericUpDown1_ValueChanged(
              object sender, EventArgs e) {
        //get the index value from the updown control
        int index = Convert.ToInt32 (numericUpDown1.Value );
        //create an instance of the BitList class
```

```
        BitList bits = new BitList(txNum.Text );

        //get that bit value using the indexer
        int bit = bits[ index];
        //add it to the list box
        lsBits.Items.Add (Convert.ToString (bit));

}
```

正如所看到的那样，索引器对类似于数组的引用是很方便的，但是它不提供函数，不如类方法那样容易实现。

4.14　运算符重载

在C#代码的不安全块中，使用指针的同时也允许重载大多数常用的运算符。

```
+  -  *  /  %  &  |  ^  <<  >>  ==  !=  >  <  >=  <=
```

可以为任何给定的类重新指定这些运算符的含义。可以为一个假定的Complex类定义“+”运算符，方法如下：

```
public static Complex operator +(Complex c1, Complex c2 )
  {
     return new Complex(c1.real + c2.real,
                c1.imaginary + c2.imaginary);
  }
```

这样做会使程序非常难以理解，我们在本书中不使用这一特性。

4.15　小结

在本章中介绍了C#类，指出类能够包含公有的和私有的方法，还可以包含数据。每个类可以有多个实例，每个实例可以包含不同的数据值。类也可以用属性方法设置、取回数据。属性方法提供了一个比通常的getXxx和setXxx访问方法更简单的语法，但没有其他实质上的优点。

4.16　随书附带光盘中的程序

Temperature Conversion	`\UsingClasses\CalcTemp`
Temperature Conversion Using Classes	`\UsingClasses\ClsCalcTemp`
Temperature Conversion Using Classes	`\UsingClasses\AllClsCalcTemp`
Time Conversion	`\UsingClasses\Formatvalue`
String Tokenizer	`\UsingClasses\TokenDemo`
Swimmer times	`\UsingClasses\SwimmerTokenizer`
Delegate	`\UsingClasses\Delegate`
Indexer	`\UsingClasses\Indexer`

第5章 继 承

现在开始学习OO语言（如C#和VB.NET）的一个重要特性：继承。创建Windows窗体时，比如Hello窗体，IDE（集成开发环境）会生成一个下列类型的声明。

```
public class Form1 : System.Windows.Forms.Form
```

这表明我们创建的窗体是Form类的一个子类，而不是它的实例。这具有非常重要的含义：可以创建一些可视化的对象，并重定义它们的某些属性，这会让每个对象的行为稍微有些不同。下面会看到一些这样的例子。

5.1 构造函数

所有的类都有专门的构造函数，在创建类实例时调用它。构造函数的名字总是与类名相同，这一点适用于Form类，也适用于非可视化类。下面是系统为Form1类中的Hello窗口生成的构造函数。

```
public class Form1  {
     public Form1 () {            //constructor
            InitializeComponent ();
      }
```

在创建自己的类时，应当创建构造函数去做初始化工作，可以把参数传到类中，将类中的数据初始化为指定值。如果编写的类里没有构造函数，系统会自动生成一个不带参数的构造函数。

InitializeComponent方法也是由IDE生成的，它包含了创建和安置窗口中所有可视化控件的代码。如果需要为Form类增加其他初始化代码，通常是编写一个私有的init方法，在InitializeComponent方法调用之后调用它。

```
public Form1(){
       InitializeComponent();
       init();
}

private void init(){
      x = 12.5f;        //set initial value of x
}
```

5.2 C# 中的绘图和 Graphics 对象

在第一个例子中，我们要编写一个在窗体上的图片框（PictureBox）里画一个矩形的程序。C#中的控件由Windows系统重新绘制，paint（绘制）事件发生时，可以将它连到paint事件处理程序上来完成自己的绘制工作。当窗口改变大小、重新可见或刷新时，发生paint事件。为了说明这一点，我们创建一个窗体（Form），它包含一个图片框，如图5.1所示。

图 5.1 在窗体上插入一个图片框

然后，在窗口设计器中选择PictureBox，在Properties窗口选择Events按钮（带有闪电形图标），就会出现一个发生在PictureBox上的所有事件的列表，如图5.2所示。

图 5.2 为PictureBox选择Paint事件

双击Paint事件，在Form的代码里会生成下列空方法：

```
private void pic_Paint (object sender,  PaintEventArgs e )  {
}
```

在InitializeComponents方法里也为PictureBox生成了将该方法连接到Paint事件的代码。

```
this.pic.Paint += new PaintEventHandler(this.pic_Paint);
```

PaintEventArgs对象由底层系统传递给子例程，可以由该对象获得要绘制的图形。为了绘图，必须创建一个Pen对象的实例，并定义颜色，还可以选择它的宽度。下面的代码使用了一个默认宽度为1的黑色画笔。

```
private void pic_Paint(object sender, PaintEventArgs e )   {

  Graphics g = e.Graphics;              //get Graphics surface
```

```
    Pen rpen = new Pen(Color,Black);          //create a Pen
    g.drawLine(rpen, 10,20,70,80);            //draw the line
}
```

在这个例子里，每次发生Paint事件时，就创建Pen对象。也可以在窗口的构造函数里或者在它调用的init方法里，只创建一次画笔。

5.3 使用继承

在C#中，使用继承能够从现有的类派生出新类。在新的派生类里，只需要给出新的方法或修改了的方法，其他方法由被继承的基类自动提供。为了说明继承是如何工作的，我们编写一个简单的Rectangle类，它能在窗体中绘制自身。该类只有两个方法：构造函数和draw方法。

```
namespace CsharpPats
{
public class Rectangle{
        private int x, y, w, h;
        protected Pen rpen;
        public Rectangle(int x_, int y_, int w_, int h_)
        {
                                //save coordinates
                x = x_;
                y = y_;
                w = w_;
                h = h_;
                //create a pen
                rpen = new Pen(Color.Black);
        }
        //-----------------
        public void draw(Graphics g) {
                //draw the rectangle
                g.DrawRectangle (rpen, x, y, w, h);
        }
}
}
```

5.4 名字空间

前面已经提到过名字空间。Visual Studio.NET为每个项目创建一个和项目名相同的名字空间。可以在属性页修改该名字空间，也可以把它变为空白，这样该项目就不在名字空间里。仍然可以创建属于自己的名字空间，Rectangle类为这种做法提供了一个好的理由。程序需要使用System.Drawing名字空间的Graphics对象，而该名字空间也包含一个Rectangle类。正如前面所看到的那样，不需要对新的Rectangle类重新命名来避免名字交叠或冲突，只是把整个Rectangle类放在它自己的名字空间。

接下来在主Form窗口里声明变量，我们将其声明为该名字空间的成员。

```
CsharpPats.Rectangle rec;
```

在主Form窗口里，创建自己的Rectangle类的实例。

```
private void init() {
        rect = new CsharpPats.Rectangle (10, 20, 70, 100);
}
```

```
//---------------
public Form1()  {
        InitializeComponent();
        init();
}
```

然后将绘制代码添加到完成绘制工作的Paint事件处理程序中，将图形传递给Rectangle实例。

```
private void pic_Paint (object sender, PaintEventArgs e )  {
        Graphics g = e.Graphics;
        rect.draw (g);
}
```

得到如图5.3中的显示。

图5.3 矩形绘制程序的输出结果

5.4.1 由Rectangle类创建Square类

正方形只不过是矩形的特例，无需编写任何新代码就可以从Rectangle类派生出Square类。下面是完整的Square类：

```
namespace CsharpPats  {
        public class Square : Rectangle    {
        public Square (int x, int y, int w ):base (x, y, w, w)   {
                }
        }
}
```

该Square类只包含一个构造函数，父类Rectangle的构造函数作为Square构造函数的一部分，通过调用它将正方形的尺寸传递给底层Rectangle类。

```
base( x, y, w, w)
```

注意这一不寻常的语法：父类构造函数的调用跟在冒号后面，并放在了自身构造函数的开始花括号的前面。

Rectangle类创建画笔并完成实际的绘制工作。注意，根本没有Square类的draw方法。如果没有指定新方法，会自动使用父类的方法，而这正是我们所希望发生的。

绘制矩形和正方形的程序有一个简单的初始化函数，对象实例在这里创建。

```
private void init ()   {
       rect = new Rectangle ( 10, 20, 70, 100);
       sq = new Square (150, 100, 70);
}
```

该程序还有一个绘制子例程，在这里画出对象实例。

```
private void pic_Paint ( object sender, PaintEventArgs e )   {
       Graphics g = e.Graphics;
       rect.draw (g);
       sq.draw (g);
}
```

绘制的矩形和正方形如图 5.4 所示。

图 5.4 Rectangle 类和由它派生出的 Square 类

5.5 公有的、私有的和保护的

在C#中，可以将变量和类方法声明为公有的（public）、私有的（private）或保护的（protected）。公有方法可以被其他的类访问，私有方法只能在类内部访问。通常将所有的类变量声明为私有的，并编写 getXxx 和 setXxx 访问函数来设置或获取数据。让类中的变量能在类外直接访问，通常是一个很差的主意，因为它违反了封装原理。换句话说，类应该是惟一知道实际数据表示的地方，你应该能修改类内部的算法，但类外面的任何人都不能这样做。

C# 也使用 protected 关键字，变量和方法都可以是受保护的，受保护的变量能在该类及其派生类里访问。同样地，受保护的方法也只能由该类和它的派生类访问。类外部不能访问受保护的变量和方法。如果没有声明任何访问级别，默认为私有访问。

5.6 重载

同其他面向对象语言一样，C# 也允许有若干个名字相同的方法，只要这些方法有不同的调用参数。例如，我们要创建一个 StringTokenizer 类的实例，在其中定义字符串和分隔符。

```
tok = new StringTokenizer ("apples, pears", ",");
```

通过把构造函数声明为带有不同数目的参数，就可以重载构造函数。下面是两个构造函数。

```
public StringTokenizer(string dataLine)          {
       init(dataLine," ");
}
//---------
public StringTokenizer(string dataLine, string delim){
       init(dataLine, delim);
}
private void init(string data, string delim){
       //...
}
```

当然，C#也允许重载任何方法，只要提供的参数能让编译器区分被重载的（或多态）方法。

5.7 virtual和override关键字

如果基类中有一个方法，而想在派生类中覆盖它，应该把基类中的方法声明为virtual，它的含义是，让派生类里具有同样名字和参数标识的方法被调用，而不是调用基类中的方法。然后，在派生类中必须用override关键字声明该方法。

如果在派生类里使用了override关键字，但没有把基类中的方法声明为virtual，编译器会将其标记为错误。如果在派生类中创建的方法与基类中的方法名字相同、参数标识也相同，但是没有把该方法声明为override，这也是一个错误。如果在派生类里创建一个方法并且没有把它声明为override，也没有把基类中的方法声明为virtual，程序编译时会发出警告，但能正常工作，正如读者预期的那样，调用的是派生类中的方法。

5.8 在派生类里覆盖方法

假设想从Rectangle类中派生一个新类DoubleRect，它使用两种颜色并偏移几个像素来画矩形。我们把基类中的draw方法声明为virtual。

```
public virtual void draw (Graphics g )   {
  g.DrawRectangle (rpen, x, y, w, h);
}
```

在派生类DoubleRect的构造函数中，为了完成额外的绘制工作，需要创建一个红色画笔。

```
public class DoubleRect:Rectangle          {
private Pen rdPen;
public DoubleRect(int x, int y, int w, int h):
             base(x,y,w,h)  {
       rdPen  = new Pen (Color.Red, 2);
}
```

这意味着新的DoubleRect类需要自己的draw方法，该draw方法仍然可以用父类的draw方法，但需要添加自己的绘制语句。

```
public override void draw ( Graphics g )   {
      base.draw ( g );        //draw one rectangle using parent class
      g.DrawRectangle  (rdPen, x +5, y+5, w, h);
}
```

注意，我们要使用矩形的坐标和尺寸，它们是在 Rectangle 的构造函数里指定的。可以在 DoubleRect 类中保存这些参数的另一副本，或者把基类 Rectangle 中的这些变量由私有的改为保护的。

```
protected int x, y, w, h;
```

图 5.5 给出的是绘制出最终矩形的窗口。

图 5.5 DoubleRect 类

5.9 使用 new 替换方法

当不能把基类方法声明为virtual时，另一种取代基类中方法的方式是在派生类中声明方法时使用 new 关键字。这样做能有效地隐藏基类中同一名字（与参数无关）的方法。这种情况下，不能在派生类调用基类中的同名方法，而且必须把所有的代码都放在替换方法中。

```
public new void draw ( Graphics g )   {
        g.DrawRectangle (rpen, x, y, w, h);
        g.DrawRectangle (rdPen, x +5, y+5, w, h);
}
```

5.10 覆盖 Windows 控件

在 C# 里，使用继承能很容易地根据现有的 Windows 控件创建新的控件。这里要创建一个 TextBox 控件，当按下 Tab 键将焦点移动到控件里时，可以高亮显示所有的文本。在 C# 里，我们只需从 TextBox 类派生出一个新类，就能创建这一新控件。

先用窗口设计器创建一个拥有两个 TextBox 的窗口，然后用 Project | Add User Control 菜单添加一个 HiTextBox 对象。我们将 HiTextBox 改为继承 TextBox，而不是继承 UserControl。

```
public class HiTextBox : Textbox   {
```

在进一步修改这个程序前，先编译它，新的 HiTextBox 控件会出现在开发环境左边的 Toolbox 的底部，可以在创建的任何窗口里生成 HiTextBox 的可视化实例，如图 5.6 所示。

现在我们修改这个类，插入完成高亮显示的代码。

```
public class HiTextBox : System.Windows.Forms.TextBox
{
        private Container components = null;
```

```
//-------------
private void init() {
       //add event handler to Enter event
       this.Enter += new System.EventHandler (highlight);
}
//-------------
//event handler for highlight event
private void highlight(object obj, System.EventArgs e) {
       this.SelectionStart =0;
       this.SelectionLength =this.Text.Length;
}
//-------------
public HiTextBox()          {
       InitializeComponent();
       init();
}
```

图 5.6 Toolbox 中显示了刚创建的新控件，在一个窗体的窗口设计器窗格上显示了 HiTextBox 的一个实例

这就是整个处理过程。我们用大约十行的代码派生了一个新的 Windows 控件，效率是相当高的。在图 5.6 中能看出程序的结果。如果运行该程序，可能首先会想到，原来的 TextBox 和现在的 HiTextBox 会有同样的行为，因为在它们之间按 Tab 键移动焦点时都会高亮显示。这是 C# 中的 TextBox 的“自动高亮显示”属性。尽管如此，如果单击 TextBox 和 HiTextBox，并前后移动焦点，会看到图 5.7 中只有派生的 HiTextBox 一直是高亮显示。

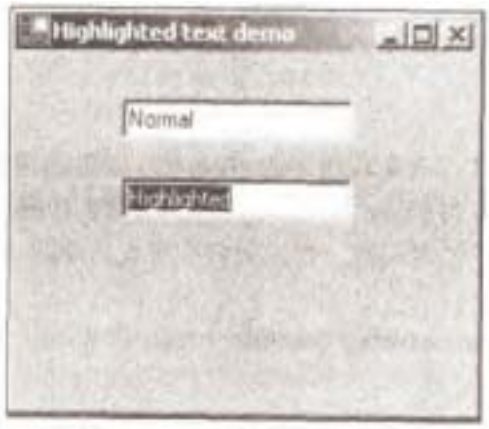


图 5.7 一个新派生的 HiTextBox 控件和一个常规的 TextBox 控件

5.11 接口

接口是一个声明，它规定了一个类要包含的带有指定参数的一系列专门方法。如果一个类含有这样一些方法，就说它实现了接口。接口本质上是一个规约，即一个类应该包含接口所描述的所有方法。接口声明公有方法的签名，但不包含方法主体。

若一个类实现了 Xyz 接口，则可以假定它就是 Xyz 类型来引用这个类，也可以通过它自己的类型来引用它。由于 C# 只允许树形的单继承结构，接口是让一个类有两个或两个以上基类的惟一方式。

考虑一个类的例子，该类提供了一个类似于 ListBox 或一系列复选框的多选择列表的接口。

```
//an interface to any group of components
//that can return zero or more selected items
//the names  are returned in an Arraylist
       public interface Multisel   {
              void clear();
              ArrayList getSelected();
              Panel getWindow();
       }
```

当在具体类里实现接口的方法时，应该声明这个类使用了该接口，还要提供接口中每个方法的实现过程，我们下面举例说明。

```
///ListSel class implements  MultiSel interface
      public class ListSel : Multisel{
            public ListSel() {
            }
            public void clear() {
            }
            public ArrayList getSelected() {
                  return new ArrayList();
            }
            public Panel getWindow() {
                  return new Panel();
            }
      }
```

在讨论生成器模式时会看到如何使用该接口。

5.12 抽象类

抽象类声明了一个或多个没有实现的方法。如果把一个方法声明为抽象的，也要把类声明为抽象的。例如，我们定义一个基类 Shape，该类保存一些数据，并创建一个用于绘图的 Pen 对象。因为不同种类的图形需要不同的绘图过程，所以我们不实现具体的 draw 方法。

```
public abstract class Shape                {
       protected int height, width;
       protected int xpos, ypos;
       protected Pen bPen;
       //-----
       public Shape(int x, int y, int h, int w)       {
              width = w;
              height = h;
              xpos = x;
```

```
            ypos = y;
            bPen = new Pen(Color.Black );
        }
        //-----
        public abstract void draw(Graphics g);
        //-----
        public virtual float getArea() {
            return height * width;
        }
}
```

注意，这里把draw方法声明为抽象的，并且是以分号结束，没有花括号和其中的代码。还要把整个类也声明为抽象的。

不能创建抽象类（如Shape）的实例，只能创建实现了抽象方法的派生类的实例。这里创建一个Rectangle类，它去完成这些工作。

```
public class Rectangle:Shape    {
        public Rectangle(int x, int y, int h, int w):
                base(x,y,h,w) {}
        //-----
        public override void draw(Graphics g) {
            g.DrawRectangle (bPen, xpos, ypos, width, height);
        }
}
```

这是一个完整的、可实例化的类，它拥有真正的draw方法。

再以同样的方式创建一个Circle类，它拥有自己的draw方法。

```
public class Circle :Shape {
        public Circle(int x, int y, int r):
            base(x,y,r,r) {    }
        //-----
        public override void draw(Graphics g) {
            g.DrawEllipse (bPen, xpos, ypos, width, height);
        }
}
```

现在，如果要画圆形和矩形，只要在构造函数所调用的init方法里创建相应的实例。注意，因为它们有同样的基类Shape，所以可将它们都作为Shape对象。

```
public class Form1 : System.Windows.Forms.Form  {
        private PictureBox pictureBox1;
        private Container components = null;
        private Shape rect, circ;
        //-----
        public Form1()      {
            InitializeComponent();
            init();
        }
        //-----
        private void init() {
            rect = new CsharpPats.Rectangle (50, 60, 70, 100);
            circ = new Circle (100, 60, 50);
        }
```

最后，在前面创建的Paint事件处理程序中调用draw方法来绘制这两个对象。

```
private void pictureBox1_Paint( object sender, PaintEventArgs e )   {
        Graphics g = e.Graphics;
        rect.draw ( g );
        circ.draw ( g );
}
```

在图5.8中可以看到程序的执行结果。

图5.8 绘制一个矩形和一个圆形的抽象类系统的输出结果

5.13 接口和抽象类的比较

创建一个接口就是创建了一个或多个方法的定义，在每个实现该接口的类中必须实现这些方法。系统不会生成任何默认的方法代码，必须自己完成实现过程。接口的优点是它提供了一种让一个类成为两个类的子类的方式：一个是继承，一个来自子接口，如果实现该接口的类漏掉了一个接口方法，编译器会产生错误。

创建一个抽象类就是创建了这样一个基类，它可能有一个或多个完整的、可以工作的方法，但至少有一个方法未实现并声明为抽象的。不能实例化一个抽象类，而必须从它派生出类，这些类包含了抽象方法的实现过程。如果一个抽象类的所有方法在基类中都未实现，它在本质上就等同于一个接口，但限制条件是，一个类不能从它继承，也不能继承其他类层次结构，而使用接口则可以这样做。抽象类的作用是对派生类如何工作提供一个基类定义，允许程序员在不同的派生类中填充这些实现过程。

另一种相关方式是创建带空方法的基类，这些空方法可以保证所有的派生类都能够编译，但是每个事件的默认操作是根本什么都不做。下面是这种形式的Shape类。

```
public class NullShape      {
             protected int height, width;
             protected int xpos, ypos;
             protected Pen bPen;
             //-----
             public Shape(int x, int y, int h, int w)     {
                    width = w;
                    height = h;
                    xpos = x;
                    ypos = y;
                    bPen = new Pen(Color.Black );
             }
```

```
        //-----
        public virtual void draw(Graphics g) {}
        //-----
        public virtual float getArea() {
                return height * width;
        }
}
```

注意，draw方法现在是一个空方法。派生类没有编译错误，但不会做任何事情，也没有提示要覆盖哪个方法，而使用抽象类可以得到这样的提示。

5.14 小结

在本章里学习了C#的大部分重要特性。C#提供了继承、构造函数以及重载方法的功能，甚至能创建Windows控件的派生版本。在后面的章节里，会看到如何用C#编写设计模式。

5.15 随书附带光盘中的程序

Rectangle and Square	\Inheritance\RectDraw
DoubleRect	\Inheritance\DoubleRect
A Highlighted TextBox	\Inheritance\Hitext
Abstract Shape	\Inheritance\abstract

第6章 UML图

本书中的模式图都采用统一建模语言（Unified Modeling Language，UML）绘制，这种简单的图示方式是由Grady Booch，James Rumbaugh和Ivar Jacobson在工作中发展而成的，这一思想的出现成了一种规范，最后形成一种标准。可以在很多书中查阅到使用UML的细节，例如由Booch等(1999)，Fowler和Scott(1997)以及Grand(1998)所著的书。本章中我们将概述后面要用到的一些基础知识。

基本的UML图由表示类的框图组成。考虑下面的类（它几乎没有什么实际功能）。

```
public class Person     {
        private string name;
        private int age;
        //-----
        public Person(string nm, int ag)          {
                name = nm;
                age = ag;
        }
        public string makeJob() {
                return "hired";
        }
        public int getAge() {
                return age;
        }
        public vcid splitNames() {
        }
}
```

我们用UML表示这个类，如图6.1所示。

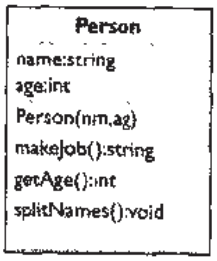
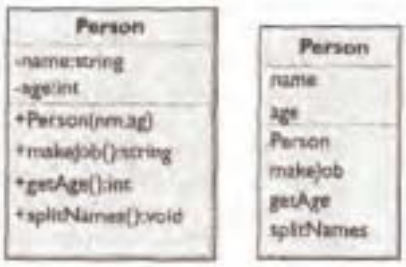


图6.1 Person类，给出了私有变量和公有方法

框图的顶层包含了类名和包名（如果有的话），中间层列举出类中的变量，底层列举出类的方法。名字前面的符号表示成员的可视性，其中“+”表示公有的，“-”表示私有的，“#”表示受保护的。静态方法带有下划线，抽象方法写成斜体或带有“{abstract}”标签。

在必要时，也可以在UML图中给出全部的类型信息，如图6.2(a)所示。

UML不要求给出类的所有属性，通常只给出那些讨论中涉及的部分。例如在图6.2(b)中，就省略了方法的一些细节。



(a)　　(b)

图 6.2　Person 类的 UML 图，图(a)给出了所有类型信息，图(b)没有提供方法的类型

6.1　继承

我们考虑一个 Person 类，它具有公有的、受保护的及私有的变量和方法，以及一个由它派生出的 Employee 类。在基类 Person 中，将方法 getJob 声明为抽象的，也就是说要用关键字 abstract 来声明它。

```
public abstract class Person        {
              protected string name;
              private int age;
              //-----
              public Person(string nm, int ag)        {
                     name = nm;
                     age = ag;
              }
              public string makeJob() {
                     return "hired";
              }
              public int getAge() {
                     return age;
              }
              public void splitNames() {
              }
              public abstract string getJob();   //must override
}
```

接下来由它派生出 Employee 类，并为 getJob 方法填充代码。

```
public class Employee:Person       {
       public Employee(string nm, int ag):base(nm, ag){
       }
       public override string getJob() {
              return "Worker";
       }
}
```

用一条实线和一个空心箭头表示继承。对 Employee 类来说，它是 Person 类的一个子类，我们用 UML 来表示，如图 6.3 所示。

注意，Employee 类的名字没有采用斜体字，因为它是一个具体的类，已为原来的抽象方法 getJob 给出了具体的实现。尽管用箭头向上指向超类来表示继承，已经成为习惯，但 UML 不要求这样做，有时候不同的布局会更清楚，也会更有效地使用空间。

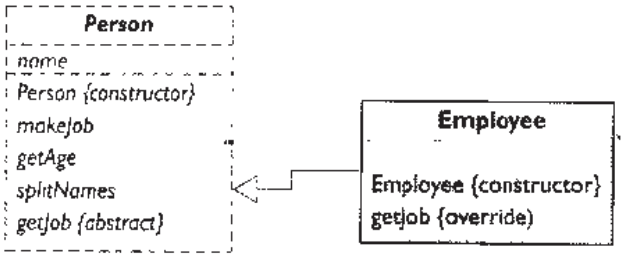


图 6.3 说明 Employee 类派生于 Person 类的 UML 图

6.2 接口

除了箭头后面是虚线外，接口的UML图看起来很像继承，如图6.4所示。也可以给出名字，放在尖括号中（<<interface>>）。

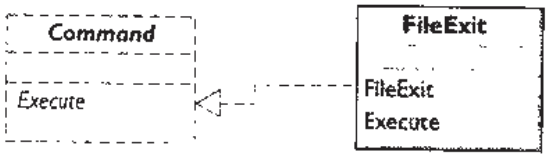


图 6.4 FileExit 实现了 Command 接口

6.3 组合

多数时候，类层次结构的UML表示里应当包括对象如何包含在其他对象里的内容。例如，某个类 Company 可能包括一个 Employee 和一个 Person 实例（可能表示承包人）。

```
public class Company    {
      private Employee emp;
      private Person prs;
      public Company()        {
      }
}
```

用UML来表示这种结构，如图6.5所示。

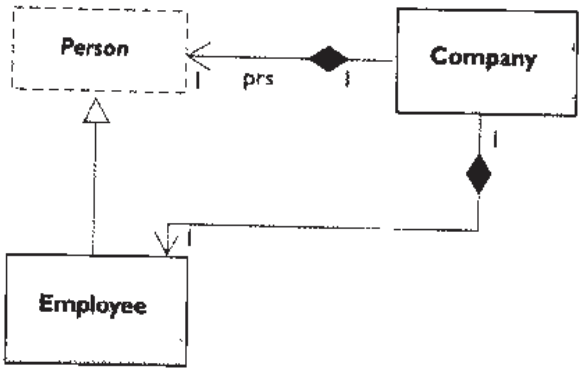


图 6.5 Company 类包含了 Person 和 Employee 实例

类之间的线段表示Company中有0~1个Person实例，0~1个Employee实例。菱形框表示Company类为类的聚集。

如果一个类中有某个类的多个实例，如 Employee 数组，如下所示：

```
public class Company    {
      private Employee[ ]  emps;
      private Employee emp;
      private Person prs;
      public Company()      {
      }
}
```

则我们把对象组合表示成一条标有一个“*”或一个“0”的线段，如图 6.6 所示。

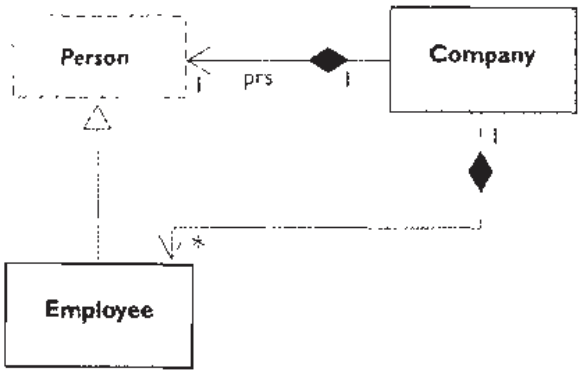


图 6.6　Company 类包含任意数目的 Employee 实例

有些作者用实心菱形箭头表示聚集包含，用圆形箭头表示单个对象组合，但在这里不做要求。

6.4　图注

读者会发现注解 UML 图，或插入解释哪个类调用哪个类的方法的注释都很方便，可以将注释放置在 UML 图中的任何地方。注释可以放在带折角的方框中，或者就写成文本形式。文本注释通常和箭头线段放在一起，表示被调用方法的出处，如图 6.7 所示。也可以用一条线段连接注释和特定的元素。

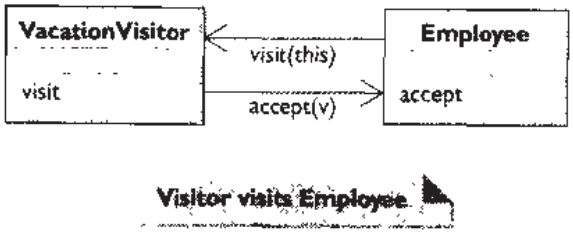


图 6.7　注释通常放在带折角的方框里

UML 是表示程序中对象关系的一种有效方式，在完整的 UML 规范中，有更多的图表属性，但是，我们前面的简要叙述已经覆盖了本书中要使用的标记方法。

6.5　用 WithClass 绘制 UML 图

本书中所有的 UML 图都是用 MicroGold 的 WithClass 程序绘制的，该程序读入被编译过的类，生成这里给出的 UML 类图。我们对多数的类图进行了编辑，只给出重要的方法和关系。尽管如此，每个设计模式的完整的 WithClass 图文件仍保存在相应的图目录中，这样，读者可以运行随书附带光盘中的 WithClass 演示版程序，读入并研究详细的 UML 图，其开头部分是与读者在本书看到的相同的图表。

6.6 C#项目文件

本书中所有的程序都用Visual Studio.NET编写成项目文件。随书附带的光盘中包含了相应的项目文件，读者可以加载项目并进行编译。

第7章　C#中的数组、文件和异常

C#处理数组和文件相当容易，并引入异常来简化错误处理。

7.1　数组

C#中的数组元素从零开始，如果声明一个数组：

```
int[ ] x = new int[ 10];
```

该数组有10个元素，下标从0到9。这样，数组元素就排成了一行，与C，C++及Java中的排列方式一样。

```
const int MAX = 10;
float[ ] xy = new float [ MAX];
for (int i = 0; i < MAX; i++ )   {
       xy[ i] = i;
}
```

应当习惯于数组的循环是从零到数组边界值减1，正如我们在前面的代码中所做的那样。

所有的数组变量都有一个长度属性，可以根据它查出数组的大小。

```
float[ ] z = new float[ 20];
for (int j = 0; j < z.Length; j++)  {
       z[ j] = j;
}
```

C#中的数组是动态的，可以在任何时候重新分配空间。在类内部创建一个数组的引用，然后分配空间，可使用下面的语法：

```
float[ ] z;            //declare here
z = new float [ 20];   //create later
```

7.2　集合对象

System.Collections名字空间包含了许多很有用的可变长数组对象，可以通过几种方式添加、获取项目。

7.2.1　ArrayList

ArrayList（数组表）对象本质上是一个可变长的数组，可以根据需要添加元素。使用ArrayList的方法可以向数组中添加元素，或取出、修改某个元素。

```
float[ ] z = {1.0f, 2.9f, 5.6f};
ArrayList arl = new ArrayList ( );
for (int j = 0; j < z.Length; j++ )  {
       arl.Add (z [ j] );
}
```

ArrayList有一个Count属性，可用于确定它所包含的元素数目。可以从0循环到该数目减1来访问所有元素，这里是把ArrayList当做了一个数组。

```
for (j = 0; j < ar1.Count; j++ ) {
        Console.WriteLine (ar1 [ j] );
}
```

也可以使用foreach循环结构顺序访问ArrayList对象中的成员，此时不需要创建索引变量，也不需要知道ArrayList的长度。

```
foreach (float a in ar1 ) {
        Console.WriteLine (a);
}
```

ArrayList中可供使用的方法如表7.1所示。

表7.1 ArrayList的方法

Clear	删除ArrayList的内容
Contains(object)	如果ArrayList包含该对象，则返回true
CopyTo(array)	将ArrayList全部内容拷贝到一个一维数组中
IndexOf(object)	返回该对象的第一个索引值
Insert(index,object)	将元素插入到指定的位置
Remove(object)	从表中删除该元素
RemoveAt(index)	删除指定位置的元素
Sort	对ArrayList排序

从ArrayList中取出的每个对象都是object类型，这就意味着，在使用前通常需要将其转换成合适的类型。

```
float x = (float) ar1[ j] ;
```

7.2.2 Hashtable

Hashtable（哈希表）是一个可变长数组，表中的每个项目都通过关键字值来访问。关键字一般是某个字符串，也可以是其他类型的对象。尽管元素本身不要求是惟一的，但每个元素必须有一个惟一的关键字。使用Hashtable可以快速访问一个大而无序的记录表，还可以将关键字和项目值颠倒过来，创建一个每条项目都是惟一的表。

```
Hashtable hash = new Hashtable ( );
float freddy = 12.3f;
hash.Add ("fred",freddy);     //add to table
//get this one back out
float temp = (float)hash [ "fred"] ;
```

注意，同ArrayList一样，从Hashtable中得到的数据也必须转换成合适的类型。Hashtable也有一个Count属性，可用来获取关键字的列表或项目的列表。

7.2.3 SortedList

SortedList（排序表）类保存了两个内部数组，可以通过基于零的下标或字母关键字来获取元素。

```
float sammy = 44.55f;
SortedList slist = new SortedList();
slist.Add("fred",freddy);
slist.Add("sam",sammy);
```

```
//get by index
float newFred = (float)slist.GetByIndex(0);
//get by key
float newSam = (float)slist["sam"];
```

在System.Collections名字空间还可以看到Stack（堆栈）对象和Queue（队列）对象，它们的特性与读者期望的一样，可以在系统帮助文档中找到它们的方法。

7.3 异常

C#用异常完成错误处理，而不是采用其他难以使用的错误检测方法。异常处理就是把引起错误的语句包含在try块中，然后用catch语句捕获错误。

```
try {
  //Statements
}
 catch(Exception e){
  //do these if an error occurs
}
 finally {
  //do these anyway
}
```

一般使用这种方法检查文件处理语句的错误，也可用于捕获数组下标越界和许多其他错误情形。它的工作过程是：执行try块中的语句，如果没有错误，控制转移到finally语句（如果有的话），退出该块；如果发生错误，控制转移到catch语句，可以在此处理错误，然后控制转移到finally语句，接着退出块。

下面的例子给出了对异常的检测过程，由于最后一次循环有一个元素超出了ArrayList的范围，出现了一个错误。

```
try {
     //note- one too many
     for (int i = 0; i <= arl.count; i++)
            Console.writeLine(arl[i]);
}
catch(Exception e) {
            Console.writeLine (e.Message);
}
```

这段程序输出了错误信息和程序中的调用位置，然后继续运行。

```
0123456789Index was out of range.
Must be non-negative and less than the size of the collection.
Parameter name:index
   at System.Collections.ArrayList.get_Item(Int32 index)
   at arr.Form1..ctor( ) in form1.cs:line 58
```

相反，如果不捕获异常，运行时会给出一条错误信息，程序将退出运行不再继续下去。

比较常见的一些异常在表7.2中给出。

表7.2 C#中的常用异常类

AccessException	访问类的方法或数据时出错
ArgumentException	传给方法的参数是无效的
ArgumentNullException	参数是空的

（续表）

ArithmeticException	上溢或下溢
DivideByZeroException	被0除
IndexOutOfRangeException	数组下标越界
FileNotFoundException	文件没找到
EndOfStreamException	访问超出了输入数据流（如文件）的末端
DirectoryNotFoundException	目录没找到
NullReferenceException	对象变量没有初始化成一个实际的值

7.4 多个异常

也可以在一系列catch块中捕获多个异常并分别进行处理。

```
try {
     for(int i =0; i<= arl.count; i++) {
            int k = (int)(float)arl[ i];
            Console.Write(i + " "+k/i);
     }
}
catch(DivideByZeroException e){
      printZErr(e);
}
catch(IndexOutofRangeException e) {
      printOErr(e);
}
catch(Exception e) {
      printErr(e);
}
```

这样做可以让程序以不同方式从各种错误中恢复过来。

7.5 抛出异常

在发生异常的地方可以不处理该异常，使用throw语句将异常传递回调用程序，在调用程序中抛出异常。

```
try {
//statements
}
catch (Exception e) {
        throw (e);      //pass on to calling program
}
```

注意，C#不支持Java语法throws。Java中的throws声明一个方法要抛出一个异常，且必须为该异常提供相应的处理程序。

7.6 文件处理

C#中的文件处理对象提供了一些相当灵活的处理文件的方法。

7.6.1 File对象

File（文件）对象代表一个文件，并为检测文件是否存在、文件更名和文件删除提供了一些有用的方法。所有这些方法都是静态的，也就是说，不能使用new运算符创建File的实例，但可以直接使用File的方法。

```
if (File.Exists ("Foo.txt"))
       File.Delete ("foo.txt");
```

也可以用File对象获得用于读写文件数据的文件流（FileStream）。

```
//open text file for reading
       StreamReader ts = File.OpenText ("fool.txt");

//open any type of file for reading
       FileStream fs = File.OpenRead ("foo2.any");
```

下面是一些比较有用的File方法。

静态方法	作用
File.FileExists(filename)	若文件存在，则为true
File.Delete(filename)	删除该文件
File.AppendText(String)	追加文本
File.Copy(fromFile,toFile)	拷贝一个文件
File.Move(fromFile,toFile)	移动文件，删除旧的文件
File.GetExtension(filename)	返回文件的扩展名
File.HasExtension(filename)	若文件有扩展名，则为true

7.6.2 读文本文件

要读一个文本文件，首先使用File对象获取一个StreamReader对象，然后用文本流的读方法读取数据。

```
StreamReader ts = File.OpenText ("fool.txt");
String s =ts.ReadLine ( );
```

7.6.3 写文本文件

要创建一个文本文件并写入数据，可使用CreateText方法获取一个StreamWriter对象。

```
//open for writing
       StreamWriter sw = File.CreateText ("foo3.txt");
       sw.WriteLine ("Hello file");
```

如果向一个已经存在的文件追加数据，可以直接创建一个StreamWriter对象，并将用于追加的那个布尔参数设置为true。

```
//append to text file
StreamWriter asw = new StreamWriter ("fool.txt", true );
```

7.7 文件处理中的异常

大量经常发生的异常是在处理文件的输入和输出时引起的。非法文件名、文件不存在、目录不存在、非法文件名参数及文件保护错误等都会引起异常。因此，处理文件输入和输出的最好方法是将文件处理代码放在try块中，这样可以保证捕获所有可能的错误，避免出现令人为难的致命错误。各种文件类的方法都在其文档中给出了它们能抛出的异常。只要捕获通用Exception对象就能保证捕获所有异常，但是如果需要对不同的异常进行不同的处理，就要对异常分别进行检测。

例如，可以按照下列方式打开一个文本文件。

```
try {
//open text file for reading
       StreamReader ts = File.OpenText("fool.txt");
```

```
        string s =ts.ReadLine();
}
catch(Exception e){
        console.writeLine(e.Message);
}
```

7.8 检测文件末尾

有两种可用的方法能保证不会超过文本文件的末尾：（1）查找空（null）异常，（2）查找数据流的末尾。当读数据超过了文本文件的末尾时，不会出现错误，也不会抛出文件结束的异常。但是，如果在文件末尾后读入一个字符串，会返回一个空值，可以用它在一个文件读入类中创建一个文件结束函数。

```
private StreamReader rf;
private bool eof;
//------------
public String readLine () {
        String s = rf.ReadLine ();
        if(s == null)
                eof = true;
        return s;
}
//-----------
public bool fEof(){
        return eof;
}
```

另外一种保证读数据不会超过文件末尾的方法是，使用Stream的Peek方法在读数据前先查看一下。Peek方法返回文件中下一个字符的ASCII码，如果没有字符则返回-1。

```
public String read_Line(){
        String s = ""
        if(rf.Peek() > 0){
                s = rf.ReadLine();
        }
        else {
                eof = true;
        }
        return s;
}
```

7.9 csFile类

有时候，将文件方法包装在一个带有易用方法的简单类中会更方便。我们在csFile类中完成这样的工作，在后面章节的某些例子里，会用到这个类。

这里把文件名（fileName）和路径包含在构造函数里，也可以用重载的OpenForRead和OpenForWrite语句把它传递进来。

```
public class csFile
        {
                private string fileName;
                StreamReader ts;
                StreamWriter ws;
                private bool opened, writeOpened;
```

```
        //------------
        public csFile() {
                init();
        }
        //------------
        private void init() {
                opened = false;
                writeOpened = false;
        }
        //------------
        public csFile(string file_name) {
                fileName = file_name;
                init();
        }
}
```

有两个方法能够打开用于读数据的文件：一个方法包含了文件的名字，另一个将文件名作为参数。

```
public bool OpenForRead(string file_name){
        fileName = file name;
        try {
                ts = new StreamReader (fileName);
                opened = true;
        }
        catch(FileNotFoundException e) {
                return false;
        }
        return true;
}
//------------
public bool OpenForRead() {
        return OpenForRead(fileName);
}
```

接下来，可以用 readLine 方法从文本文件读数据。

```
public  string readLine() {
         return ts.ReadLine();
}
```

同样，下面的方法可以打开一个用于写数据的文件，并向其中写入若干行文本。

```
public void writeLine(string s) {
        ws.WriteLine (s);
}
//-----------
public bool OpenForWrite() {
        return OpenForWrite(fileName);
}
//-----------
public bool OpenForWrite(string file_name) {
        try{
                ws = new StreamWriter (file_name);
                fileName = file_name;
                writeOpened = true;
                return true;
        }
        catch(FileNotFoundException e) {
```

```
            return false;
        }
    }
```

在任何时候需要读入文件时，都会使用这个简化的文件方法包装类。

7.10 随书附带光盘中的程序

Illustrates csFile class	\Files\FileReader

第二部分

创建型模式

掌握了前面介绍的对象、继承、接口后，我们准备开始正式讨论设计模式。前面曾经提到，设计模式只是编写更好的面向对象程序的一种方法。按照Gang of Four的分类方式，把设计模式分成三类：创建型、结构型和行为型。在这部分讨论创建型模式（Creational Pattern）。

所有的创建型模式都涉及到创建对象实例的方式。这一点很重要，因为程序不应该依赖于对象如何创建和如何安排。当然，使用new运算符是C#创建一个对象实例最简单的方法。

```
Fred fred1 = new Fred();        //instance of Fred class
```

然而，这实际上相当于硬编码，它和程序中如何创建对象有关。很多情况下，创建对象的本意随程序的需求不同而不同，将创建过程抽象成一个专门的“创造器”类，会使程序更灵活、更通用。

工厂方法模式（Factory Method Pattern）提供了一个简单的决策类，它根据提供的数据返回一个抽象基类的多个子类中的一个。工厂模式里，我们先介绍简单工厂模式，然后再介绍工厂方法模式。

抽象工厂模式（Abstract Factory Pattern）提供了一个创建并返回一系列相关对象的接口。

单件模式（Singleton Pattern）是指某个类只能有一个实例。它提供了一个访问该实例的全局访问点。

生成器模式（Builder Pattern）将一个复杂对象的构建与它的表示分开，这样就能根据程序的需要创建不同的表示形式。

原型模式（Prototype Pattern）是先实例化一个类，然后拷贝或克隆该类来创建新的实例，可以用公有方法进一步修改这些实例。

第8章　简单工厂模式

在OO程序中，我们经常看到的一种模式就是简单工厂模式（Simple Factory Pattern）。简单工厂模式根据提供给它的数据，返回几个可能类中的一个类的实例。通常它返回的类都有一个共同的父类和共同的方法，但每个方法执行的任务不同，而且根据不同的数据进行了优化。简单工厂模式实际上不属于23个GoF模式，但它可以作为我们稍后要讨论的工厂方法模式的一个引导。

8.1　简单工厂模式如何工作

为了理解简单工厂模式，我们来看图8.1中的类图。在该图中，X是一个基类，XY类和XZ类从它派生出来，XFactory类根据给出的参数决定返回哪一个子类。在右边定义了一个getClass方法，传递给它某个值（如abc），然后返回类X的某个实例。返回哪一个类的实例与程序员无关，因为这些类有同样的方法，只是实现不同。返回哪一个类的实例完全取决于工厂，工厂功能可能很复杂，但通常都是相当简单的。

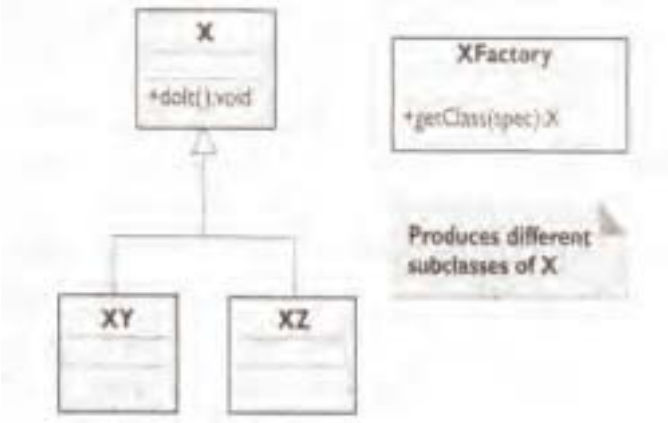


图8.1　一个简单工厂模式

8.2　示例代码

接下来考虑一个简单的C#例子，在它里面用到了Factory类。假设我们有一个输入窗体，允许用户输入他的姓名，可以是“firstname lastname”形式或“lastname，firstname”形式。现在进一步简化这个问题，假设我们总是能通过判断lastname和firstname之间是否有逗号来确定姓名的顺序。

这是一个相当容易做出的判断，可以在类中用一个简单的if语句实现，但是，我们这里是用它说明工厂如何工作以及工厂能生成哪些对象。先定义一个简单的类，它用构造函数接收表示名字的字符串，并且可以从类中取回名字。

```
//Base class for getting split names
      public class Namer    {
            //parts stored here
            protected string frName, lName;
```

```
        //return first name
        public string getFrname(){
                return frName;
        }
        //return last name
        public string getLname() {
                return lName;
        }
}
```

注意，在这个基类里没有设置名字的构造函数。

8.3　两个派生类

接下来编写两个非常简单的派生类，它们实现了接口，并在构造函数中将名字分成了两部分。在FirstFirst类中，做了一个简单的假设：最后一个空格前面的所有部分都属于firstname。

```
public class FirstFirst : Namer {
        public FirstFirst(string name) {
                int i = name.Trim().IndexOf (" ");
                if(i > 0) {
                        frName = name.Substring (0, i).Trim ();
                        lName = name.Substring (i + 1).Trim ();
                }
                else {
                        lName = name;
                        frName = "";
                }
        }
}
```

在LastFirst类里，用逗号给lastname划分界限。当空格或逗号不存在时，两个类里都提供了错误校正处理。

```
public class LastFirst : Namer    {
        public LastFirst(string name)        {
                int i = name.IndexOf (",");
                if(i > 0) {
                        lName = name.Substring (0, i);
                        frName = name.Substring (i + 1).Trim ();
        }
        else {
                lName = name;
                frName = "";
        }
    }
}
```

两种情况下，我们把拆分后的名字分别保存在基类Namer中的保护变量lName和frName里。注意，在派生类里根本不需要任何getFrname和getLname方法，因为已经在基类里给出来了。

8.4　构造简单工厂

现在很容易给出简单工厂类。只检测逗号是否存在，然后返回其中的一个类的实例。

```
public class NameFactory    {
        public NameFactory() {}

        public Namer getName(string name) {
                int i = name.IndexOf (",");
                if(i > 0)
                        return new LastFirst (name);
                else
                        return new FirstFirst (name);
        }
}
```

8.4.1 使用工厂

接下来看看怎样把上述部分组合在一起。为了响应Compute按钮单击事件，用一个NameFactory实例返回正确的派生类。

```
private void btCompute_Click(
                object sender, System.EventArgs e){
        Namer nm = nameFact.getName(txName.Text);
        txFirst.Text = nm.getFrname();
        txLast.Text = nm.getLname();
}
```

然后调用getFrname方法和getLname方法获得正确拆分的名字。这里不需要知道使用的是哪一个派生类，工厂为我们提供了这个类，所需要知道的就是它有两个取出（get）方法。完整的类图如图 8.2 所示。

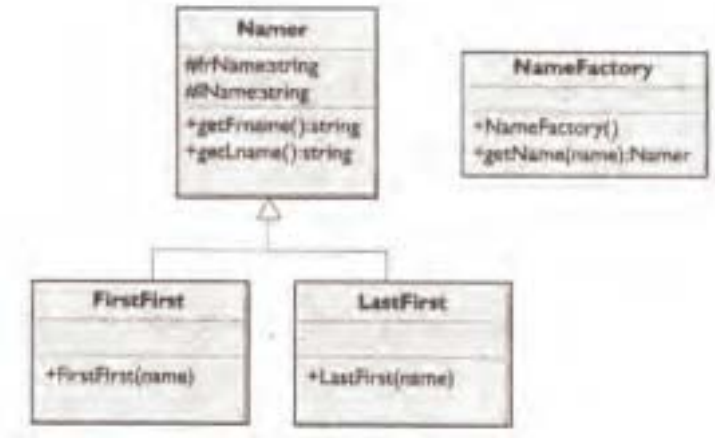


图 8.2 NameFactory 程序

我们构建了一个简单用户接口，允许用户以两种方式输入名字，然后显示拆分后的名字。在图 8.3 中可以看到该程序的运行结果。

输入一个名字，然后单击Compute按钮，拆分后的名字显示在下面的文本域中。该程序的关键之处在于 Compute 方法，它取回输入的文本，获得 Namer 类的实例，然后显示结果。

这就是简单工厂模式的基本原理。创建了一个抽象工厂，它决定返回哪一个类的实例并将该实例返回。接下来可以调用那个类实例的方法，但不需要知道具体使用的是哪一个子类，这种方法把和数据相关的问题与类的其他方法分隔开来。

图 8.3　Namer 程序运行示例

8.5　数学计算中的工厂模式

大多数使用工厂模式的人都认为，工厂模式可以作为简化杂乱的编程类的工具。然而，工厂模式用在执行数学运算的程序里也是非常合适的。例如，在快速傅里叶变换（FFT）中，下列四个方程式要对大量的点对重复执行许多遍，才能完成转换。根据这些计算结果绘制的图形形状，将下面四个方程式组成 FFT "Butterfly（蝴蝶）" 类的一个实例。四个方程式如下所示：

$$R_1' = R_1 + R_2\cos(y) - I_2\sin(y) \quad (1)$$
$$R_2' = R_1 - R_2\cos(y) + I_2\sin(y) \quad (2)$$
$$I_1' = I_1 + R_2\sin(y) + I_2\cos(y) \quad (3)$$
$$I_2' = I_1 - R_2\sin(y) - I_2\cos(y) \quad (4)$$

当角度 y 为零时，每次遍历数据时，仍然需要多次计算。这种情况下，可将复杂的数学计算简化为公式(5) ~ (8)：

$$R_1' = R_1 + R_2 \quad (5)$$
$$R_2' = R_1 - R_2 \quad (6)$$
$$I_1' = I_1 + I_2 \quad (7)$$
$$I_2' = I_1 - I_2 \quad (8)$$

我们先定义一个保存复数的类。

```
public class Complex    {
      float real;
      float imag;
//--------------------------------
public Complex(float r, float i) {
      real = r; imag = i;
}
//--------------------------------
public void setReal(float r) { real = r;}
//--------------------------------
public void setImag(float i) { imag= i;}
//--------------------------------
public float getReal() { return real;}
//--------------------------------
```

```
public float getImag() { return imag;}
}
```

基类 Butterfly 是一个抽象类，任何一个具体类都要填充 Execute 命令的实现过程。

```
public abstract class Butterfly   {
      float y;
      public Butterfly() {
      }
      public Butterfly(float angle) {
            y = angle;
      }
      abstract public void Execute(Complex x, Complex y);
}
```

接下来在Butterfly的基础上创建一个简单的加法类，它实现了公式(5)~(8)的加法和减法运算。

```
class AddButterfly : Butterfly  {
   float oldr1, oldi1;
      public AddButterfly(float angle) {
            }
      public override void Execute(Complex xi, Complex xj) {
            oldr1 = xi.getReal();
            oldi1 = xi.getImag();
            xi.setReal(oldr1 + xj.getReal());
            xj.setReal(oldr1 - xj.getReal());
            xi.setImag(oldi1 + xj.getImag());
            xj.setImag(oldi1 - xj.getImag());
      }
}
```

TrigButterfly 类与此类似，只是 Execute 方法包含了公式(1)~(4)中的三角函数的计算。

```
public class TrigButterfly:Butterfly   {
       float y, oldr1, oldi1;
       float cosy, siny;
       float r2cosy, r2siny, i2cosy, i2siny;

       public TrigButterfly(float angle) {
             y = angle;
             cosy = (float) Math.Cos(y);
             siny = (float) Math.Sin(y);

       }
       public override void Execute(Complex xi, Complex xj) {
             oldr1 = xi.getReal();
             oldi1 = xi.getImag();
             r2cosy = xj.getReal() * cosy;
             r2siny = xj.getReal() * siny;
             i2cosy = xj.getImag()*cosy;
             i2siny = xj.getImag()*siny;
             xi.setReal(oldr1 + r2cosy +i2siny);
             xi.setImag(oldi1 - r2siny +i2cosy);
             xj.setReal(oldr1 - r2cosy - i2siny);
             xj.setImag(oldi1 + r2siny - i2cosy);
       }
}
```

然后创建一个简单工厂类，它决定返回哪一个类实例。因为已经创建了 Butterfly 类，所以，我们把工厂类称为 Cocoon（茧）。这里不需要实例化 Cocoon，所以将它的方法声明为静态的。

```
public class Cocoon   {
       static public Butterfly getButterfly(float y) {
              if (y != 0)
                     return new TrigButterfly(y);
              else
                     return new addButterfly(y);
       }
}
```

8.6　小结

我们已经知道，简单工厂能返回具有同样方法的类的实例，它们可以是不同的派生子类的实例，也可以是实际上毫无关系仅仅是共享了相同接口的类。不管哪一种形式，这些类实例中的方法必须是相同的，并且能够被交替使用。

8.7　思考题

1. 考虑一个类似于 Quicken 的私人支票簿管理程序。它管理几个银行账号和资产，并能处理账单支付。设计这样一个程序时，能够在哪些地方使用工厂模式？
2. 假设你要编写一个程序，帮助房屋主人设计房子的配套设施。可以使用工厂生成哪些对象？

8.8　随书附带光盘中的程序

The Name Factory	`\Factory\Namer`
An FFT Example	`\Factory\FFT`

第9章　工厂方法模式

我们在前面已经看了两个简单工厂的例子，工厂的概念反复出现在面向对象程序设计中，在C#本身和其他设计模式（例如生成器模式）中，就能找到几个例子。在这些例子中，有一个类负责决定在单继承体系结构中实例化哪一个子类。

工厂方法模式（Factory Method Pattern）对这种思想进行了巧妙的扩展，它不是用一个专门的类来决定实例化哪一个子类。相反，超类把这种决定延迟到每个子类。这种模式实际上没有决策点，即没有直接选择一个子类实例化的决策。按这种模式编写的程序定义了一个抽象类，它去创建对象，但让子类决定创建哪一种对象。

这里考虑一个相当简单的例子，在游泳比赛中为运动员确定泳道。在一个赛事中，游泳选手完成几次预赛后，按照前面预赛中最慢的到最后预赛中最快的顺序，对运动员的成绩进行排序，在接下来的比赛中，把游得最快的选手安排在中央泳道上。这种确定泳道的方式称为直接排位。

目前，游泳选手参加锦标赛时，通常游两次。每个选手都参加预赛，前12名或前16名选手会在决赛上再比一次。为了使预赛更公平，对预赛循环排位：最快的三名选手安排在最快的三组里的中央泳道上，第二快的三名选手排在头三组的紧邻中央的泳道上，依次类推。

我们怎样构建对象才能实现这种泳道分配机制并说明工厂方法模式呢？首先，设计一个抽象类Event。

```
public abstract class Event        {
       protected int numLanes;
       protected ArrayList swimmers;

       public Event(string filename, int lanes) {
            numLanes = lanes;
            swimmers = new ArrayList();
            //read in swimmers from file
             csFile f = new csFile(filename);
            f.OpenForRead ();
            string s = f.readLine();
            while (s != null) {
                  Swimmer sw = new Swimmer(s);
                  swimmers.Add (sw);
                  s = f.readLine();
            }
            f.close();
       }
       public abstract Seeding getSeeding();
       public abstract bool isPrelim();
       public abstract bool isFinal();
       public abstract bool isTimedFinal();
 }
```

注意，该类并非完全没有内容。由于所有的派生类都需要从文件读数据，我们把完成该工作的代码放在这个基类中。

这些抽象方法表明了具体Event类应该实现的部分。接下来就由Event类派生两个具体的类，分别为PrelimEvent类和TimedFinalEvent类。这两个类惟一的差别是，一个类返回一种泳道分配方式，另一个类返回另一种泳道分配方式。

我们还定义了一个带有如下方法的抽象类Seeding。

```
public abstract class Seeding    {
             protected int        numLanes;
             protected int[]      lanes;
             public abstract IEnumerator getSwimmers();
             public abstract int getCount();
             public abstract int getHeats();
             protected abstract void seed();
             //-------------------------------
             protected void calcLaneOrder() {
             //complete code on CD
             }
}
```

注意，实际上在calcLaneOrder方法里有代码，但为了简化省略了。派生类通过创建基类Seeding的一个实例来调用这些函数。

接下来创建两个具体的Seeding子类：StraightSeeding类和CircleSeeding类。PrelimEvent类会返回一个CircleSeeding的实例，而TimedFinalEvent类会返回一个StraightSeeding的实例。这样，我们有两个继承体系结构：一个是关于Event的，一个是关于Seeding的。

在Event继承结构中（图9.1），会看到两个Event的派生类，它们都含有getSeeding方法，一个返回一个StraightSeeding实例，另一个返回一个CircleSeeding的实例。可以看出，它与我们前面的简单例子不同，没有实际的工厂决策点。实例化哪一个Event类的决策就是决定实例化哪一个Seeding类的决策。

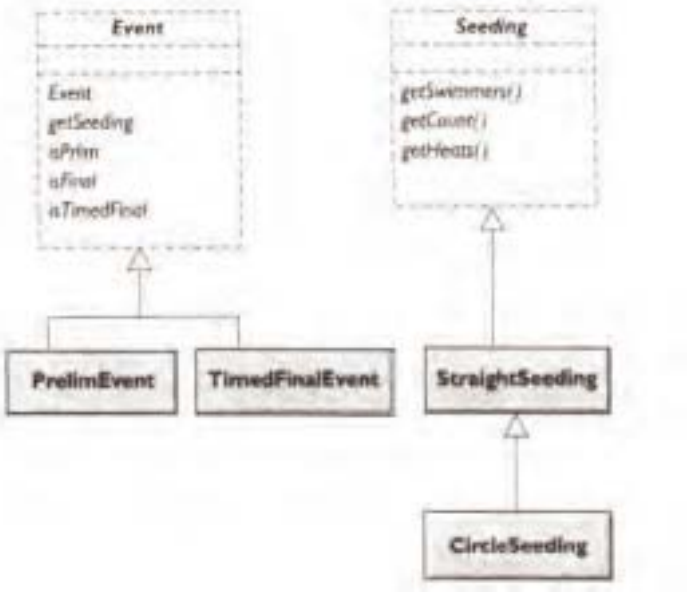


图9.1　给出Seeding接口和派生类的Seeding类图

在两个类继承体系结构中，尽管看起来有一对一的对应关系，但这不是必须的。可以有许多种Event类，而只使用几种Seeding类。

9.1 Swimmer 类

除了说过 Swimmer 类含有名字、俱乐部、年龄、排位、时间以及在选拔赛后的分组和泳道外，对 Swimmer 类我们并没有介绍太多。Event 类从某个数据库（在例子中用的是一个文件）读入 Swimmer 数据，然后在为某项赛事调用 getSeeding 方法时，将数据传给 Seeding 类。

9.2 Event 类

我们在前面已经看到了抽象基类 Event。在实际应用中，用它读入选手数据，并将其传给 Swimmer 类的一个实例去进行分析。在抽象基类 Event 里，判断赛事是预赛、决赛还是计时决赛的方法都是空的，在派生类中会实现这些方法。

PrelimEvent 类返回 CircleSeeding 的一个实例。

```
public class PrelimEvent:Event   {
      public PrelimEvent(string filename, int lanes):
             base(filename,lanes) {}
             //return circle seeding
      public override Seeding getSeeding() {
            return new CircleSeeding(swimmers, numLanes);
      }
      public override bool isPrelim() {
            return true;
      }
      public override bool isFinal() {
            return false;
      }
      public override bool isTimedFinal() {
            return false;
      }
}
```

TimedFinalEvent 类返回 StraightSeeding 的一个实例。

```
public class TimedFinalEvent:Event  {

      public TimedFinalEvent(string filename,
             int lanes):base(filename, lanes) {}
      //return StraightSeeding class
      public override Seeding getSeeding() {
             return new StraightSeeding(swimmers, numLanes);
      }
      public override bool isPrelim() {
             return false;
      }
      public override bool isFinal() {
             return false;
      }
      public override bool isTimedFinal() {
             return true;
      }
}
```

在这两段代码中，两种赛事类都包含基类 Event 的一个实例，用它读取数据文件。

9.3　直接排位

实际编写这个程序时，我们发现大多数工作都是在直接排位（StraightSeeding）中完成的，为循环排位（CircleSeeding）而进行的改动相当少。实例化StraightSeeding类，并拷入选手数据和泳道号。

```
protected override void seed() {
      //loads the swmrs array and sorts it
      sortUpwards();

      int lastHeat = count % numLanes;
      if (lastHeat < 3)
            lastHeat = 3;    //last heat must have 3 or more
      int lastLanes = count - lastHeat;
      numHeats = count / numLanes;
      if (lastLanes > 0)
            numHeats++;
      int heats = numHeats;
      //place heat and lane in each swimmer's object
      //Add in last partial heat
      //copy from array back into ArrayList
      //details on CDROM
}
```

这段程序的目的是，在调用getSwimmer方法时，使选手排位的整个数组都是可用的。

9.3.1　循环排位

CircleSeeding类从StraightSeeding类派生，因此一开始先调用父类的seed方法，然后重新安排比赛。

```
protected override void seed() {
      int circle;
      base.seed();          //do straight seed as default
      if (numHeats >= 2 ) {
            if (numHeats >= 3)
                  circle = 3;
            else
                  circle = 2;
            int i = 0;
      for (int j = 0; j < numLanes; j++) {
            for (int k = 0; k < circle; k++) {
                  swmrs[ i] .setLane(lanes[ j] );
                  swmrs[ i++] .setHeat(numHeats - k);
            }
      }
}
```

9.4　排位程序

在这个例子中，我们从网上拿到了参加500 yd（码，1 yd = 0.914 4 m）和100 yd自由泳比赛的选手数据，并使用这些数据构建TimedFinalEvent类和PrelimEvent类。在图9.2里，会看到这两种排位方式的结果。在上面的图中，选择的是500 Free事件，可以看到选手按照由快到慢的顺序直接安排泳道。在下面的图中，选择的是100 Free事件，进行的是循环排位。

图 9.2 500 Free 的直接排位和 100 Free 的循环排位

9.5 其他工厂

我们在前面略过了一个问题：读入运动员数据的程序如何决定生成哪一项赛事。在我们读入数据时，通过创建正确类型的事件来巧妙地实现这一点。这部分代码在窗体的 init 方法中。

```
private void init() {
            //create array of events
            events = new ArrayList ();
            lsEvents.Items.Add ("500 Free");
            lsEvents.Items.Add ("100 Free");
            //and read in their data
            events.Add (new TimedFinalEvent ("500free.txt", 6));
            events.Add (new PrelimEvent ("100free.txt", 6));
      }
```

很明显，这是一个需要 EventFactory 来决定生成哪一项赛事的例子，这又涉及到了我们开始时所讨论的简单工厂。

9.6 何时使用工厂方法

在下列情况下，应该考虑使用工厂方法：

- 一个类无法预测它要创建的对象属于哪一个类。
- 一个类用它的子类来指定所创建的对象。
- 把要创建哪一个类的信息局部化的时候。

对于实现工厂模式，还需要考虑几个问题：

1. 基类是一个抽象类，模式必须返回一个完整的可工作的类。
2. 基类包含默认方法，除非默认方法不能胜任，才会调用这些方法。
3. 可以将参数传递给工厂，告诉工厂返回哪一个类型的类。这种情况下，类可以共享相同的方法名，但完成的工作可以不同。

9.7　思考题

泳道的排位是从里到外进行的，为了完成类似于泳道的排位工作，还需要创建哪些类？

9.8　随书附带光盘中的程序

Seeding Program	\FactoryMethod\Seeder

第10章　抽象工厂模式

抽象工厂模式（Abstract Factory Pattern）比工厂模式具有更高层次的抽象性。当要返回一系列相关类中的某一个，而每个类都能根据需要返回不同的对象时，可以使用这种模式。换句话说，抽象工厂是一个工厂对象，它能返回一系列相关类中的一个类。可以用简单工厂决定返回哪一个类。

通常认为，试验式的例子应该引用汽车制造厂。我们希望丰田汽车厂完全使用丰田的配件，而福特汽车厂完全使用福特的配件。可以把每个汽车制造厂设想为抽象工厂，配件作为一组相关的类。

10.1　花园规划工厂

我们考虑一个实际的例子，在应用程序里要用到抽象工厂。假设读者编写一个程序来完成花园的规划，这可能是季生植物型花园、蔬菜型花园或多年生植物型花园。不管读者规划哪一种类型的花园，都会遇到同样的问题：

1. 边缘种植什么植物？
2. 中央种植什么植物？
3. 阴凉部分种植什么？

（可能还会有其他许多种植问题，我们在这里不予考虑。）

这里想设计一个C#基类Garden，用类中的方法回答这些问题。

```
public class Garden {
        protected Plant center, shade, border;
        protected bool showCenter, showShade, showBorder;
        //select which ones to display
        public void setCenter() {showCenter = true;}
        public void setBorder() {showBorder =true;}
        public void setShade() {showShade =true;}
        //draw each plant
        public void draw(Graphics g) {
                if (showCenter) center.draw (g, 100, 100);
                if (showShade) shade.draw (g, 10, 50);
                if (showBorder) border.draw (g, 50, 150);
        }
}
```

Plant对象设置植物名字，在draw方法被调用时会绘制自己。

```
public class Plant    {
        private string name;
        private Brush br;
        private Font font;

        public Plant(string pname) {
```

```
            name = pname;      //save name
            font = new Font ("Arial", 12);
            br = new SolidBrush (Color.Black );
        }
        //-------------
        public void draw(Graphics g, int x, int y) {
            g.DrawString (name, font, br, x, y);
        }
    }
```

用设计模式术语来讲，Garden 接口就是抽象工厂。它定义了具体类中的方法，并返回一系列相关类中的某个类。这里将中央植物、边缘植物和喜阴植物作为返回的三个相关类。抽象工厂也能返回更具体的花园信息，例如土壤的 PH 值或灌溉需求等。

在实际系统中，规划每一种类型的花园都要查阅一个详尽的植物信息库，而在这个简单的例子里，每类植物只给出一种。例如，对于蔬菜型花园，只给出下列几种植物：

```
public class VeggieGarden : Garden    {
        public VeggieGarden() {
                shade = new Plant("Broccoli");
                border = new Plant ("Peas");
                center = new Plant ("Corn");
        }
}
```

我们用类似的方法创建 Garden 类的两个子类：PerennialGarden 和 AnnualGarden。因为每个具体类都实现了父类中的方法，所以都可以看做一个具体工厂。现在我们有了一系列的 Garden 对象，每一个对象都能创建一类 Plant 对象。在图 10.1 中的类图说明了这一点。

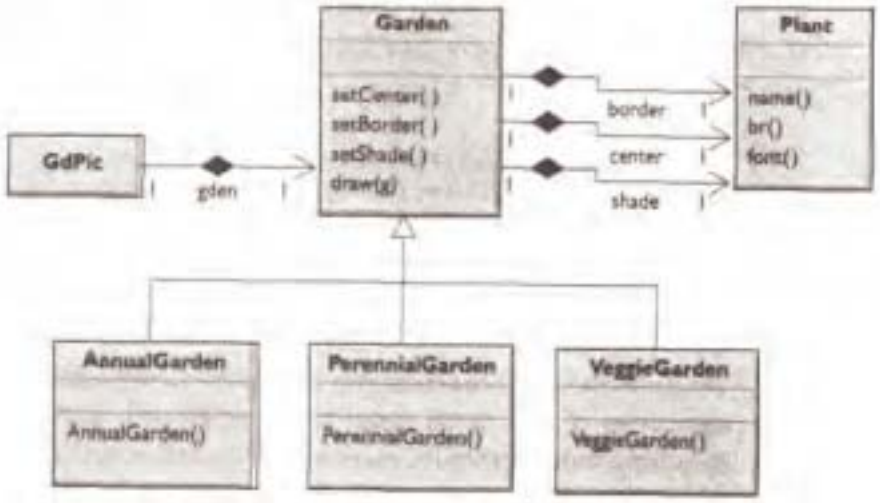


图 10.1　Gardener 程序中的主要对象

我们很容易构建出抽象工厂的驱动程序，它根据用户选择的单选按钮返回一个 Garden 对象，用户界面如图 10.2 所示。

每次选择一种新的花园类型时，都要清除屏幕，将复选框置为未选中状态。然后，选择一个复选框，画出相应的植物类型。

记住，C# 不需要你在程序里直接对屏幕绘图，而是在 paint 事件发生时由系统刷新屏幕，你只要告诉 paint 例程绘制什么对象。

图 10.2 Gardener 程序的用户界面

由于每种花园（及植物）都要知道如何绘制自己，所以应该有一个 draw 方法，在屏幕上画出相应的植物名。由于我们用复选框来说明要画的植物类型，所以设置了一个布尔型变量，用它指出要画的每种植物类型。

Garden 对象包含了三个设置方法，用于指出绘制的每一种类型的植物。

```
public void setCenter ( ) { showCenter = true; }
public void setBorder ( ) { showBorder = true; }
public void setShade ( ) { showShade = true; }
```

10.1.1 图片框

我们在图片框（PictureBox）里用圆表示阴影区域，在图片框中还要给出植物的名字。从 PictureBox 类派生出一个新类 GardenPic，并将画圆和花园植物名等信息传递给它，这是完成上述任务的最好方法。因而，不仅需要在 GardenMaker 窗口类中添加一个 Paint 方法，还要在该窗口所包含的 PictureBox 类中也添加该方法。它覆盖了基类 Control 中的 OnPaint 事件。

```
public class GdPic : System.Windows.Forms.PictureBox {
      private Container components = null;
      private Brush br;
      private Garden gden;
      //---------
      private void init () {
            br = new SolidBrush (Color.LightGray );
      }
      //---------
      public GdPic()  {
            InitializeComponent();
            init();
      }
      //---------
      public void setGarden(Garden garden) {
            gden = garden;
      }
      //---------
      protected override void OnPaint ( PaintEventArgs pe ){
            Graphics g = pe.Graphics;
            g.FillEllipse (br, 5, 5, 100, 100);
            if(gden != null)
                  gden.draw (g);
      }
```

注意，因为OnPaint方法只在整个图片需要重新绘制的时候才调用，所以不必每次都清除植物名的文本。

10.1.2　处理单选按钮和按钮事件

单击三个单选按钮中的某一个，就会创建一个该类型的花园，并把它传给图片框，还要清除所有的复选框。

```
private void opAnnual_CheckedChanged(
             object sender, EventArgs e) {
      setGarden( new AnnualGarden ());
}
//-----
private void opVegetable_CheckedChanged(
             object sender, EventArgs e) {
      setGarden( new VeggieGarden ());
}
//-----
private void opPerennial_CheckedChanged(
             object sender, EventArgs e) {
      setGarden( new PerennialGarden ());
}
//-----
private void setGarden(Garden gd) {
      garden = gd;                //save current garden
      gdPic1.setGarden ( gd);     //tell picture bos
      gdPic1.Refresh ();          //repaint it
      ckCenter.Checked =false;    //clear all
      ckBorder.Checked = false;   //check
      ckShade.Checked = false;    //boxes
}
```

单击一个复选框去显示相应的植物名，要调用对应类型的花园方法去设置要显示的植物名，然后调用图片框的Refresh方法使之重画。

```
private void ckCenter_CheckedChanged(
             object sender, System.EventArgs e) {
      garden.setCenter ();
      gdPic1.Refresh ();
}
//-----
private void ckBorder_CheckedChanged(
             object sender, System.EventArgs e) {
      garden.setBorder();
      gdPic1.Refresh ();
}
//-----
private void ckShade_CheckedChanged(
             object sender, System.EventArgs e) {
      garden.setShade ();
      gdPic1.Refresh ();
}
```

10.2　添加更多的类

抽象工厂的一个强大功能是很容易添加新的子类。例如，如果需要一个GrassGarden类或WildFlowerGarden类，则通过子类化Garden就可以创建这些类。在现有代码中，惟一需要读者真正改动的是添加某种方法来选择这些新类型的花园。

10.3 抽象工厂的效果

抽象工厂的一个主要目的是它能隔离要生成的具体类。这些类的实际类名隐藏在工厂里，在客户端根本不需要知道。

由于类的隔离，可以自由改动或交换这些生成类系列。此外，由于只生成一类具体的类，系统会避免读者误用不同生成系列中的类。但是，添加新的类系列要费一些心思，因为读者要定义一些新的、无二义性的条件使工厂能够返回新的类系列。

尽管抽象工厂生成的所有类都具有相同的基类，但还是无法避免某些子类具有额外的、与其他类不同的方法。例如，BonsaiGarden 类有一个 Height 或 WateringFrequency 方法，这是其他类所没有的。在其他子类中也存在同样的问题：你不知道能否调用一个类方法，除非知道该子类是否允许使用那些方法。这个问题有两种解决方案：或者在基类中定义所有的方法，即使这些方法没有实际的功能；或者是再派生出一个新的基类接口，它包含读者需要的所有方法和所有花园类型的子类。

10.4 思考题

如果读者要编写一个程序记录各种投资，比如股票、债券、期货、易货等，如何用抽象工厂去完成该程序？

10.5 随书附带光盘中的程序

The Gardener Program	`\AbstractFactory\GardenPlanner`

第11章 单件模式

尽管在某种程度上，单件模式（Singleton Pattern）是限制而不是改进类的创建，但它仍和其他创建型模式分在一组。单件模式可以保证一个类有且只有一个实例，并提供一个访问它的全局访问点。在程序设计过程中，有很多情况需要确保一个类只能有一个实例。例如，系统中只能有一个窗口管理器、一个打印假脱机，或者一个数据引擎的访问点。PC机中可能有几个串口，但只能有一个COM1实例。

11.1 使用静态方法创建单件

让一个类只有一个实例，最容易的方法是在类中嵌入一个静态变量，并在第一个类实例中设置该变量，而且每次进入构造函数都要做检查。不管类有多少个实例，静态变量只能有一个实例。为了防止类被多次实例化，我们把构造函数声明为私有的，这样只能在类的静态方法里创建一个实例。下面创建一个getSpooler方法，它返回Spooler的一个实例，如果类已经被实例化，返回值为空（null）。

```
public class Spooler  {
       private static bool instance_flag= false;
       private Spooler()  {
       }
       public static Spooler getSpooler() {
              if (! instance_flag)
                     return new Spooler ();
              else
                     return null;
       }
 }
```

这种方法的主要优点是，如果单件已经存在，不需要考虑异常处理，因为只不过从getSpooler方法返回了一个空值而已。

```
Spooler sp = Spooler.getSpooler();
if (sp1 != null)
       Console.WriteLine ("Got 1 spooler");
Spooler sp2 = Spooler.getSpooler ();
if (sp2 == null)
       Console.WriteLine ("Can\'t get spooler");
}
```

如果试图直接创建Spooler类的实例，编译会失败，因为构造函数被声明为私有的。

```
//fails at compiler time
Spooler sp3 = new Spooler ( );
```

最后，如果需要修改程序，允许该类有两个或三个实例，则修改Spooler类可以很容易实现这一点。

11.2 异常与实例

前面的方法有个缺点，需要程序员检查getSpooler方法的返回值，以确保它不是空的。让程序员始终记得去检查错误的设想是招致失败的开始，应该尽力避免。

我们换种方法做，创建一个这样的类，如果试图多次实例化该类，它会抛出一个异常，这时才需要程序员采取行动，因而这是一个安全的方法。这里先为这个例子创建自己的异常类。

```
public class SingletonException:Exception     {
      //new exception type for singleton classes
      public SingletonException (string s ):base (s)   {
      }
}
```

注意，除了通过base调用了父类的构造函数外，新的异常类实际上什么也没做。尽管如此，拥有自己命名的异常类还是很方便的：在我们创建一个Spooler的实例时，如果抛出了这种类型的异常，系统会警告我们。

11.3 抛出异常

接下来给出PrintSpooler类的框架。这里略去了所有的打印方法，只集中于正确实现单件模式。

```
public class Spooler     {
      static bool instance_flag = false; //true if one instance
      public Spooler()  {
      if (instance_flag)
            throw new SingletonException(
                  "Only one printer allowed");
            else
                  instance_flag=true;      //set flag
                  Console.WriteLine ("printer opened");
            }
      }
}
```

11.4 创建一个类实例

我们已经在PrintSpooler类里创建了一个简单的单件模式，接下来就去了解如何使用它。记住，应该把可能抛出异常的每个方法都包含在try-catch块中。

```
public class singleSpooler {
      static void Main(string[] args) {
            Spooler pr1, pr2;
//open one printer--this should always work
  Console.WriteLine ("Opening one spooler");
    try {
     pr1 = new Spooler();
     }
     catch (SingletonException e)     {
        Console.WriteLine (e.Message);
}
     //try to open another printer --should fail
     Console.WriteLine ("Opening two spoolers");
      try{
```

```
            pr2 = new Spooler();
            }
            catch (SingletonException e) {
                    Console.WriteLine (e.Message);
        }
}
```

然后执行该程序，会得到下列结果：

```
Opening one spooler
printer opened
Opening two spoolers
Only one spooler allowed
```

正如我们所料，最后一行的输出表明抛出了一个异常。

11.5 提供一个单件的全局访问点

由于使用单件可以提供一个类的全局访问点，即使C#中没有全局变量，设计程序时也必须为整个程序提供引用单件的方法。

一种解决方案是在程序的开头创建单件，并将其作为参数传递到需要使用它的类中。

```
prl = iSpooler.Instance ( );
Customers cust = new Customers (prl);
```

这种方法的缺点是，在某次程序运行中，可能不需要所有的单件，这样会影响程序的性能。

另一种更灵活的解决方案是，在程序中创建一个所有单件类的注册表，并使注册表始终是可用的。每次实例化一个单件，都将其记录在注册表中。程序的任何部分都能使用标识符串访问任何一个单件实例，并能取回相应的实例变量。

注册表方法的缺点是减少了类型检查，因为注册表中的单件表可能把所有的单件都保存成对象类型，例如，Hashtable中的对象类型。另外，注册表本身也可能是一个单件，必须使用构造函数或其他 set 函数把它传递给程序的所有部分。

提供一个全局访问点的最常用方式是使用类的静态方法。类名始终是可用的，静态方法只能由类调用，不能由类的实例调用，所以，不管程序中有多少个地方调用该方法，永远只能有一个这样的类实例。

11.6 单件模式的其他效果

1. 子类化一个单件很难，因为只有在基类单件没有被实例化时，才能实现这一点。
2. 可以很容易修改一个单件，使它有少数几个实例，这样做是允许的而且是有意义的。

11.7 随书附带光盘中的程序

Shows how print spooler could be written thowing exception	`\Singleton\SinglePrinter`
Creates one instance or returns null	`\Singleton\InstancePrinter`

第12章　生成器模式

本章中，我们学习如何使用生成器模式（Builder Pattern）从部件构建对象。前面已经了解到，工厂模式能够根据传递给构造方法的参数返回几个不同子类中的一个。假设我们不但需要一个用于计算的算法，还需要一个由于显示数据的不同而完全不同的用户界面。典型的例子就是E-mail地址簿，在地址簿中，既有个人信息也有组的信息，而读者希望能根据信息的不同，改变地址簿的显示，这样在个人屏幕中能显示姓名、公司、E-mail地址和电话号码。

另一方面，如果显示一个组的地址页，希望能看到组名、组的职能、成员表及他们的E-mail地址。单击一个人的时候得到一种显示，单击一个组的时候得到另一种显示。假设所有的E-mail地址都保存在Address类的一个对象里，个人和组都派生于这个基类，如图12.1所示。

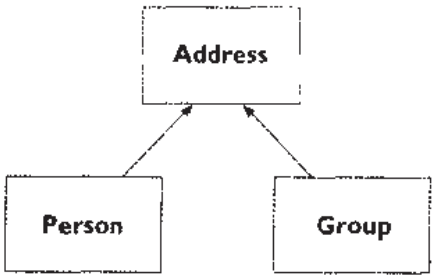


图12.1　Person类和Group类都派生于Address类

根据选择的Address对象类型的不同，我们希望看到该对象属性的不同显示。这比工厂模式稍复杂些，因为我们不是返回基类显示对象的子类对象，而是返回由显示对象的不同组合构成的完全不同的用户界面。根据数据的不同，Builder模式以不同形式组装多个对象，例如显示控件。此外，通过用类表示数据和用窗体表示显示，可以将数据和显示方法清晰地分开。

12.1　一个投资跟踪程序

我们考虑一个稍微简单一点的例子，在这个例子中，用一个类构造一个用户界面。假设我们要编写一个程序来跟踪投资的效益。我们有股票、债券和基金等投资项目，对每一种投资项目都要显示持有量的列表，这样就能够选择一种或多种投资项目并在图上标注出它们之间的效益对比。

尽管不能事先预测在给定的时间内每种投资项目的拥有量，但我们还是希望，无论是对大数量的投资项目（例如股票），还是小数量的投资项目（例如基金），都有一种易于使用的显示方式。每种情况下我们都想要一种多选显示，这样就能够选择一个或多个项目来标注。如果投资项目的数量大，可以使用一个多选的列表框显示；如果只有三个或更少的项目，则使用几个复选框来表示。让Builder类根据需要显示的项目个数生成界面，同时还要有相同的方法来返回结果。

我们的程序运行结果应如图12.2所示，上图是包含了大量股票的显示，下图是包含了少量债券的显示。接下来我们考虑如何构造界面，来完成这种不同的显示。先创建一个multiChoice接口，它定义了需要实现的方法。

```
public interface MultiChoice
{
        ArrayList getSelected();
        void clear();
        Panel getWindow();
 }
```

getWindows方法返回一个包含多选显示的面板（Panel），这里使用两个显示面板实现该界面：一个是复选框面板，一个是列表框面板。

```
public class CheckChoice:MultiChoice  {
```

或

```
public class ListChoice:MultiChoice    {
```

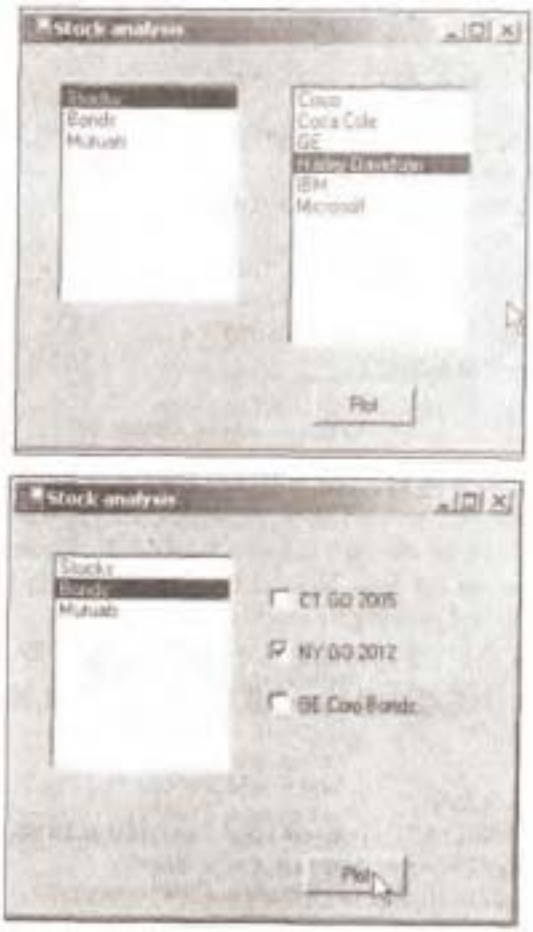


图 12.2　股票的列表框界面和债券的复选框界面

C#在设计生成器类方面具有相当大的灵活性，我们可以直接访问这样一些方法，这些方法允许我们由基本组件构建一个窗口。对本例来说，让每个生成器构建一个包含所需组件的面板，然后将面板添加到窗体中，并确定它的位置。当改变显示时，删除旧的面板，添加上新的面板。在C#里，面板只是一个无边框的、能包含多个Windows组件的容器。这里面板的两种实现过程都必须满足multiChoice接口。

接下来创建一个抽象基类Equities，并由它派生出Stocks，Bonds和Mutuals类。

```
public abstract class Equities   {
```

```
        protected ArrayList array;
        public abstract string ToString();
        //----------
        public ArrayList getNames() {
               return array;
        }
        //----------
        public int count() {
               return array.Count;
        }
}
```

注意抽象方法ToString，我们用该方法在列表框中显示每种股票。Stocks类只包含了向ArrayList中添加股票名称的代码。

```
public class Stocks:Equities     {
       public Stocks()      {
              array = new ArrayList();
              array.Add ("Cisco");
              array.Add ("Coca Cola");
              array.Add ("GE");
              array.Add ("Harley Davidson");
              array.Add ("IBM");
              array.Add ("Microsoft");
       }
       public override string ToString() {
              return "Stocks";
       }
}
```

其余的全部代码（getNames方法和count方法）已经在基类Equities实现。Bonds类和Mutuals类与此完全类似。

12.1.1 StockFactory

我们需要一个简单的类来决定，要返回的是一个复选框面板还是一个列表框面板。我们把这个类称为StockFactory。由于我们只需要该类的一个实例，因而在创建该类时让其中的一个方法为静态的。

```
public class StockFactory    {
       public static MultiChoice getBuilder(Equities stocks)  {
            if (stocks.count ()<=3) {
                 return new CheckChoice (stocks);
            }
            else {
                 return new ListChoice(stocks);
            }
           }
}
```

由于只需要该类的一个实例，所以我们把getBuilder方法声明为静态的，然后就可以直接调用它而无需创建该类的实例。用设计模式的语言来讲，这个简单工厂类叫做Director，而由multiChoice类派生出来的实际类就是Builder。

12.1.2　CheckChoice 类

这里的复选框生成器构建了一个包含 0 到 3 个复选框的面板，并将面板返回给调用程序。

```
//returns a panel of 0 to 3 check boxes
 public class CheckChoice:MultiChoice      {
       private ArrayList stocks;
       private Panel panel;
       private ArrayList boxes;
//------
       public CheckChoice(Equities stks)          {
       stocks = stks.getNames ();
       panel = new Panel ();
       boxes = new ArrayList ();
       //add the check boxes to the panel
       for (int i=0; i< stocks.Count; i++) {
              CheckBox ck = new CheckBox ();
              //position them
              ck.Location = new Point (8, 16 + i * 32);
              string stk = (string)stocks[ i] ;
              ck.Text =stk;
              ck.Size = new Size (112, 24);
              ck.TabIndex =0;
              ck.TextAlign = ContentAlignment.MiddleLeft;
              boxes.Add (ck);
              panel.Controls.Add (ck);
       }
}
}
```

返回窗口和选中名称列表的方法如下所示。注意，这里把由 ArrayList 返回的对象类型转换为该方法实际需要的 CheckBox 类型。

```
//------
//uncheck all check boxes
public  void clear() {
      for(int i=0; i< boxes.Count; i++) {
        CheckBox ck = (CheckBox)boxes[ i] ;
             ck.Checked =false;
      }
}
//------
//return list of checked items
public ArrayList getSelected() {
      ArrayList sels = new ArrayList ();
      for(int i=0; i< boxes.Count; i++) {
             CheckBox ck = (CheckBox)boxes[ i] ;
             if (ck.Checked ) {
                    sels.Add (ck.Text );
             }
      }
      return sels;
}
//------
//return panel of check boxes
public Panel getWindow() {
      return panel;
}
```

12.1.3 ListboxChoice类

该类创建一个多选列表框，并将其插入到一个面板中，然后将名称加到列表里。

```
public class ListChoice:MultiChoice {
      private ArrayList stocks;
      private Panel panel;
      private ListBox list;
//------
//constructor creates and loads the list box
      public ListChoice(Equities stks) {
            stocks = stks.getNames ();
            panel = new Panel ();
            list = new ListBox ();
            list.Location = new Point (16, 0);
            list.Size = new Size (120, 160);
            list.SelectionMode =SelectionMode.MultiExtended;
            list.TabIndex =0;
            panel.Controls.Add (list);
            for(int i=0; i< stocks.Count; i++) {
                  list.Items.Add (stccks[ i] );
            }
      }
```

由于这是一个多选列表框，我们可以把所有选中的项目都存放在一个SelectedIndices集合中。该方法只对多选列表框起作用，对于单选列表框它只返回一个1。这里用这种方法将选中的名称加到ArrayList中，程序如下所示。

```
      //returns the Panel
      //------
      public  Panel getWindow() {
            return panel;
      }
      //returns an array of selected elements
      //------
      public ArrayList getSelected() {
            ArrayList sels = new ArrayList ();
            ListBox.SelectedObjectCollection
                  coll = list.SelectedItems;
            for(int i=0; i< coll.Count; i++) {
                  string item = (string)coll[ i];
                        sels.Add (item );
            }
            return sels;
      }
      //------
      //clear selected elements
      public void clear() {
            list.Items.Clear();
}
```

12.2 使用ListBox控件中的Items集合

C#中的列表框不只限于使用字符串。向Items集合中添加的数据，可以是具有ToString方法的任何类型的对象。

由于我们创建的三个Equities子类都有一个ToString方法，因而可以在主程序的构造函数里，将这些类直接加到列表框中。

```
public class WealthBuilder : Form     {
       private ListBox lsEquities;
       private Container components = null;
       private Button btPlot;
       private Panel pnl;
       private MultiChoice mchoice;
       private void init() {
              lsEquities.Items.Add (new Stocks());
              lsEquities.Items.Add (new Bonds());
              lsEquities.Items.Add (new Mutuals());
       }
       public WealthBuilder() {
              InitializeComponent();
              init();
       }
```

任何时候单击列表框中的一行数据，单击方法会获得相应 Equities 类的实例，将该实例传给 MultiChoice 工厂，它接下来在相应的类中生成一个包含项目的面板，然后删除旧的面板，添加新的面板。

```
private void lsEquities_SelectedIndexChanged(object sender,
             EventArgs e) {
      int i = lsEquities.SelectedIndex;
      Equities eq = (Equities)lsEquities.Items[ i] ;
      mchoice= StockFactory.getBuilder (eq);
      this.Controls.Remove (pnl);
      pnl = mchoice.getWindow ();
      setPanel();
}
//------
private void setPanel() {
      pnl.Location = new Point(152, 24);
      pnl.Size = new Size(128, 168);
      pnl.TabIndex = 1;
      Controls.Add(pnl);
}
```

12.2.1　在图上标注数据

本例中，我们并不真正地实现实际的标注过程，只是提供一个 getSelected 方法，它返回来自 MultiChoice 的股票名称，该方法返回一个选中项目的 ArrayList。在处理单击 Plot（标注）按钮的方法中，将这些名称添加到信息框中并显示出来。

```
private void btPlot_Click(object sender, EventArgs e) {
//display the selected items in a message box
      if(mchoice != null) {
            ArrayList ar  = mchoice.getSelected ();
            string ans = "";
            for(int i=0; i< ar.Count; i++) {
                  ans += (string)ar[ i]  +" ";
            }
            MessageBox.Show (null, ans,
            "Selected equities",MessageBoxButtons.OK  );
      }
}
```

12.2.2 最终的选项程序

现在，已经创建了全部所需要的类，可以运行该程序了。开始运行时，在右边先显示一个空白面板，这样就可以在以后删除那里的面板。每次单击一个Equities名称，该位置的旧面板就被删除，添加一个新面板。在图 12.3 中会看到三种运行情况。

图 12.3 WealthBuilder 程序，显示各种资产列表及列表框面板、复选框面板和标注面板

可以在图 12.4 中的 UML 类图里看出类之间的关系。

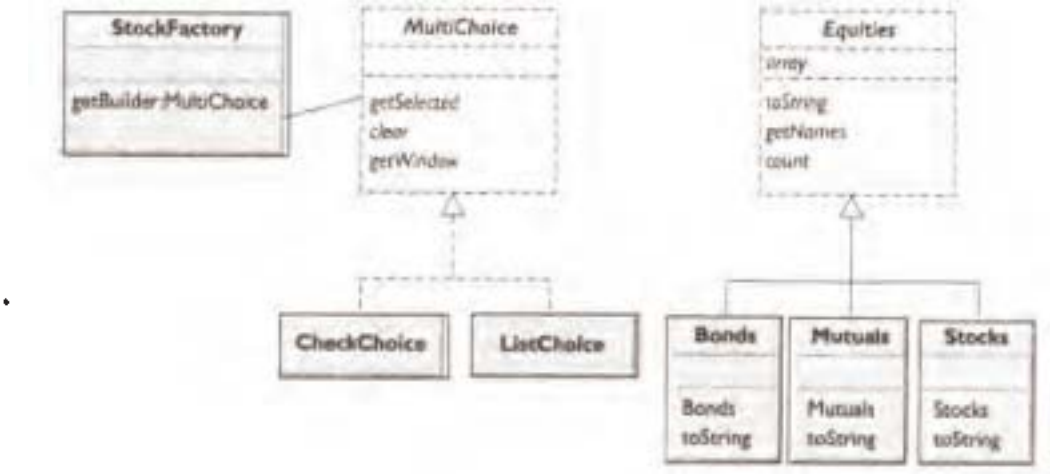


图 12.4 生成器模式中的继承关系

12.3 生成器模式的效果

1. 生成器允许读者改变产品的内部表示，同时也隐藏了产品如何组装的细节。

2. 每个特定的生成器都独立于其他的生成器，同时独立于程序的其他部分，这一点提高了对象的模块性，并使添加其他的生成器变得相对简单。
3. 由于每个生成器是根据数据一步一步构建最终结果，所以能精确地控制生成器构建的每个结果。

生成器模式有点类似于抽象工厂模式，两者都返回由许多方法和对象组成的类。它们之间的主要差别是，抽象工厂返回的是一系列相关的类，而生成器是根据提供给它的数据一步一步地构建一个复杂的对象。

12.4　思考题

1. 某些字处理程序和绘图程序能根据要显示的数据内容自动构建菜单。这种情况下如何有效地应用生成器模式？
2. 不是所有的生成器都必须构建可视化对象。假设你为一场由五到六个不同项目组成的田径运动会记分，如何在这里使用生成器？

12.5　随书附带光盘中的程序

Basic Equities Builder	`\Builders\Stocks`

第13章 原型模式

当在程序中确定了所需要的通用类，但需要将具体类延迟到运行时才能确定时，原型模式（Prototype Pattern）是另一种可使用的工具。原型模式与生成器模式的相似之处是，都由某个类确定组成最终类的部件或细节；不同之处在于，原型模式中目标类的构建是通过克隆一个或多个原型类，然后按预期的行为更改或补充被克隆类的细节而实现的。

当你需要的类仅在它们提供的处理方式上存在不同时，就可以使用原型模式，例如，在分析以不同进制数表示的字符串时。从这种意义上讲，原型模式几乎等同于Coplien(1992)描述的Examplar模式。

考虑一个有大量数据的数据库的例子，我们需要用一系列查询来构建一个答案。一旦得出这个答案的结果序列，就可以处理它来产生其他答案，不需要再发出其他的请求。

类似于我们在前面做过的一个例子，考虑由某个联合会或全国性组织中的大量游泳选手组成的数据库。在整个赛季中，每名运动员都会游若干场和若干距离。运动员的最好成绩记录在按年龄分组的表中，即使在一个四个月的赛季里，许多运动员也会度过他们的生日而排到新的年龄组。因而，查询一个赛季中各年龄组成绩最好的运动员与每次比赛的日期及各游泳选手的生日有关。汇总这个成绩表的计算成本是相当高的。

一旦拥有了一个包含按性别分类的表的类实例，还会希望能够检索只按时间分类或按实际年龄分类的信息，而不是按年龄组来检索。重新去计算这些数据是不明智的，而且我们也不想破坏原始数据的顺序，这时就需要有某种数据对象的拷贝。

13.1 C# 中的克隆

克隆（Clone）一个类实例（完全拷贝）的思想不是C#的固有属性，但也没有什么理由可以阻止你自己完成这样的操作。克隆方法出现在C#中的惟一地方是在ADO DataSet处理里。读者可以创建一个DataSet作为数据库查询的结果，其行指针可以一次移动一行。如果出于某种原因，需要把索引保持在DataSet的两个位置上，就需要两个“当前行”。C#处理这种情况的最简单方法是克隆DataSet。

```
DataSet cloneSet;
cloneSet = myDataSet.Clone( );
```

这种方法并不产生数据的两个拷贝，它只生成两套行指针，用于在记录之间相互独立地移动。因为实际上只有一个数据表，所以，DataSet的一个克隆体中的任何改变，都会立即反映到另一个克隆体中。我们在下面的例子中会讨论一个与此类似的问题。

13.2 使用原型

我们接下来编写一个简单的程序，从数据库读入数据，然后克隆结果对象。在这个例子中，与前面的论述一样，原始数据是从一个大数据库中导出来的，而我们只是从一个文件读入数据。文件具有如下格式：

```
Kristen Frost, 9, CAT, 26.31, F
Kimberly Watcke, 10, CDEV, 27.37, F
Jaclyn Carey, 10, ARAC, 27.53, F
Megan Crapster, 10, IEHY, 27.68, F
```

这里要用到先前开发的csFile类。首先，创建一个Swimmer类，它包含一个运动员的名字、俱乐部名字、性别和时间（即成绩），使用第4章开发的csFile类和StringTokenizer类读入这些数据。

```
public class Swimmer   {
      private string name;            //name
      private string lname, frname;   //split names
      private int age;                //age
      private string club;            //club initials
      private float time;             //time achieved
      private bool female;            //sex
 //---------
      public Swimmer(string line) {
            StringTokenizer tok = new StringTokenizer(line,",");
            splitName(tok);
            age = Convert.ToInt32 (tok.nextToken());
            club = tok.nextToken();
            time = Convert.ToSingle (tok.nextToken());
            string sx = tok.nextToken().ToUpper ();
            female = sx.Equals ("F");
      }
      //---------
      private void splitName(StringTokenizer tok) {
            name = tok.nextToken();
            int i = name.IndexOf (" ");
            if(i >0 ) {
                  frname = name.Substring (0, i);
                  lname = name.Substring (i+1).Trim ();
            }
      }
      //---------
      public bool isFemale() {
            return female;
      }
      //---------
      public int getAge() {
            return age;
      }
      //---------
      public float getTime() {
            return time;
      }
      //---------
      public string getName() {
            return name;
      }
      //---------
      public string getClub() {
            return club;
      }
```

接下来创建一个SwimData类，它包含一个从数据库读入的Swimmer对象的ArrayList。

```
public class SwimData {
      protected ArrayList swdata;
      private int index;
//-----
public SwimData() {
      swdata = new ArrayList ();
}
//-----
public SwimData(ArrayList swd) {
      swdata = swd;
      index=0;
}
//-----
public SwimData(string filename)
      swdata = new ArrayList ();
      csFile f1 = new csFile(filename);
      f1.openForRead();
      string s = f1.readLine();
      while(s != null){
            Swimmer sw = new Swimmer(s);
            swdata.Add(sw);
            s = f1.readLine();
      }
      f1.close();
}
//-----
public void moveFirst(){
      index = 0;
}
//-----
public bool hasMoreElements(){
      return (index < swdata.Count-1);
}
//-----
public void sort(){
}
//-----
public Swimmer getSwimmer(){
      if(index < swdata.Count)
            return (Swimmer)swdata[ index++];
      else
            return null;
}
}
```

然后，就可以使用该类读入选手数据，并把它们显示在列表框中。

```
private void init() {
      swdata = new SwimData ("swimmers.txt");
      reload();
}
//-----
private void reload() {
      lsKids.Items.Clear ();
      swdata.moveFirst ();
      while (swdata.hasMoreElements() ) {
            Swimmer sw = swdata.getSwimmer ();
```

```
            lsKids.Items.Add (sw.getName() );
        }
}
```

这个程序可以用图13.1来说明。

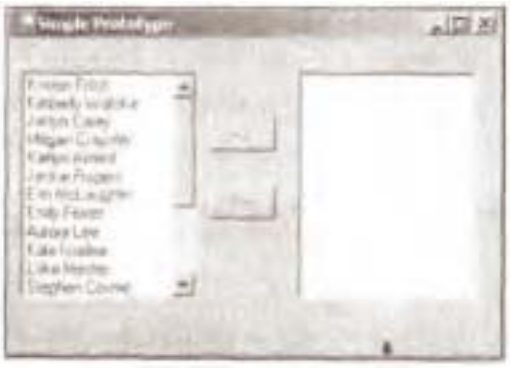


图13.1 一个简单的原型模式程序

单击"-->"按钮，克隆SwimData类，并在新类中对数据进行不同的排序。因为创建一个新的类实例会比较慢，而且我们想以两种形式保存数据，所以也对数据进行了克隆。

```
private void btClone_Click(object sender, EventArgs e) {
        SwimData newSd = (SwimData)swdata.Clone ();
        newSd.sort ();
        while(newSd.hasMoreElements() ) {
                Swimmer sw = (Swimmer)newSd.getSwimmer ();
                lsNewKids.Items.Add (sw.getName() );
        }
}
```

在图13.2中可以看到排序结果。

图13.2 原型模式程序的排序结果

13.3 克隆类

尽管不是特别需要，这里还是让SwimData类实现了ICloneable接口。

```
public class SwimData : ICloneable    {
```

这表示该类必须有一个Clone方法，它返回一个对象。

```
public object Clone ( )    {
```

```
        SwimData newsd = new SwimData ( swdata );
        return newsd;
    }
```

当然，使用这个接口意味着：就像我们以前做的那样，在收到Clone方法返回的对象时，必须把对象类型转回到SwimData类型。

```
SwimData newSd = (SwimData ) swdata.Clone ( );
```

接下来单击“<--”按钮，把原始数据重新加载到左侧的列表框里。图13.3给出的是一种打乱了的结果。

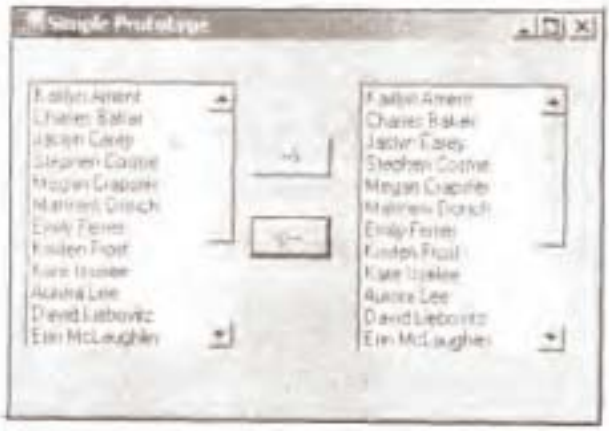


图13.3 原型模式程序显示出左边列表框中的数据进行过重新排序

为什么左边列表框中的名字也被重新排序了？我们的排序例程如下：

```
public void sort ( )   {
      //sort using IComparable interface of Swimmer
      swdata.Sort ( 0, swdata.Count, null )
}
```

注意，这里对实际的ArrayList进行了适当的排序。排序方法假设ArrayList中的每个元素都实现了IComparable接口。

```
public class Swimmer : IComparable   {
```

这表示Swimmer类必须要有一个整型的CompareTo方法，它根据两个对象之间的比较结果是小于、等于还是大于，返回-1，0或1。在这个例子中，我们使用字符串类的CompareTo方法来比较两个名字并返回结果。

```
public int CompareTo(object swo )  {
      Swimmer sw = (Swimmer ) swo;
      return lname.CompareTo (sw.getLName( )  );
}
```

现在就能理解图13.3所示的结果了。在新类里，原始数组被重新排序，而且实际上只有一个版本的数组。这是因为，克隆方法是对原始类的“影子”拷贝，换句话说，拷贝的只是数据对象的索引，而这些索引指向的是相同的数据。因而，对拷贝的数据执行的任何操作，也都同样发生在Prototype类中的原始数据上。

在某些情况下，这种影子拷贝是可以接受的，但如果读者想要对数据做深层拷贝，必须为要克隆的类编写一个深层克隆例程。在这个简单的类中，只要创建一个新的ArrayList对象，并把原来类中的ArrayList元素拷贝到新的对象中，就能实现深层拷贝。

```
public object Clone() {
      //create a new ArrayList
      ArrayList swd = new ArrayList ();
      //copy in swimmer objects
      for(int i = 0; i< swdata.Count; i++)
            swd.Add (swdata[ i] );
      //create new SwimData object with this array
      SwimData newsd = new SwimData (swd);
      return newsd;
}
```

13.4 使用原型模式

在需要创建大量的类时，或者在类创建后又需要修改时，都可以使用原型模式。只要所有的类都具有相同的接口，它们就都能完成不同的操作。

我们考虑一个更灵活的例子，列出刚才讨论的游泳选手的列表。这里不只是对游泳选手进行排序，还要创建对数据进行操作的子类，修改它并在列表框中显示结果。这里从同样的基类SwimData开始创建。

接下来，就可以根据应用程序的需要，编写SwimData的不同派生类。总是以SwimData类开始，为不同的显示克隆这个类。例如，SexSwimData类是按照性别对数据重新排序，并只显示一种性别的数据，如图13.4所示。

图13.4 SexSwimData类只在右边显示一种性别的选手姓名

在SexSwimData类中，我们按照名字排序，但是当返回数据用于显示时，是基子被显示的选手是男孩还是女孩（即根据性别返回数据）。这个类具有这种形式的排序方法。

```
public void sort(bool isFemale) {
      ArrayList swd = new ArrayList();
      for (int i = 0; i < swdata.Count; i++) {
            Swimmer sw =(Swimmer)swdata[ i] ;
            if (isFemale == sw.isFemale() ) {
                  swd.Add (sw);
            }
      }
      swdata = swd;
}
```

每次单击一个性别选项按钮时，要将这些按钮的当前状态传递给类。

```
private void btClone_Click(object sender, System.EventArgs e) {
        SexSwimData newSd = (SexSwimData)swdata.Clone ();
        newSd.sort (opFemale.Checked );
        lsNewKids.Items.Clear();
        while(newSd.hasMoreElements() ) {
                Swimmer sw = (Swimmer)newSd.getSwimmer ();
                lsNewKids.Items.Add (sw.getName() );
        }
}
```

注意，btClone_Click事件克隆了SexSwimdata基类的实例swdata，并将结果转换为SexSwimData类型。这意味着SexSwimData类的Clone方法必须覆盖SwimData类的Clone方法，因为它返回的是不同的数据类型。

```
public object Clone() {
        //create a new ArrayList
        ArrayList swd = new ArrayList ();
        //copy in swimmer objects
        for(int i=0; i< swdata.Count; i++)
                swd.Add (swdata[ i] );
        //create new SwimData object with this array
        SexSwimData newsd = new SexSwimData (swd);
        return newsd;
}
```

每次派生一个特别相似的新类时，都要重写Clone方法，这是一个不太令人满意的方式。更好的解决方案是，取消让每个类都去实现有Clone方法的ICloneable接口，把这个处理过程改为让每个接收类克隆发送类中的数据。下面给出SwimData类的修订部分，它包含了cloneMe方法，该方法接收来自另外一个SwimData实例的数据，并将其拷贝到当前实例的ArrayList中。

```
public class swimData     {
        protected ArrayList swdata;
        private int index;
        //-----
        public void cloneMe(SwimData swdat) {
                swdata = new ArrayList ();
                ArrayList swd=swdat.getData ();
                //copy in swimmer objects
                for (int i=0; i < swd.Count; i++)
                        swdata.Add(swd[ i] );
        }
```

接下来，就可以将这种方法应用到所有的SwimData子类上，并且不用在子类之间做数据转换。

13.4.1 具有相同接口的不同类

类没有必要都那么相似。AgeSwimData类接收克隆的输入数组，并创建了一个简单的年龄直方图。如果单击“F”，会看到女孩的年龄分布，如果单击“M”就会看到男孩的年龄分布，如图13.5所示。

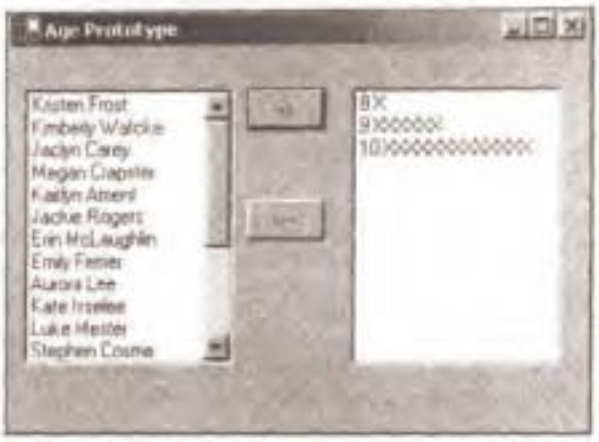


图 13.5 AgeSwimData类显示的年龄分布情况

这是一种很有趣的情况：AgeSwimData类继承了基类SwimData的cloneMe方法，但它覆盖了原有的sort方法，新的sort方法用相应年龄组中运动员数量组成的名字生成了一名新的游泳选手（proto-swimmer）。

```
public class AgeSwimData:SwimData  {
      ArrayList swd;
      public AgeSwimData() {
           swdata = new ArrayList ();
      }
      //------
      public AgeSwimData(string filename):base(filename){}
      public AgeSwimData(ArrayList ssd):base(ssd){}
      //------
      public override void cloneMe(SwimData swdat) {
           swd = swdat.getData ();
      }
      //------
      public override void sort() {
           Swimmer[ ] sws = new Swimmer[ swd.Count ];
           //copy in swimmer objects
           for(int i=0; i < swd.Count; i++) {
                sws[ i] = (Swimmer)swd[ i];
           }
           //sort into increasing order
           for( int i=0; i< sws.Length; i++) {
                for (int j = i; j< sws.Length; j++) {
                     if (sws[ i].getAge ()>sws[ j].getAge ()) {
                          Swimmer sw = sws[ i];
                          sws[ i]=sws[ j];
                          sws[ j]=sw;
                     }
                }
           }
           int age = sws[ 0].getAge ();
           int agecount = 0;
           int k = 0;
           swdata = new ArrayList ();
           bool quit = false;

           while( k < sws.Length && ! quit ) {
```

```
                while(sws[ k] .getAge() ==age && ! quit) {
                      agecount++;
                      if(k < sws.Length -1)
                            k++;
                      else
                            quit= true;
                }
          //create a new Swimmer with a series of X's for a name
          //for each new age
                string name = "";
                for(int j = 0; j < agecount; j++)
                      name +="X";
                Swimmer sw = new Swimmer(age.ToString() + " " +
                      name + "," + age.ToString() +
                      ",club,0,F");
                      swdata.Add (sw);
                      agecount = 0;
                      if(quit)
                            age = 0;
                      else
                            age = sws[ k] .getAge ();
          }

     }
}
```

由于我们的原始类显示的是选中选手的firstname和lastname，现在为了实现同样的显示，在返回的Swimmer对象里，把firstname设置成年龄串，把lastname设置成直方图。

图13.6中的UML图清楚地说明了本系统中类的关系。主要的GUI类保存了两个SwimData实例，但没有具体指明是哪一个类的。AgeSwimData类派生于SwimData类，两者都包含了Swimmer类的实例。

图13.6 各种SwimData类的UML图

读者还应该注意到，可以不必局限于我们这里图示的几个子类。创建其他的具体类，并以代码选择合适的具体类来对它们注册，这都是很简单的。在我们的例子中，因为是用户单击其中的一个按钮，用户就是决策点或工厂。在一种更灵活的情况里，每个具体类都有一个特征数组，决策点可

以是类注册表或原型管理器，它去检查这些特征值并选择最合适的类。也可以将工厂方法模式和原型模式组合在一起使用，每个具体类都可以从可用的具体类中选择其中的一个来使用。

13.5 原型管理器

原型管理器类用于决定将哪一个具体类返回给客户端，它也能同时管理几个系列的原型。例如，除了能返回几个游泳选手类中的一个类，也能返回游了指定场数和距离的不同组的选手，还能管理要返回的是哪一种显示类型的列表框，包括表、多列列表和图形显示等。它的好处是，不管返回哪一个子类，都不需要转换成程序中要用到的新类类型。换句话说，抽象基类的方法已经足够了，客户端不必知道它处理的是哪一个子类。

13.6 原型模式的效果

用原型模式能根据需要克隆类，这样，在运行时就可以添加或删除类。根据程序运行情况，可以在运行时更改一个类的内部数据表示，也可以在运行时指定新对象而无需创建一个新类。

用C#实施原型模式的困难在于：如果类早已存在，则不能改变它们来增加需要的克隆方法。另外，间接引用其他类的类也不能被真正克隆。

类似于前面讨论的单件中的注册表，读者也可以创建一个能被克隆的原型类的注册表，访问注册表对象可得到一列可用的原型，这样就可以克隆一个已存在的类而不是勉强再编写一个。

注意，为了使用原型注册表，要当做原型的每个类自身必须被实例化（可能需要些代价），这可能会影响性能。

最后，拷贝原型类的想法意味着，你对类中的数据或方法有足够的访问权，这样在克隆之后可以修改它们。这可能需要向原型类中添加数据访问方法，这样在克隆了该类后，就能修改数据。

13.7 思考题

编写一个程序显示一句口号，它能在不同的时间出现在屏幕的不同位置上，并以不同的字体和大小来显示。使用原型模式设计该程序。

13.8 随书附带光盘中的程序

Age Plot	`\Prototype\Ageplot`
Deep Prototype	`\Prototype\DeepProto`
Display by Sex	`\Prototype\OneSex`
Shallow Copy	`\Prototype\SimpleProto`
Age and Sex Display	`\Prototype\TwoclassAgePlot`

13.9 创建型模式小结

工厂模式（Factory Pattern）根据提供给工厂的数据，从一系列相关的类中选择一个类实例并返回。

抽象工厂模式（Abstract Factory Pattern）用于返回一组类中的一个，在某些情况下，它实际上为一组类返回了一个工厂。

生成器模式（Builder Pattern）根据提供给它的数据及其表示，将一系列对象组装成一个新对象。通常选择何种方式组装对象由工厂决定。

当创建新实例代价比较高的时候，原型模式（Prototype Pattern）拷贝或克隆一个现有的类，而不是创建一个新实例。

单件模式（Singleton Pattern）可以保证有且只有一个对象实例，并提供一个该实例的全局访问点。

第三部分

结构型模式

结构型模式（Structural Pattern）描述的是如何组合类和对象以获得更大的结构。类模式和对象模式之间的区别是，类模式描述的是如何使用继承提供更有用的程序接口；而对象模式描述的是通过使用对象组合或将对象包含在其他对象里，将对象组合成更大的结构。

例如，我们后面会看到的适配器模式（Adapter Pattern）可以使一个类的接口匹配另外一个接口，从而使程序设计变得更容易。还会看到其他的结构型模式，能将对象组合来实现新的功能。例如组合模式（Composite Pattern），正如它的名字一样——对象的组合，其中每个对象既可以是简单对象也可以是组合对象。代理模式（Proxy Pattern）通常是一个简单对象，它代替一个比较复杂的、稍后会被调用的对象，例如，当程序运行在一个网络环境中时。

享元模式（Flyweight Pattern）用于共享对象，其中的每个实例都不包含自己的状态，而是将状态存储在外部。当存在大量的对象而它们只有几种类型时，对象共享可以节省存储空间。

外观模式（Facade Pattern）用一个类表示一个子系统，桥接模式（Bridge Pattern）将对象的接口和它的实现分离，这样就可以独立地改变它们。最后，我们会看到装饰模式（Decorator Pattern），它可以动态地给对象添加职责。

读者会看到这些模式之间存在着一些重叠，它们甚至还会和后面章节中的行为型模式相重叠。在介绍完这些模式之后，我们会总结这些相似点。

第14章　适配器模式

适配器模式（Adapter Pattern）可以将一个类的程序设计接口转换成另外一个接口。当我们想让不相关的类在一个程序里一起工作时，可以使用适配器。适配器的概念相当简单：编写一个具有所需要的接口的类，然后让它和拥有不同接口的类进行通信。

有两种方法可以做到这一点：通过继承和通过对象组合。第一种情况里，读者从一个不一致的类里派生出一个新类，然后添加需要的方法，使新的派生类能匹配所需要的接口。另一种方式是，把原始类包含在新类里，然后在新类里创建方法去转换调用。这两种方法分别叫做类适配器和对象适配器，二者都很容易实现。

14.1　在列表之间移动数据

我们考虑一个简单的例子，从一个列表中选择一些名字，然后移到另一个列表中，并显示出与这些人相关的详细信息。初始列表由一个游泳队的花名册组成，第二个表由名字和时间（或成绩）组成。

在图14.1所示的简单程序中，程序在初始化时从一个花名册文件读入名字，单击一个名字，然后单击箭头按钮，就把该名字移到右边的列表框中。单击一个名字，然后单击反向箭头按钮，就把该名字从右边的列表框移到左边的列表框中。

图 14.1　一个选择名字进行显示的简单程序

这是一个很容易编写的C#程序，它由可视化布局和各个按钮单击事件的动作例程组成。我们从花名册文件读入数据，将每个选手的名字和成绩存储在一个Swimmer对象中，然后把所有的对象存储在一个叫swdata的ArrayList中。选择一个名字以扩充形式显示，先取得左边列表框中被选中选手的表索引，再将这个选手的数据显示在右边的列表框中。

```
private void btClone_Click(object sender, EventArgs e) {
int i = lskids.SelectedIndex ();
  if( i >= 0) {
```

```
        Swimmer sw = swdata.getSwimmer (i);
        lsnewKids.Item.Add (sw.getName() +"\t"+sw.getTime ());
        lskids.SelectedIndex = -1;
    }
}
```

同样地，如果想从右边列表框中删除一个名字，先获得选中者的索引，再删除该名字。

```
private void putBack_Click ( object sender, EventArgs e )  {
      int i = lsnewKids.SelectedIndex  ();
      if ( i >= 0)
            lsNewKids.Items.RemoveAt (i);
}
```

注意，我们是用制表符（即 \t）得到数据间的列间隔，只要名字的长度相差不多，就不会出现什么问题。然而，如果一个名字比其他名字长或短很多，列表会使用一个不同的制表列宽度结束。这正是例子中第三个名字的情况。

14.2　创建一个适配器

列表框中的 Item 集合是用在某些操作上而不是其他的操作，记住这一点有点困难。由于这个原因，我们希望有一个类能隐藏这些复杂性，并将界面转换成我们所希望的较简单的形式，而不是像 VB6 中的 ListBox 那样的界面。先在 ListAdapter 类中创建一个简单接口，然后对 ListBox 的实例进行操作。

```
public class ListAdapter    {
      private ListBox listbox;   //operates on this one
      public ListAdapter(ListBox lb)       {
            listbox = lb;
      }
      //-----
      public void Add(string s) {
            listbox.Items.Add (s);
      }
      //-----
      public int SelectedIndex() {
            return listbox.SelectedIndex;
      }
      //-----
      public void Clear() {
            listbox.Items.Clear ();
      }
      //-----
      public void clearSelection() {
            int i = SelectedIndex();
            if(i >= 0) {
                  listbox.SelectedIndex =-1;
            }
      }
}
```

然后，就能让程序变得简单一点。

```
private void btClone_Click(object sender, EventArgs e) {
      int i = lskids.SelectedIndex ();
      if( i >= 0) {
```

```
            Swimmer sw = swdata.getSwimmer (i);
            lsnewKids.Add (sw.getName() +"\t"+sw.getTime ());
            lskids.clearSelection ();
      }
}
```

如果总是像这样间隔地添加游泳选手的名字和时间，那么在ListAdapter类中也许应该有一个方法直接处理Swimmer对象。

```
public void Add( Swimmer sw ) {
      listbox.Items.Add ( sw.getName () + "" + sw.getTime ());
}
```

这会进一步简化Click事件处理程序。

```
private void btClone_Click(object sender, EventArgs e) {
      int i = lsKids.SelectedIndex ();
      if( i >= 0) {
            Swimmer sw = swdata.getSwimmer (i);
            lsNewKids.Add (sw);
            lsKids.clearSelection ();
      }
}
```

我们所做的就是创建了一个包含ListBox类的Adapter类，并简化了ListBox的使用。接下来，会看到如何使用同样的方法为两个或更多的复杂可视化控件创建适配器。

14.3 使用 DataGrid

为了避免简单列表框中的制表列问题，我们改成使用网格显示数据。Visual Studio.NET将网格形式的表称为DataGrid（数据网格），它可以绑定在一个数据库上或一个内存数组上。为了使用不带数据库的DataGrid，需要创建一个DataTable类的实例，并向其加入DataColumn实例，默认情况下，DataColumn是字符串类型，读者也可以在创建时将其定义为其他类型。下面是使用DataTable创建一个DataGrid的通用框架。

```
Datatable dTable = new DataTable("Kids");
dTable.MinimumCapacity = 100;
dTable.CaseSensitive = false;

DataColumn column =
new DataColumn("Fname",System.Type.GetType("System.String"));
dTable.Columns.Add(column);
column = new DataColumn("Lname",
      System.Type.GetType("System.String"));
dTable.Columns.Add(column);
column = new DataColumn("Age",
      System.Type.GetType("System.Int16"));

dTable.Columns.Add(column);

dGrid.DataSource = dTable;
dGrid.CaptionVisible = false;     //no caption
dGrid.RowHeadersVisible = false; //no row headers
dGrid.EndInit();
```

为了向DataTable加入文本，我们让表生成一个行对象，然后将行对象中的元素设置成相应的数据。如果数据都是String类型，那么就拷贝字符串，但如果有一列数据是不同的类型，比如这里的年龄列是整数类型，就必须保证在设置该列数据时使用相应的类型。注意，读者可以使用名字或索引值引用相应的数据列。

```
DataRow row=dTable.NewRow();
row[ "Frname"] =sw.getFrname();
row[ 1] =sw.getName();
row[ 2] =sw.getAge(); //This one is an integer
dTable.Rows.Add(row);
dTable.AcceptChanges();
```

我们希望，不用修改为简单列表框编写的任何代码就能直接使用网格，可以通过创建实现下列接口的GridAdapter类来实现这一点。

```
public interface LstAdapter     {
      void Add ( Swimmer sw );
      int SelectedIndex ();
      void Clear ();
      void clearSelection ();
}
```

GridAdapter类实现了该接口，并用一个网格实例来初始化自己。

```
public class GridAdapter:LstAdapter   {
      private DataGrid grid;
      private DataTable dTable;
      private int row;
      //-----
      public GridAdapter(DataGrid grd)      {
            grid = grd;
            dTable = (DataTable)grid.DataSource;
            grid.MouseDown +=
                  new System.Windows.Forms.MouseEventHandler
                  (Grid_Click);
            row = -1;
      }
      //-----
      public void Add(Swimmer sw) {
            DataRow row = dTable.NewRow();
            row[ "Frname"]  = sw.getFrname();
            row[ 1]  = sw.getLName();
            row[ 2]  = sw.getAge();  //This one is an integer
            dTable.Rows.Add(row);
            dTable.AcceptChanges();
      }
      //-----
      public int SelectedIndex() {
            return row;
      }
      //-----
      public void Clear() {
            int  count = dTable.Rows.Count;
            for(int i=0; i< count; i++) {
                  dTable.Rows[ i] .Delete ();
            }
      }
```

```
        //-----
        public void clearSelection() {}
}
```

14.3.1 检查选择的行

DataGrid 没有 SelectedIndex 属性，行对象也没有 Selected 属性。必须用 MouseEvent 处理程序检查 MouseDown 事件，得到 HitTest 对象后，再判断用户是否单击了一个网格单元。

```
public void Grid_Click ( object sender, MouseEventArgs e)  {
        DataGrid.HitTestInfo hti = grid.HitTest (e.X, e.Y);
        if ( hti.Type == DataGrid.HitTestType.Cell )  {
              row = hti.Row;
        }
}
```

注意，在单击“-->”按钮时，可以简单地调用 GridAdapter 类的 Add 方法，不用考虑使用的是哪一种显示控件。

```
private void btClone_Click(object sender, System.EventArgs e) {
        int i = lskids.SelectedIndex ();
        if( i >= 0) {
               Swimmer sw = swdata.getSwimmer (i);
               lsNewKids.Add (sw);
               lskids.clearSelection ();
        }
}
```

我们在图 14.2 中看到的是最终的网格显示情况。

图 14.2 一个网格适配器

14.4 使用 TreeView

如果使用 TreeView（树形列表）控件显示选中的数据，读者会发现，没有一个方便的接口能保持代码不做修改。

对于要创建的每一个树节点，都要创建一个 TreeNode 类的实例，再将根 TreeNode 集合加到其他节点上。在使用 TreeView 的程序版本中，我们把游泳选手的名字加到根节点集合里，把时间加到辅助节点里。下面是完整的 TreeAdapter 类。

```
public class TreeAdapter:LstAdapter   {
        private TreeView tree;
```

```
//------
public TreeAdapter(TreeView tr)      {
     tree=tr;
}
//------
public void Add(Swimmer sw) {
     TreeNode nod;
  //add a root node
     nod = tree.Nodes.Add(sw.getName());
  //add a child node to it
  nod.Nodes.Add(sw.getTime().ToString ());
  tree.ExpandAll ();
}
//------
public int SelectedIndex() {
     return tree.SelectedNode.Index;
}
//------
public void Clear() {
       TreeNode nod;
       for (int i=0; i< tree.Nodes.Count; i++) {
              nod = tree.Nodes [ i];
              nod.Remove ();
       }
}
//------
public void clearSelection() {}
}
```

TreeDemo 程序如图 14.3 所示。

图 14.3　TreeDemo 程序

14.5　类适配器

在类适配器方法里，我们要从 ListBox（或网格、树形控件）派生一个新类，并向其中加入需要的方法。在下面这个类适配器的例子里，创建了一个新类 MyList，它派生于 ListBox 类，并实现了下面的接口。

```
public interface ListAdapter       {
      void Add(Swimmer sw );
      void Clear ( );
      void clearSelection ( );
}
```

派生类 MyList 如下所示：

```
public class MyList : System.Windows.Forms.ListBox, ListAdapter {
      private System.ComponentModel.Container components = null;
   //-----
      public MyList()           {
            InitializeComponent();
      }
      //-----
      public void Add(string s) {
            this.Items.Add (s);
      }
      //-----
      public void Add(Swimmer sw) {
            this.Items.Add (sw.getName() +
                  "\t" + sw.getAge ().ToString () );
      }
      //-----
      public void Clear() {
             this.Items.Clear ();
      }
      //-----
      public void clearSelection() {
             this.SelectedIndex = -1;
      }
```

图 14.4 给出了类图。其余的代码类似于对象适配器版本中的代码。

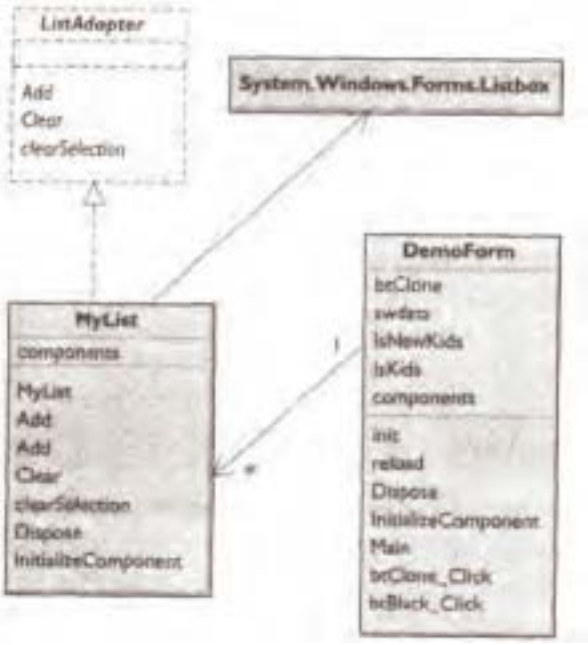


图 14.4 列表适配器的类适配器方法

在类适配器和对象适配器之间还有一些差别，尽管这些差别不如在 C++ 中重要。

- 当我们想匹配一个类和它的所有子类时，类适配器将不能胜任，因为在创建子类时就已定义了派生它的基类。
- 类适配器允许适配器更改某些被匹配的类的方法，同时还允许使用其他未修改的方法。

- 对象适配器通过将子类传递给构造函数而允许匹配所有子类。
- 对象适配器要求读者将希望使用的、被匹配对象的方法提到表面上来。

14.6　双向适配器

双向适配器是一个巧妙的概念，它允许不同的类把一个对象看做是ListBox类型或DataGrid类型。因为基类中所有的方法对派生类来说都是自动可用的，所以使用类适配器很容易实现双向适配器。尽管如此，只有在读者没有用其他行为覆盖基类中的方法时，它才能起作用。

14.7　C# 中对象适配器与类适配器的对比

前面举例说明的C#的List，Tree和Grid适配器都是对象适配器，也就是说，它们都是包含我们要匹配的可视化组件的类。编写一个List或Tree的类适配器也很容易，让它派生于基类，并包含新的添加（add）方法。

在使用DataGrid的情况下，这可能是一个不好的主意，因为在DataGrid类中需要创建DataTable实例和DataColumn实例，使它成为一个庞大而复杂的类，带有太多其他类如何工作的信息。

14.8　可插入的适配器

可插入适配器是能动态地匹配几个类的适配器。当然，适配器只能匹配它能识别的类，通常适配器根据不同的构造函数或参数设定（set-Parameter）方法决定匹配哪一个类。

14.9　思考题

如何编写一个类适配器，使DataGrid看起来像一个两列的列表框？

14.10　随书附带光盘中的程序

Tree Adapter	`\Adapter\TreeAdapter`
List Adapter	`\Adapter\ListAdapter`
Grid Adapter	`\Adapter\GridAdapter`
Class-Based List Adapter	`\Adapter\ClassAdapter`

第15章 桥接模式

初看起来，桥接模式（Bridge Pattern）非常像适配器模式，都是用一个类将一种接口转换成另一种接口。但是适配器模式的意图是：使一个或多个类的接口看起来像一个特定类的接口。桥接模式将类的接口和它的实现分离，无需修改客户端代码就可以改变或替换实现过程。

桥接模式的参与者有 Abstraction，它定义类的接口；Refined Abstraction，它扩展并实现由 Abstraction 定义的接口；Implementor，它定义实现类的接口；ConcreteImplementor，它是实现类。

假设我们有一个程序，要在窗口中显示一系列产品。最简单的显示界面是一个简单的ListBox，但在售出大量产品后，我们希望能在表中显示出产品名称以及销售数量。

由于刚刚讨论过适配器模式，读者可能会立即想到基于类的适配器，把ListBox接口转换成这种显示所需要的简单接口。在简单的程序中，这么做能满足要求，但正如我们要看到的，这种方法有些局限性。

对上述问题做进一步假设，我们需要给出产品数据的两种显示：一个是消费者视图，就是我们前面提到的产品列表；一个是管理者视图，除产品列表外，还要显示发货数量。我们在普通的ListBox 中显示产品列表，在 DataGrid 表中显示管理者视图。两种显示方式都由显示类实现，如图 15.1 所示。

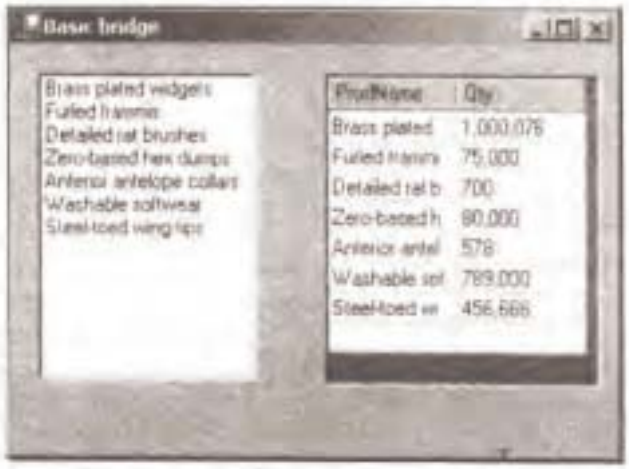


图 15.1 用桥接模式对同样信息给出了两种显示方式

15.1 桥接接口

我们现在定义一个简单的接口，它保留相同的部分而不管实际实现类的类型和复杂性。先定义 Bridger 接口。

```
//Bridge interface to display list classes
       public interace Bridge     {
              void addData (ArrayList col );
       }
```

该类只接收一个数据 ArrayList，并将其传递给显示类。

还要定义一个Product类，它包含了产品的名字和数量，并分析来自于数据文件的输入串。

```
public class Product : IComparable    {
      private string quantity;
      private string name;
      //-----
      public Product(string line)          {
           int i = line.IndexOf ("--");
           name =line.Substring (0, i).Trim ();
           quantity = line.Substring (i+2).Trim ();
      }
      //-----
      public string getQuantity() {
           return quantity;
      }
      //-----
      public string getName() {
            return name;
      }
 }
```

桥接模式的另一部分是实现类，它们通常都有一个更灵活的、低层次的接口。这里让实现类向显示屏幕添加数据行，一次添加一行数据。

```
public interface VisList    {
      //add a line to the display
      void addLine(Product p);
      //remove a line from the display
      void removeLine(int num);
 }
```

左边的接口和右边的实现之间的桥梁是ListBridge类，它实例化其中的一个列表显示类。注意，它实现了Bridger接口供应用程序使用。

```
 public class ListBridge : Bridger    {
      private VisList vis;
      //------
      public ListBridge(VisList v)          {
           vis = v;
      }
      //-----
      public virtual void addData(ArrayList ar) {
           for(int i=0; i< ar.Count; i++) {
                 Product p = (Product)ar[ i] ;
                 vis.addLine (p);
           }
      }
 }
}
```

注意，我们把VisList的变量声明为保护的，把addData方法声明为虚函数，这样在以后就可以扩展这个类。在程序设计顶层，只是用ListBridge类创建了一个表实例和一个列表实例。

```
private void init() {
      products = new ArrayList ();
      readFile(products);  //read in the data file
      //create the product list
      prodList = new ProductList (lsProd);
```

```
        //Bridge to product VisList
        Bridger lbr = new ListBridge (prodList);
        //put the data into the product list
        lbr.addData (products);
        //create the grid VisList
        gridList = new GridList(grdProd);
        //Bridge to the grid list
        Bridger gbr = new ListBridge (gridList);
        //put the data into the grid display
        gbr.addData (products);
    }
```

15.2 VisList类

两个VisList类实际上非常相似，消费者版本的类对ListBox操作并将名字加入其中。

```
//A VisList class for the ListBox
 public class ProductList : VisList  {
       private ListBox list;
       //-----
       public ProductList(ListBox lst)      {
            list = lst;
       }
       //-----
       public void addLine(Product p) {
            list.Items.Add (p.getName() );
       }
       //-----
       public void removeLine(int num) {
            if(num >=0 && num < list.Items.Count ){
                 list.Items.Remove (num);
            }
       }
 }
```

除了要向表中的两个列添加产品名称和数量外，VisList的GridList版本与上面的类很相似。

```
public class GridList:VisList    {
       private DataGrid grid;
       private DataTable dtable;
       private GridAdapter gAdapter;
       //-----
       public GridList(DataGrid grd) {
              grid = grd;
              dtable = new DataTable("Products");
              DataColumn column = new DataColumn("ProdName");
              dtable.Columns.Add(column);
              column = new DataColumn("Qty");
              dtable.Columns.Add(column);
              grid.DataSource = dtable;
              gAdapter = new GridAdapter (grid);
       }
       //-----
       public void addLine(Product p) {
            gAdapter.Add (p);
      }
```

15.3 类图

图15.2中的Bridge类的UML图清楚地表明了接口和实现的分离。左边的Bridger类是Abstraction，ListBridge类是它的实现。VisList接口描述了列表类ProductList和GridList的公共接口。VisList接口定义了Implementor的接口，Concrete Implementor是ProductList类和GridList类。

注意，尽管这两个Concrete Implementor都支持VisList接口，但它们在细节上很不相同。

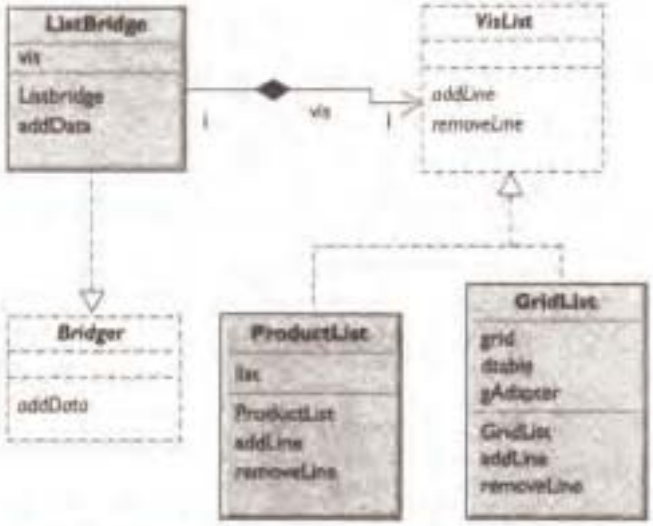


图15.2 对产品信息进行两种显示的桥接模式的UML图

15.4 扩展Bridge

现在假设，需要在列表显示数据的方式上做些修改。例如，以字母顺序显示产品。读者可能会想到修改或子类化列表类和表格类，这很容易导致维护上的噩梦，特别是如果在某个时候需要两个以上这样的显示时。我们换种做法，派生一个类似于ListBridge的新类SortBridge。

为了对Product对象进行排序，让Product类实现IComparable接口，目的是让它拥有CompareTo方法。

```
public class Product : IComparable    {
       private string quantity;
       private string name;
       //-----
       public Product(string line)          {
              int i = line.IndexOf ("--");
              name =line.Substring (0, i).Trim ();
              quantity = line.Substring (i+2).Trim ();
       }
       //-----
       public string getQuantity() {
              return quantity;
       }
       //-----
       public string getName() {
              return name;
       }
       //-----
       public int CompareTo(object p) {
```

```
            Product prod =(Product) p;
            return name.CompareTo (prod.getName ());
}
```

有了上述改动以后，Product 对象的排序就很简单了。

```
public class SortBridge:ListBridge    {
      //-----
      public SortBridge(VisList v):base(v){
      }
      //-----
      public override void addData(ArrayList ar) {
            int max = ar.Count;
            Product[ ] prod = new Product[ max] ;
            for(int i=0; i< max; i++) {
                  prod[ i] = (Product)ar[ i] ;
            }
            for(int i=0; i < max; i++) {
                  for (int j=i; j < max; j++) {
                        if(prod[ i] .CompareTo (prod[ j] )>0)  {
                              Product pt = prod[ i] ;
                              prod[ i] = prod[ j] ;
                              prod[ j] = pt;
                        }
                  }
            }
            for(int i = 0; i< max; i++) {
                  vis.addLine (prod[ i] );
            }
  }
}
```

在图 15.3 中可以看到排序结果。

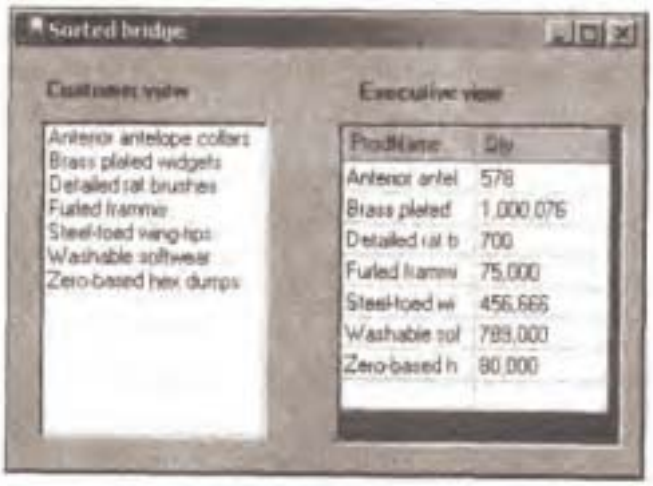


图 15.3 用 SortBridge 类生成的排序表

这清楚地表明，不用修改实现就能改变接口，反过来也是一样。例如，创建一个其他类型的列表显示类并替换当前的显示，只要新的列表类也实现了 VisList 接口，就不需要改变其他的程序。下面是 TreeList 类。

```
public class TreeList:VisList    {
      private TreeView tree;
      private TreeAdapter gAdapter;
```

```
        //-----
        public TreeList(TreeView tre) {
               tree = tre;
               gAdapter = new TreeAdapter (tree);
        }
        //-----
        public void addLine(Product p) {
               gAdapter.Add (p);
        }
```

注意，我们利用了前一章编写的TreeAdapter类，并修改它使之能对Product对象操作。

```
public class TreeAdapter    {
        private TreeView tree;
        //------
        public TreeAdapter(TreeView tr)      {
             tree=tr;
        }
        //------
        public void Add(Product p) {
               TreeNode nod;
          //add a root node
          nod = tree.Nodes.Add(p.getName());
          //add a child node to it
          nod.Nodes.Add(p.getQuantity ());
          tree.ExpandAll ();
        }
```

图15.4中，创建了一个树形列表组件，它实现了VisList接口，并替换了原来的列表而没有修改类的公共接口。

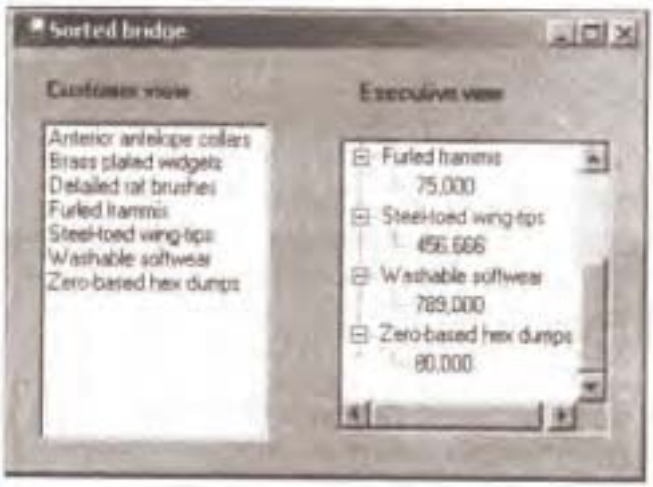


图15.4 使用Bridge的树形列表显示

15.5 Windows窗体充当Bridge

.NET中的可视化控件本身就是一个理想的实现了桥接模式的例子。控件是一种可复用的软件组件，可以在构造器工具里可视化地处理。为了便于修改，所有的C#控件都支持一个查询接口，使构造器程序能列举出它们的属性并显示出来。图15.5是在Visual Studio.NET上截取下来的屏幕，它

显示了带有一个文本域和一个复选框的面板。图右边的构造器面板表明，使用一个简单的可视化接口可以修改组件中的每个属性。

换句话说，所有的 ActiveX 控件都具有 Builder 程序要用到的同样接口，可以将一个控件替换成其他控件，而且还仍然使用同样的接口去处理它的属性。你构建的实际程序能按常规方式使用这些类，每个类都有自己的方法，但从构造器的观点来看，它们都是一样的。

图 15.5 Visual Studio.NET 的 Properties 接口

15.6 桥接模式的效果

1. 桥接模式可以保持客户端程序的接口不变，而允许读者修改显示类或要使用的类。这可以防止重新编译一系列复杂的用户接口模块，而只需要重新编译Bridge和实际的最终显示类。
2. 可以分别扩展实现类和 Bridge 类，二者之间通常不会有相互作用。
3. 对客户端程序很容易隐藏实现细节。

15.7 思考题

绘制一个股票的收益曲线时，通常要显示价格和一段时间的收益率，而在绘制公共基金曲线时，通常要显示价格和每季度的收益。考虑如何使用 Bridge 来完成两种显示。

15.8 随书附带光盘中的程序

Bridge from List to Grid	`\Bridge\BasicBridge`
Sorted Bridge	`\Bridge\SortBridge`

第16章 组合模式

通常在程序员开发的系统中，组件既可以是单个的对象，也可以是对象的集合。组合模式（Composite Pattern）包括了这两种情况，可以用组合模式构建部分-整体层次结构或构建数据的树形表示。总而言之，组合就是对象的集合，其中的每个对象既可以是一个组合，也可以是简单的对象。在树的术语中，对象可以是带有其他分支的节点，也可以是叶子。

开发方面存在的问题是，对组合中的所有对象都要求具有一个简单的、单一访问接口并要求能够区分节点与叶子，这二者是相互矛盾的。节点可以有孩子并允许添加孩子，而叶子节点不允许有孩子，在某些实现中要防止对它们添加孩子节点。

有些作者建议，为节点和叶子创建一个单独的接口，叶子具有如下方法：

```
public string getName ( );
public float getValue ( );
```

节点还包括另外一些方法：

```
public ArrayList elements ( );
public Node getChild ( string nodeName );
public void add ( Object obj );
public void remove ( Object obj );
```

这就给我们遗留下一个程序设计问题：在构建组合时，要确定哪个元素是哪种类型。设计模式建议：不管元素是组合型的还是简单型的，都应该有相同的接口，这点很容易做到，但还是有一个问题，即当对象实际上是一个叶子时，getChild 方法应该完成什么样的操作。

C#使这个问题变得很容易，由于每个节点或叶子都能返回一个关于孩子节点的 ArrayList，如果没有孩子节点，count属性返回值为零。这样，如果我们能获得每个元素的孩子节点的 ArrayList，就能通过检查 count 属性，迅速判断出元素是否有孩子。

同样困难的是，向组合中的元素添加叶子或从组合中删除叶子的问题。一个非叶子节点可以添加孩子作为叶子，但叶子节点不能这样做。尽管如此，我们还是希望组合中的所有部件都具有相同的接口。必须防止向叶子节点添加孩子，可以这样设计叶子节点类：如果程序试图向它添加一个节点，就报错。

16.1 一个组合的实现

我们考虑在初期只有一个人的小公司，他当然就是 CEO，尽管在初期他过于繁忙，不会考虑到这一点。接下来，他雇佣了两个人来分别管理销售和生产，很快这两个人又分别雇佣了另外一些助手，帮助做广告、运输等业务，这两个人于是成为公司的两位副总经理。随着公司的日益兴旺，公司人员持续增长，最后形成图 16.1 的组织成员图。

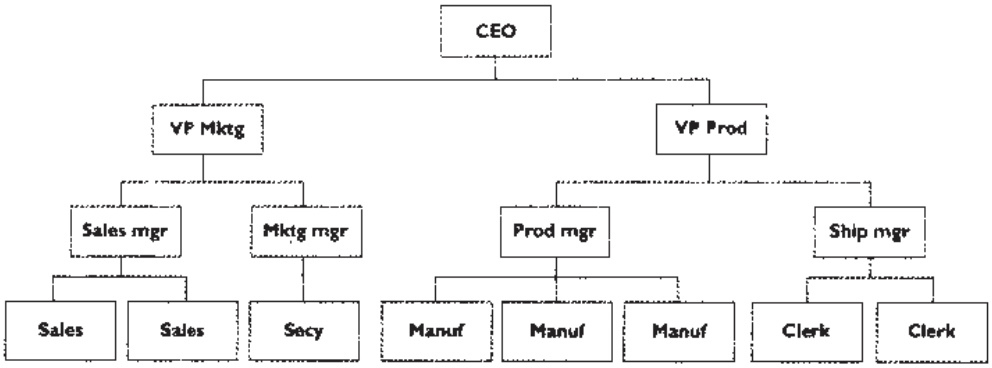


图 16.1 一个典型的组织图表

16.2 计算薪水

如果公司盈利，公司中的每个成员都会得到一份薪水，在任何时候都会要求计算从每个员工到整个公司的控制成本。将控制成本定义为员工及其所有下属的薪水。这是一个理想的组合例子。

- 每个雇员的成本就是他或她的薪水（和津贴）。
- 领导一个部门的雇员的成本是他或她的薪水加上其下属的薪水。

我们希望用一个简单的接口就能正确地生成薪水总和，不管雇员是否有下属。

```
float getSalaries ( );          //get salaries of all
```

此时会认识到，让所有的组合在接口中都具有相同的标准方法名是不太现实的，这里尽量让公有方法名与我们实际开发的类相关，所以，这时没有采纳通用的方法名（如 getValue），而是使用 getSalaries。

16.3 Employee 类

我们将公司设想为由节点（管理人员和雇员）构成的一个组合。使用一个类表示所有的员工是可以的，但由于每个层次的雇员有不同的属性，所以至少定义两个类（Employee 类和 Boss 类）会更有效。Employee 是叶子，他们下面没有雇员，Boss 是节点，他们下面可以有雇员节点。

我们先创建 AbstractEmployee 类，然后从中派生出具体的 Employee 类。

```
public interface AbstractEmployee       {
      float getSalary();                //get curent salary
      string getName();                 //get name
      bool isLeaf();                    //true if leaf
      void add(string nm,float salary);     //add subordinate
      void add(AbstractEmployee emp);       //add subordinate
      IEnumerator getSubordinates();        //get subordinates
      AbstractEmployee getchild();          //get child
      float getSalaries();              //get sum of salaries
}
```

C# 中有一个内置的枚举接口叫 IEnumerator，该接口由下列方法组成。

```
bool MoveNext ( );          //False if no more left
object Current ( )          //get Current object
```

```
void Reset ( );              //move to first
```

这样，就可以创建一个返回一个Enumerator的AbstractEmployee接口。使用下面的方法遍历枚举量，枚举量允许为空。

```
e.Reset ( );
while ( e.MoveNext ( ) )   {
  Emp = ( Employee ) e.Current ( );
   //..do computation..
}
```

Enumerator当然可以为空，因而，可以将它用于组合里的节点和叶子两种情况。

具体的Employee类保存了每个雇员的姓名和薪水，需要时可以取出来。

```
public class Employee :AbstractEmployee    {
      protected float salary;
      protected string name;
      protected ArrayList subordinates;
      //------
      public Employee(string nm,float salary)      {
            subordinates=new ArrayList();
            name=nm;
            salary=salry;
      }
      //------
      public float getSalary(){
            return salary;
      }
      //------
      public string getName(){
            return name;
      }
      //------
      public bool isLeaf(){
            return subordinates.count==0;
      }
      //------
      public virtual AbstractEmployee getChild(){
            return null;
      }
```

Employee类必须有add，remove，getChild和subordinates等方法的具体实现过程。由于Employee是叶子，所以，这些方法都要返回某种错误提示。subordinates方法可以返回一个空值，但是，如果subordinates方法返回一个空的枚举量，会使程序更具有一致性。

```
public IEnumerator getSubordinates ( ) {
      return subordinates.GetEnumberator ( );
}
```

由于Employee类的成员不能有下属，它的add和remove方法就必须产生错误提示。如果在Employee类调用这些方法，就让它们抛出一个异常。

```
public virtual void add(string nm, float salary) {
  throw new Exception(
       "No subordinates in base employee class");
  }
  //------
  public virtual void add(AbstractEmployee emp) {
       throw new Exception("No subordinates in base employee class");
}
```

16.4 Boss 类

Boss类是Employee的一个子类，它还保存了下属雇员，我们将下属雇员存储在一个称为subordinates的ArrayList中，可通过枚举量返回下属。这样，如果某一个Boss临时缺少Employee，枚举量置为空就可以了。

```
public class Boss:Employee{
      public Boss(string name,float salary):base(name,salary){}
      //------
      public override void add(string nm,float salary){
            AbstractEmployee emp=new Employee(nm,salary);
            subordinates.Add(emp);
      }
      //------
      public override void add(AbstractEmployee emp){
            sbuordinates.Add(emp);
      }
      //------
```

如果要得到某个管理者的雇员列表，可以直接从ArrayList中获得他们的Enumeration。同样地，可以使用同一个ArrayList返回任意一个雇员及其下属的薪水总和。

```
  public float getSalaries() {
        float sum;
        AbstractEmployee esub;
        //get the salaries of the boss and subordinates
        sum = getSalary();
        IEnumerator enumSub = subordinates.GetEnumerator();
        while (enumSub.MoveNext())  {
              esub = (AbstractEmployee)enumSub.Current;
              sum += esub.getSalaries();
        }
  return sum;
}
```

注意，这个方法是从当前雇员的薪水开始，然后对每一个下属调用getSalaries( )方法。当然，这是一个循环过程，任何拥有下属的雇员都会包含在内。图16.2给出了这些类的图表。

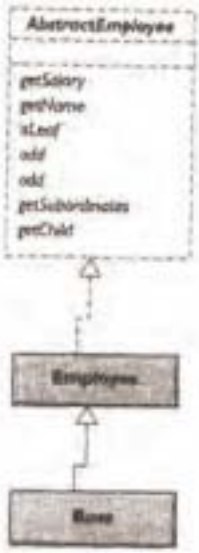


图 16.2 AbstractEmployee类以及由它派生出的Employee类和Boss类

16.5 构造 Employee 树

我们先创建一个 CEO Employee，然后添加他的下属，再添加这些人的下属，如下所示。

```
private void buildEmployeeList() {
       prez = new Employee("CEO", 200000);
       marketVP = new Employee("Marketing VP", 100000);
       prez.add(marketVP);
       salesMgr = new Employee("Sales Mgr", 50000);
       advMgr = new Employee("Advt Mgr", 50000);
       marketVP.add(salesMgr);
       marketVP.add(advMgr);
       prodVP = new Employee("Production VP", 100000);
       prez.add(prodVP);
       advMgr.add("Secy", 20000);
       //add salesmen reporting to sales manager
       for (int i = 1; i<=5; i++){
              salesMgr.add("Sales" + i.ToString(),
                           rand_sal(30000));
       }

       prodMgr = new Employee("Prod Mgr", 40000);
       shipMgr = new Employee("Ship Mgr", 35000);
       prodVP.add(prodMgr);
       prodVP.add(shipMgr);

       for (int i = 1; i<=3; i++){
            shipMgr.add("Ship" + i.ToString(), rand_sal(25000));
       }
       for (int i = 1; i<=4; i++){
            prodMgr.add("Manuf" + i.ToString(), rand_sal(20000));
       }
 }
```

一旦构建好这个组合结构，就可以创建一个可视化的 TreeView 列表：先创建顶端节点，然后不断地调用 addNode( )方法，直到节点中的所有叶子都加了进去。

```
private void buildTree() {
       EmpNode nod;
       nod = new EmpNode(prez);
       rootNode = nod;
       EmpTree.Nodes.Add(nod);
       addNodes(nod, prez);
 }
```

为了简化 TreeNode 对象的处理，我们派生一个 EmpNode 类，它将一个 Employee 实例作为参数。

```
public class EmpNode:TreeNode    {
       private AbstractEmployee emp;
       public EmpNode(AbstractEmployee aemp ):
                     base(aemp.getName ()) {
              emp = aemp;
       }
       //-----
       public AbstractEmployee getEmployee() {
              return emp;
       }
 }
```

最终程序的显示如图 16.3 所示。

图 16.3 以 TreeView 控件显示的全体组织成员

在这个程序的实现里，单击一个雇员，他的成本（薪水总和）显示在图的底部。这个计算过程反复调用了 getChild( )方法，来获得该雇员的所有下属。

```
private void EmpTree_AfterSelect(object sender,
             TreeViewEventArgs e) {
       EmpNode node;
       node = (EmpNode)EmpTree.SelectedNode;
       getNodeSum(node);
}
//------
private void getNodeSum(EmpNode node) {
       AbstractEmployee emp;
       float sum;

        emp = node.getEmployee();
        sum = emp.getSalaries();
        lbSalary.Text = sum.ToString ();
 }
```

16.6 自我升职

我们假设有这样一种情况，一个基层雇员还保留现有工作，但他拥有了新的下属。例如，要求一名销售员去指导销售学员。对于这种情况，比较方便的做法是：在 Boss 类中提供一个方法，它将 Employee 转成 Boss。这里另外提供一个构造函数，它将一个雇员转换成老板。

```
public Boss ( AbstractEmployee emp ) :
          base ( emp.getName ( ), emp.getSalary ( ) )     {
```

16.7 双向链表

在前面的实现里，我们在每个 Boss 类的集合（Collection）里保留了一个指向每名下属的索引，这就意味着，可以沿着总经理到雇员的链向下移动，但是不能反向移动来查找一名雇员的管理者。这很容易补救：为每个 AbstractEmployee 子类提供一个构造函数，它包含一个指向父节点的索引。

```
public class Employee :AbstractEmployee    {
       protected float salary;
       protected string name;
       protected AbstractEmployee parent;
       protected ArrayList subordinates;
       //------
       public Employee(AbstractEmployee parnt,
                     string nm, float salry)          {
             subordinates = new ArrayList();
             name = nm;
             salary = salry;
             parent = parnt;
}
```

接下来，就可以快速地向上访问树，生成一个汇报链表。

```
private void btShowBoss_Click(object sender, System.EventArgs e) {
       EmpNode node;
       node = (EmpNode)EmpTree.SelectedNode;
       AbstractEmployee emp = node.getEmployee ();
       string bosses = "";
       while(emp != null) {
             bosses += emp.getName () +"\n";
             emp = emp.getBoss();
       }
       MessageBox.Show (null, bosses,"Reporting chain");
}
```

参见图 16.4。

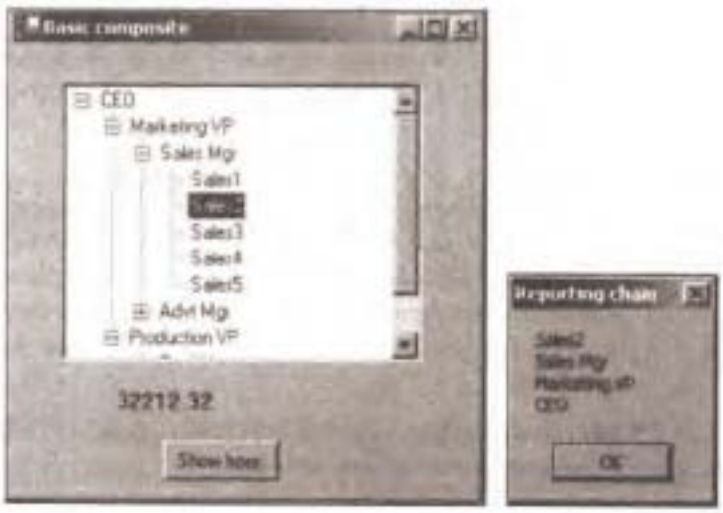


图 16.4 组合的树形列表显示，右边显示的是父节点

16.8 组合模式的效果

组合模式定义了包含简单对象和复杂组合对象的类层次结构，并使它们对客户端程序具有一致性。由于这种简化，客户端可以变得相当简单，因为节点和叶子可以用同样的方式去处理。

组合模式使得向集合添加新类型的组件变得很容易，只要这些组件提供一个相似的编程接口。另一方面，这也有缺点，就是使你的程序更加一般化，读者会发现很难限制某个类，而通常都希望能做到这一点。

16.9 一个简单的组合

组合模式的意图是，允许读者构建一棵由各种相关类组成的树，即使某些类具有不同的属性，某些类是叶子不能有孩子。然而，对于非常简单的案例，有时只需用一个简单的类展示父节点和叶子的行为。在SimpleComposite例子中，我们创建一个Employee类，它总是包含下属雇员的ArrayList。这个雇员集合可以是空的，也可以有元素，这取决于getChild方法和remove方法返回值的性质。在这个简单的例子里，我们不报错，而且总是允许提升叶子节点，使之可以具有孩子。换句话说，我们总是允许执行add方法。

尽管读者没有将这种自动提升当做缺点，但在具有大量叶子的系统中，每个叶子都要保存一个初始化的、无用的集合，这很浪费空间。在叶子节点相当少的例子中，这个问题不太严重。

16.10 .NET 中的组合

在.NET中读者会发现，添加到TreeView中的Node对象类实际上就是一个简单的组合模式。读者还会发现，组合模式描述了用户界面程序中的窗体、框架和控件的层次结构。同样地，工具栏是容器，每个容器可以包含其他任何数量的容器。

任何一个容器可以包含诸如按钮、复选框和文本框等组件，其中每个组件都是叶子，它们不能再有孩子。容器还可以包含列表框和表格，它们可以作为叶子，也可以再包含其他图形组件。读者可以使用控件集合向下遍历这样的组合树。

16.11 其他实现问题

排序部件。在某些程序里，部件的顺序是很重要的。如果该顺序和添加到父节点的顺序不同，则父节点必须为返回正确的节点顺序做额外的工作。例如，原始集合是按字母顺序排序，要返回一个重新排序的集合。

将结果缓冲存储。如果要经常访问由一系列孩子部件计算出的数据，例如访问我们前面计算的薪水，那么，将这些计算结果缓存到父节点中是很有好处的。然而，除非计算量相当大，并且能完全保证底层的数据没有改变，否则不值得去这样做。

16.12 思考题

1. 一个棒球队可以看做单个队员的聚合。如何使用组合模式表示队员和球队的成绩？
2. 一个超市的产品部门需要跟踪食品的销售情况。如何用组合模式帮助实现这个程序？

16.13 随书附带光盘中的程序

Composite shows tree	`\Composite\Composite`
Composite that uses both child links and parent links	`\Composite\DlinkComposite`
Simple Composite of same employee tree that allows any employee to move from leaf to node	`\Composite\SimpleComposite`

第17章 装饰模式

装饰模式（Decorator Pattern）给我们提供了这样一种方式：改变单个对象的行为，但不需要创建一个新的派生类。假设我们有一个使用了八个对象的程序，其中三个对象需要另外一个属性。读者可以为这三个对象创建一个派生类，在多数情况下，这是一个完全可以接受的方案。然而，如果这三个对象中的每个对象都要求有不同的属性，这就意味着要创建三个派生类。更进一步，如果其中一个类具有其他两个类中的属性，可能就要创建更复杂的类，这既容易造成混乱也没有必要。

例如，我们想在一个工具栏中画一些有特定边框的按钮。如果我们创建一个新的Button派生类，这表示新类中的所有按钮都有同样的边框，即使我们不需要。

我们换种做法，创建一个Decorator类来装饰按钮，然后从Decorator类派生出多个特定的Decorator，每一个派生类完成一种特定的装饰。为了装饰一个按钮，Decorator必须是一个派生于可视化环境的对象，这样它才能接收paint方法调用，并将调用传递给被装饰对象的图形方法。对象容器在继承上更占优势（而不是装饰），这是另外一种情况。Decorator是一个图形对象，但它包含了要装饰的对象。它可以截取某些图形方法的调用，执行一些额外的计算，再将调用传递给下面要装饰的对象。

17.1 装饰一个CoolButton

最近的Windows应用程序，例如Internet Explorer和Netscape Navigator，都有一行平面的、无边界的按钮，当鼠标移到上面时，按钮高亮度显示并显出轮廓边界。一些Windows程序员将这样的工具栏称为CoolBar，将这样的按钮称为CoolButton。C#控件中没有类似的按钮行为，但我们可以通过装饰一个面板，并用它包含一个按钮来实现这种行为。在本例中，我们用下列方式装饰按钮：画黑白边界线来高亮显示按钮，画灰色线删除按钮边界。

接下来考虑如何创建这个Decorator。“Design Patterns”中建议：Decorator应该派生于某个通用的可视化组件类，发给实际按钮的每条信息都应该由Decorator向前转发。在C#中，实际上这并不是全部，但如果我们用容器作为Decorator，所有的事件都要传递给所包含的控件。

“Design Patterns”进一步建议，Decorator类应该是抽象类，要从该抽象类中派生出所有能实际工作的（具体的）Decorator。我们在实现中定义了一个Decorator接口，它接收了需要截取的鼠标事件和绘制事件。

```
public interface Decorator        {
       void mouseMove(object sender, MouseEventArgs e);
       void mouseEnter(object sender, EventArgs e);
       void mouseLeave(object sender, EventArgs e);
       void paint(object sender, PaintEventArgs e);
 }
```

为了实现具体的装饰，我们从Panel类派生出CoolDecorator，让它作为包含被装饰按钮的容器。

接下来考虑如何实现CoolButton，我们真正需要做的是，在按钮高亮显示的时候画出按钮边界的白色线和黑色线，在非高亮显示时画灰色线。当检测到按钮的MouseMove事件时，接下来的Paint

事件应该画高亮显示的线；当鼠标离开按钮区域时，接下来的 Paint 事件应该画出灰色的轮廓。我们用下列方式完成上述工作：先设定一个 mouse_over 标志，然后通过调用 Refresh 方法使之重画。

```
public void mouseMove(object sender, MouseEventArgs e){
            mouse_over = true;
      }
      public void mouseEnter(object sender, EventArgs e){
            mouse_over = true;
            this.Refresh ();
      }
      public void mouseLeave(object sender, EventArgs e){
            mouse_over = false;
            this.Refresh ();
      }
```

下面是具体的 paint 事件。

```
public virtual void paint(object sender, PaintEventArgs e){
      //draw over button to change its outline
      Graphics g = e.Graphics;
      const int  d = 1;
      //draw over everything in gray first
      g.DrawRectangle(gPen, 0, 0, x2 - 1, y2 - 1);
      //draw black and white boundaries
      //if the mouse is over
      if( mouse_over) {
            g.DrawLine(bPen, 0, 0, x2 - d, 0);
            g.DrawLine(bPen, 0, 0, 0, y2 - 1);
            g.DrawLine(wPen, 0, y2 - d, x2 - d, y2 - d);
            g.DrawLine(wPen, x2 - d, 0, x2 - d, y2 - d);
      }
}
```

17.2 处理 Decorator 中的事件

这里构造了一个具体的 Decorator，它包含鼠标方法和绘制方法，现在需要把事件处理系统和这些方法联系起来。我们为鼠标进入和停留事件创建一个 EventHandler 类，为鼠标移动和离开事件创建一个 MouseEventHandler，在具体的 Decorator 构造函数中用这种方法完成上述工作。注意，我们捕获的事件是被包含的按钮的事件，而不是外围 Panel 的事件，这一点很重要。所以，我们这里将事件处理程序添加到按钮上。

```
public CoolDecorator(Control c) {
contl = c;        //copy in control
//mouse over, enter handler

EventHandler evh = new EventHandler(mouseEnter);
      c.MouseHover += evh;
      c.MouseEnter+= evh;
//mouse move handler
c.MouseMove += new MouseEventHandler(mouseMove);
c.MouseLeave += new EventHandler(mouseLeave);
```

同样为 paint 事件创建一个 PaintEventHandler。

```
//paint handler catches button's paint
c.Paint += new PaintEventHandler( paint);
```

17.2.1 考虑布局

如果创建一个包含按钮的Windows窗体，GUI设计器会自动生成将控件添加到该窗口控件数组中的代码。我们对代码做一些修改：将按钮添加到新面板的控件数组中，将面板添加到窗口的控件数组中，再从窗口控件数组中移去按钮。下面是Form初始化方法中添加面板和移去按钮的代码。

```
//add outside decorator to the layout
//and remove the button from the layout
this.Controls.AddRange ( new System.Windows.Form.Control [ ]  { cdec } );
this.Controls.Remove ( btButtonA);
```

下面是将按钮添加到Decorator面板的代码。

```
public CoolDecorator ( Control c )   {
     contl = c;              //copy in control
     //add button to controls contained in panel
     this.Controls.AddRange ( new Control [ ]  { contl }  );
```

17.2.2 控制尺寸和位置

我们将按钮放置在一个面板中来装饰它，需要改变它的坐标和尺寸，这样面板就拥有了按钮的尺寸和坐标，按钮在面板中的位置是（0，0），这些过程发生在CoolDecorator的构造函数中。

```
this.Location = p;
contl.Location =new Point(0,0);

this.Name = "deco"+contl.Name;
this.Size = contl.Size;
x1 = c.Location.X - 1;
y1 = c.Location.Y - 1;
x2 = c.Size.Width;
y2 = c.Size.Height;
```

还要在构造函数里创建Pen的实例，Paint方法会用到这些实例。

```
//create the overwrite pens
gPen = new Pen ( c.BackColor, 2 );     //gray pen overwrites borders
bPen = new Pen ( Color.Black, 1 );
wPen = new Pen ( Color.White, 1 );
```

图17.1所示为鼠标停留在一个按钮上时的程序运行结果。

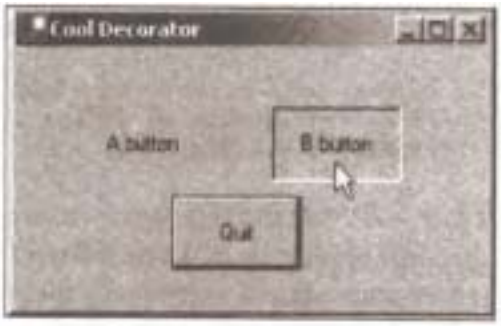


图17.1 A button和B button都是CoolButton，当鼠标停留在它们上面时画出轮廓。这里是B button的轮廓被画出

17.3 多个 Decorator

我们现在已经清楚一个Decorator是如何工作的，多个Decorator时又如何呢？比如说，我们想用另一种装饰——一条红色对角线装饰CoolButton。

这稍微复杂一点，因为我们只需将CoolDecorator包含在另外一个Decorator面板中，以此完成进一步的装饰。惟一真正要改变的是，不仅需要包装在另一个面板中的面板实例，还需要被装饰的中心对象（这里是一个按钮），因为我们需要把paint例程和中心对象的paint方法联系在一起。

因此，要为自己的Decorator创建一个构造函数，它将包含的面板和按钮都当做控件。

```
public class CoolDecorator :Panel, Decorator      {
       protected Control contl;
       protected Pen bPen, wPen, gPen;
       private bool mouse_over;
       protected float x1, y1, x2, y2;
//--------------------------------
       public CoolDecorator(Control c, Control baseC) {
       //the first control is the one layed out
       //the base control is the one whose paint method we extend
       //this allows for nesting of decorators
              contl = c;
              this.Controls.AddRange(new Control[] {contl});
```

然后添加事件处理程序，paint事件处理程序必须连接到基础控件（即按钮）。

```
//paint handler catches button's paint
      baseC.Paint += new PaintEventHandler ( paint );
```

我们将paint方法声明为virtual，以便能够覆盖它，如下所示。

```
public virtual void paint ( object sender, PaintEventArgs e ) {
      //draw over button to change its outline
      Graphics g =e.Graphics;
```

完成上述工作后，就能用最简单的方法编写绘制红色对角线的SlashDecorator，让它直接派生于CoolDecorator。可以重用所有的基类方法，面只扩展CoolDecorator中的paint方法，这样做可以节省许多工作量。

```
public class SlashDeco:CoolDecorator  {
       private Pen rPen;
       //----------------
       public SlashDeco(Control c, Control bc):base(c, bc) {
         rPen = new Pen(Color.Red, 2);
       }
       //----------------
       public override void paint(object sender,
                     PaintEventArgs e){

              Graphics g = e.Graphics;
              x1=0; y1=0;
              x2=this.Size.Width;
              y2=this.Size.Height;
              g.DrawLine (rPen, x1, y1, x2, y2);
        }
     }
```

我们就这样完成了显示两种按钮的最终程序，其运行结果如图17.2所示。图17.3所示为类图。

图 17.2　A CoolButton 也可以用 SlashDecorator 来装饰

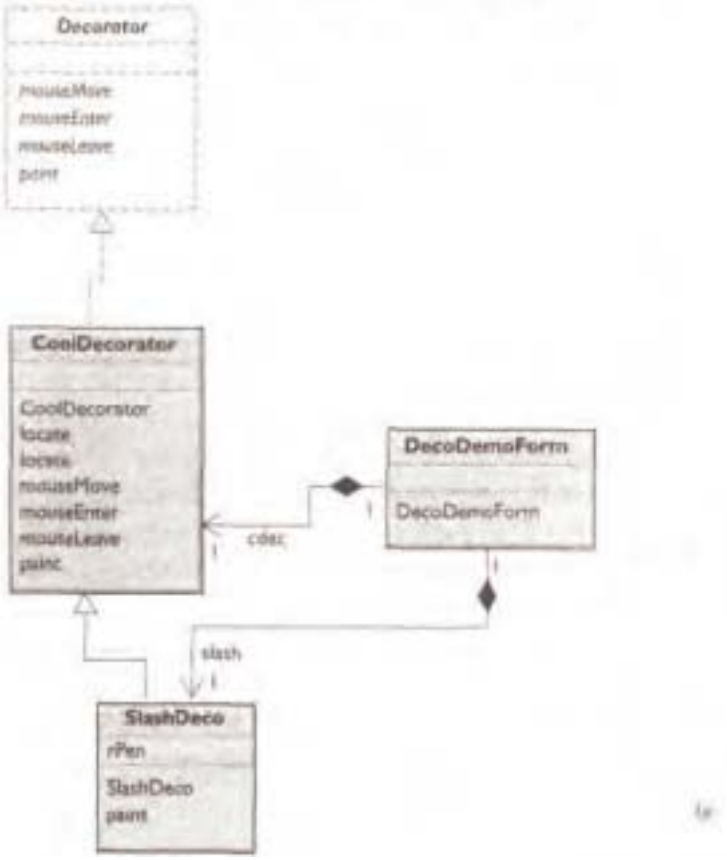


图 17.3　Decorator 类及其两个专门的 Decorator 实现类的 UML 类图

17.4　非可视化的 Decorator

Decorator 当然不仅限于增强可视类对象的功能，读者可以用同样的方式增加或修改任何对象中的方法。实际上，非可视化的对象更容易装饰，因为要截获和向前传递的方法很少。任何时候将一个类的实例放在另外一个类中，让外部类对它操作，本质上就是在装饰那个内部类，在 Visual Studio.NET 中，这是特别常用的程序设计工具。

17.5　Decorator，Adapter 和 Composite

正如“Design Patterns”中提到的，这些类之间存在着本质上的相似性，读者可能已经认识到这一点。适配器也像是在装饰一个现有的类，但它的功能是将一个或多个类的接口转换成另一个接口，这对某个特定的程序来说是很方便的。Decorator 将方法添加到类的特定实例里，而不是所有的

实例里。读者还可能认为，包含一个组件的组合本质上就是Decorator，我们再重复一次，它们的意图是不一样的。

17.6 装饰模式的效果

装饰模式提供了一种给一个类添加职责的方式，它比使用继承更加灵活，因为它能将职责加到类的指定实例中，它也允许读者定制一个类，而无需在继承层次结构中创建高层子类。“Design Patterns”中指出了装饰模式的两个缺点，一个是Decorator和它包含的组件是不一样的，这样，检测对象类型时会失败；另一个缺点是装饰模式会使一个系统带有“大量的小对象”，对于维护代码的程序员来说，它们看起来都差不多，维护起来会很头痛。

在生成架构中，装饰模式和外观模式会产生相似的图像，但在设计模式的术语中，外观模式是将复杂系统隐藏在一个简单接口里，而装饰模式通过包装一个类来增加功能。我们接下来会学习外观模式。

17.7 思考题

1. 当有人在一个网格单元中输入了一个错误的值，读者想通过改变该行的颜色来指出这个问题时，如何使用Decorator?
2. 公共基金是一个股票的集合，每支股票包含了一段时期价格的一个数组或集合。如何用Decorator为每支股票以及整个基金生成一个股票收益的报告?

17.8 随书附带光盘中的程序

C# Cool Button Decorator	`\Decorator\Cooldecorator`
C# Cool Button and Slash Decorator	`\Decorator\Redecorator`

第18章　外观模式

外观模式（Facade Pattern）可以将一系列复杂的类包装成一个简单的封闭接口。随着程序开发的进展，复杂性逐步增加。实际上，由于热衷于使用设计模式，有时候这些模式会生成许多的类，以至于难以理解程序的控制流程。此外，程序中会有很多复杂的子系统，每个子系统都具有自己的复杂接口。

外观模式通过对这些子系统提供一个简化的接口来降低这种复杂性，某种情况下这种简化会减少下层类的灵活性，但它还是为有经验的用户提供了所需要的全部功能，这些用户当然还可以访问下层的类和方法。

幸运的是，我们不必编写一个复杂的系统来证明外观模式是有用的。C#拥有一个称为ADO.NET的接口，它提供了一系列连接到数据库的类，只要数据库制造商提供了ODBC连接类——市场上的每个数据库几乎都提供，读者就可以连接到任何数据库上。我们先花几分钟复习一下如何使用数据库以及数据库如何工作等知识。

18.1　什么是数据库

数据库是采用某种文件结构的一系列信息表，允许读者依据不同标准访问这些表、选择其中的列、排序及选择行。数据库通常都用索引与这些表中的许多列相关联，这样我们就能快速地访问它们。

数据库在计算中比其他类型的结构使用更频繁，在雇员档案和工资单系统中，在旅游规划系统及各种产品制造销售系统中，读者会发现数据库都是作为中心元素出现的。

在雇员档案系统中，可能设想会有一个包括雇员名字、地址、薪水、扣税和补助等内容的表。我们考虑一下如何组织这些项目。可以这样设计一个表，让它包括雇员名字、地址和电话号码等内容，其他要存储的信息应该包括薪水、薪水范围、上次加薪日、下次加薪日和雇员绩效考核等。

所有这些信息都应该放在一张表中吗？可以肯定不是这样。各种等级雇员的薪水范围对雇员来说是不变的，因此，读者可以在雇员表中只存储雇员等级，把薪水范围存储在由等级号指向的另一张表中。考虑表18.1中的数据。

表18.1　雇员姓名表和薪水类型表

Key（关键字）	Lastname（姓）	SalaryType（薪水类型）
1	Adams	2
2	Johnson	1
3	Smyth	3
4	Tully	1
5	Wolff	2

SalaryType	Min（最小值）	Max（最大值）
1	30 000	45 000
2	45 000	60 000
3	60 000	75 000

SalaryType列中的数据指向了第二张表。我们可以为类似的事情（如每个州的居住状况、税额、健康计划扣缴等）设计同样的表。每个表都会有一个主关键字列，如每张表中左边的那一列及其他几列数据。构造数据库中的表既是一门艺术，也是一门科学。通常说表是属于第一范式、第二范式或第三范式的，缩写为1NF，2NF或3NF。

- 第一范式。表中的每个单元有且只有一个值（而不是一组值）。

- 第二范式。满足1NF，且每个非关键字列完全依赖于关键字列。这意味着，一行中的主关键字和其余单元之间存在一对一的关系。
- 第三范式。满足2NF，且所有的非关键字列都是相互独立的。这意味着，任何数据列所包含的值都不能由其他列的数据计算出。

目前，几乎所有数据库中的表都是按第三范式（3NF）构建的。这意味着，库中通常会有相当多数量的表，每个表只有相对较少的数据列。

18.2 从数据库中取出数据

假设我们要生成一个包含雇员及其薪水范围的表。该表在数据库中并不直接存在，但可以通过查询数据库构造出来。我们希望有一张类似于表18.2中数据的表。

表18.2 按名字进行排序的雇员薪水表

Name	Min	Max
Adams	$45 000.00	$60 000.00
Johnson	$30 000.00	$45 000.00
Smyth	$60 000.00	$75 000.00
Tully	$30 000.00	$45 000.00
Wolff	$45 000.00	$60 000.00

我们还想对数据按薪水递增次序排序，如表18.3所示。发现通过查询就能得到具有这样的格式的表。

表18.3 按薪水幅度进行排序的雇员薪水表

Name	Min	Max
Tully	$30 000.000	$45 000.000
Johnson	$30 000.000	$45 000.000
Wolff	$45 000.000	$60 000.000
Adams	$45 000.000	$60 000.000
Smyth	$60 000.000	$75 000.000

```
SELECT DISTINCTROW Employees.Name, SalaryRanges.Min
SalaryRanges.Max FROM Employees INNER JOIN SalaryRanges ON
Employees.SalaryKey = SalaryRanges.SalaryKey
ORDER BY SalaryRanges.Min;
```

这种语言称为结构化查询语言或SQL，它实际上是当前所有现存数据库的语言。几年来出现了多种SQL标准，绝大部分PC数据库都支持ANSI标准。SQL-92标准被认为是底层标准，以后又出现了多个更新版本。然而，不是所有的数据库都能很好地支持新版本的SQL，绝大多数数据库都提供了不同扩展形式的SQL，以发挥出它们独有的特色。

18.3 数据库的种类

由于PC机已成为一种主要的办公工具，许多流行的数据库都是为了能运行在PC机上而开发的，这包括初级数据库，如Microsoft Works，也包括较复杂的数据库，如Approach，dBase，Borland Paradox，Microsoft Access和FoxBase。

PC数据库的另一类型是使用大量PC机终端访问装在一台服务器上的数据库，包括IBM DB/2，Microsoft SQL Server，Oracle和Sybase，所有这些数据库产品支持的都是各种类似的SQL版本，因

此，这些产品最初看起来都是可互换的。当然，实际上并不能互换，原因是每种产品的设计都包含了不同的性能特色，每种产品都具有不同的用户界面和程序设计接口。读者可能会认为，因为它们都支持SQL，所以它们的编程方式也会一样，实际上完全相反，每种数据库都有自己的接收SQL查询的方式、自己的返回结果的方式。这就是下面要提到的标准（即ODBC）出现的原因。

18.4 ODBC

如果我们编写出的代码能独立于特定厂商的数据库，并且不需要改变调用程序，就能从任何一个数据库中获得同样的结果，能做到这一点就好了。如果我们只是为各种数据库编写一些包装程序，使它们具有同样的程序设计接口，这一点很容易做到。

Microsoft在1992年首先完成了这项壮举，它发布了一个叫做对象数据库互联（Object Database Connectivity，ODBC）的标准，该标准被认为是所有Windows下的数据库互联的解决方案。跟所有的第一版软件一样，它在成长过程中也经历了一些挫折，1994年发布了另外一个版本，该版本速度更快也更稳定，同时也是第一个32位版本。另外，ODBC开始移植到Windows以外的其他平台，目前在PC机和工作站上已相当流行了，几乎每一个主流数据库厂商都提供ODBC驱动程序。

18.5 数据库的结构

一个数据库的最底层是由若干表组成，每张表具有若干个命名的数据列，表之间存在着某些关系。这些都太复杂难以弄清楚，我们希望在处理数据库的代码中能对此有所简化。

C#和Visual Studio.NET中的其他所有组件都使用一种新的数据库访问模型，叫做ADO.NET，它针对的是ActiveX数据对象。ADO.NET的设计准则是，在程序和数据库之间定义一个连接，不定时地使用连接去处理多数的计算过程，这些计算实际发生在本地机器上的非连接对象里。另外，ADO.NET主要是用XML定义数据库和程序之间相互传递的对象，尽管也可以用ADO.NET的内置类访问这种数据描述。

18.6 使用ADO.NET

C#中的ADO.NET由很多种相互联系的对象组成。由于我们要执行的操作相对简单，外观模式是一种管理这些操作的理想方式。

- **OleDbConnection**：该对象表示对数据库的实际连接。可以让该类的一个实例一直保持可用，但只在需要时打开和关闭连接。使用完后，一定要在垃圾收集前关闭连接。
- **OleDbCommand**：该类表示发送给数据库的一条SQL命令，它可以有返回结果也可以没有。
- **OleDbDataAdapter**：在数据库和本地数据集（DataSet）之间提供了一个移动数据的桥梁。可以指定一个OleDbCommand、一个DataSet和一个连接。
- **DataSet**：一个或多个数据库表的一种表示，或者是本地机器上的一个查询所得结果的一种表示。
- **DataTable**：来自一个数据库或一个查询结果的一张数据表。
- **DataRow**：DataTable中的一行。

18.6.1 连接数据库

要连接一个数据库，需要在构造函数中为要使用的数据库指定一个连接字符串。例如，要使用一个Access数据库，连接字符串可以是下列形式：

```
string connectionString =
      "Provider = Microsoft.Jet.OLEDB.4.0;" +
      "Data Source = " + dbName
```

下面的代码完成实际的连接：

```
OleDbConnection conn =
      new OleDbConnection ( connectionString );
```

真正打开连接要通过调用Open方法。为了保证没有重复打开一个已经打开了的连接，可以先检查连接的状态。

```
private void openConnection ( )  {
      if ( conn.State = = ConnectionState.Closed )  {
           conn.Open ( );
     }
}
```

18.6.2 从数据库表中读取数据

要从一个数据库表中读取数据，需要用一个适当的Select语句和连接来创建ADOCommand。

```
public DataTable openTable (string tableName )  {
      OleDbDataAdapter adapter = new OleDbDataAdapter ( );
      DataTable dtable = null;
      string query = "Select * from " + tableName;
      adapter.SelectCommand = new OleDbCommand ( query, conn );
```

然后创建一个存放结果的数据集对象：

```
DataSet dset = new DataSet ("mydata");
```

接下来只需告诉命令对象，使用连接填充数据集。必须在FillDataSet方法中指定要填充的数据表名字，如下所示：

```
try {
       openConnection();
       adapter.Fill (dset);
}
       catch(Exception e) {
             Console.WriteLine (e.Message );
       }
```

数据集至少包含一张表，可以用索引或名字取出表并检查它的内容。

```
//get the table from the dataset
dtable = dset.Tables [  0 ];
```

18.6.3 执行一个查询命令

除了查询命令还可以是一条更复杂的SQL Select语句，执行一个Select查询的代码同前面的代码完全一样。下面给出的查询步骤放在了try块中，目的是为了能处理SQL或数据库的错误。

```
public DataTable openQuery(string query) {
       OleDbDataAdapter dsCmd = new OleDbDataAdapter ();
       DataSet dset = new DataSet ();
//create a dataset
       DataTable dtable = null;          //declare a data table
       try {
```

```
                //create the command
                dsCmd.SelectCommand =
                        new OleDbCommand(query, conn);
        //open the connection
                openConnection();
                //fill the dataset
                dsCmd.Fill(dset, "mine");
                //get the table
                dtable = dset.Tables[ 0] ;
                //always close it
                closeConnection();
                //and return it
                return dtable;
                catch(Exception e) {
                        console.WriteLine (e.Message);
                        return null;
                }
}
```

18.6.4 删除表中的内容

使用SQL语句“Delete * from Table”可以删除表中的内容。然而，由于它不是一条Select命令，并且没有连接到本地表，所以只能使用OleDbCommand对象的ExecuteNonQuery方法。

```
public void delete() {
        //deletes entire table
        conn = db.getConnection();
        openConn();
        if (conn.State == ConnectionState.Open ) {
                OleDbCommand adcmd =
                new OleDbCommand("Delete * from " + tableName, conn);
                try{
                        adcmd.ExecuteNonQuery();
                        closeConn();
                }
                catch (Exception e) {
                        Console.WriteLine (e.Message);
                }
}
```

18.7 使用ADO.NET向数据库表添加数据行

向一张表添加数据的步骤是密切相关的，通常先从数据库取出当前版本的表，如果表很大，只能取出表的框架而得到一张空表。我们按下列步骤添加数据行。

1. 使用数据库中该表的名字创建一个DataTable。
2. 将它添加到一个数据集中。
3. 从数据库中取出数据填充该数据集。
4. 由DataTable获得一个新的行对象。
5. 填写行对象的各列数据。
6. 将该行加到表中。
7. 所有的行添加完后，用修改了的DataTable对象更新数据库。

添加数据行的过程如下面代码所示：

```
DataSet dset = new DataSet(tableName);    //create the data set
dtable = new DataTable(tableName);        //and a datatable
dset.Tables.Add(dtable);                  //add to collection
conn = db.getConnection();
openConn();                               //open the connection
OleDbDataAdapter adcmd = new OleDbDataAdapter();
//open the table
adcmd.SelectCommand =
      new OleDbCommand("Select * from " + tableName, conn);
OleDbCommandBuilder olecb = new OleDbCommandBuilder(adcmd);
adcmd.TableMappings.Add("Table", tableName);
//load current data into the local table copy
adcmd.Fill(dset, tableName);
//get the Enumerator from the Hashtable
IEnumerator ienum = names.Keys.GetEnumerator();
//move through the table, adding the names to new rows
while (ienum.MoveNext()) {
      string name = (string)ienum.Current;
      row = dtable.NewRow();      //get new rows
      row[ columnName] = name;
      dtable.Rows.Add(row);       //add into table
}
//Now update the database with this table
try {
      adcmd.Update(dset);
      closeConn();
      filled = true;
}
catch (Exception e) {
      Console.WriteLine (e.Message);
}
```

这就是表的编辑、更新过程，是ADO程序设计的中心内容。读者可以取出表，修改表，返回来更新数据库，也可以使用同样的过程编辑或删除数据行，并更新数据库使之做出同样的改变。

18.8 构建外观模式的各个类

我们要开发一个新的Facade应用，让它处理创建、连接和使用数据库，前面的叙述只是一个开端。为了完成其余部分，我们考虑表18.4，它列出了三家商店的食品价格。

表18.4 各商店食品价格表

Store（商店）	Food（食品）	Price（价格）
stop and shop	Apples	0.27
stop and shop	Oranges	0.36
stop and shop	Hamburger	1.98
stop and shop	Butter	2.39
stop and shop	Milk	1.98
stop and shop	Cola	2.65
stop and shop	Green beans	2.29
village Market	Apples	0.29
village Market	oranges	0.29
village Market	Hamburger	2.45
village Market	Butter	2.99
village Market	Milk	1.79
village Market	Cola	3.79
village Market	Green beans	2.19

（续表）

Store（商店）	Food（食品）	Price（价格）
Waldbaum's	Apples	0.33
Waldbaum's	Oranges	0.47
Waldbaum's	Hamburger	2.29
Waldbaum's	Butter	3.29
Waldbaum's	Milk	1.89
Waldbaum's	Cola	2.99
Waldbaum's	Green beans	1.99

如果这些信息都保存在一个数据库中就好了，这样我们就可以轻易地回答出这样的问题，“哪一个商店桔子价格最低”。这样的一个数据库应该包含三张表：Stores 表、Foods 表和 Prices 表。我们还需要保存三个表之间的关系。一种简单的处理方法是，创建一个 Stores 表，它包括 StoreName 和 StoreKey 字段；一个 Foods 表，它包括 FoodName 和 FoodKey 字段；和一个 Prices 表，它包括一个 PriceKey、一个 Price 以及对 StoreKey 和 FoodKey 的引用。

在我们的 Facade 例子里，我们为三个表分别创建了自己的类，让类负责创建具体的表。由于这三张表很相似，所以让它们都派生于基类 DBTable。

18.8.1　构建价格查询

对于每种食品，我们都想得到一个关于哪个商店售价最低的报告。这意味着要对数据库编写一个简单的 SQL 查询过程。可以在 Price 类中做这项工作，并让它返回一个带有商店名和价格的 DataSet。

最后的应用程序为：一个列表框中给出食品名称，单击其中一种食品，在另一个列表框中给出价格，如图 18.1 所示。

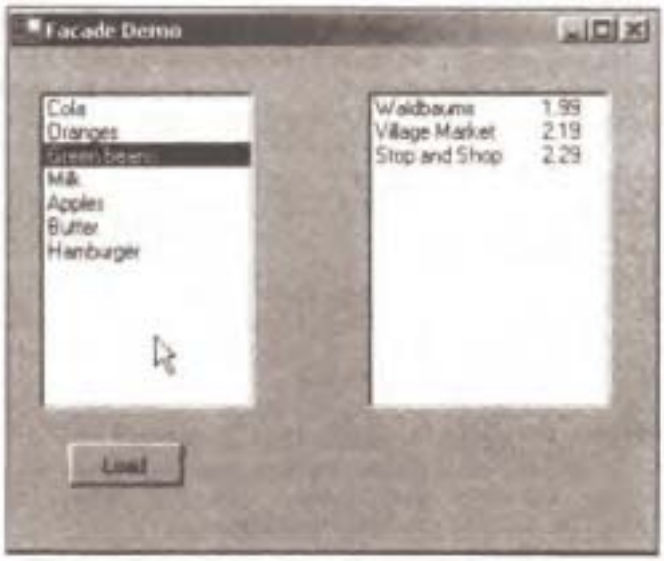


图 18.1　使用外观模式的显示食品价格的程序

18.9　创建 ADO.NET 的 Facade

在 Facade 例子里，我们要创建自己的食品数据库，先创建一个抽象类 DBase，用它表示数据库的一个连接。该类封装了创建连接、打开表和 SQL 查询等操作。

```
public abstract class DBase{
      protected OleDbConnection conn;

private void openConnection() {
      if (conn.State == ConnectionState.Closed){
            conn.Open ();
      }
}
//------
private void closeConnection() {
      if (conn.State == ConnectionState.Open ){
            conn.Close ();
      }
}
//------
public DataTable openTable (string tableName) {
      OleDbDataAdapter adapter = new OleDbDataAdapter ();
      DataTable dtable = null;
      string query = "Select * from " + tableName;
      adapter.SelectCommand = new OleDbCommand (query, conn);
      DataSet dset = new DataSet ("mydata");
      try {
            openConnection();
            adapter.Fill (dset);
            dtable = dset.Tables [ 0];
      }
      catch(Exception e) {
            Console.WriteLine (e.Message );
      }
      return dtable;
}
//------
public DataTable openQuery(string query) {
      OleDbDataAdapter dsCmd = new OleDbDataAdapter ();
      DataSet dset = new DataSet ();  //create a dataset
      DataTable dtable = null;        //declare a data table
      try {
            //create the command
            dsCmd.SelectCommand = new OleDbCommand(query, conn);
            openConnection();             //open the connection
            //fill the dataset
            dsCmd.Fill(dset, "mine");
            //get the table
            dtable = dset.Tables[ 0];
            closeConnection();            //always close it
            return dtable;                //and return it
      }
      catch (Exception e) {
            Console.WriteLine (e.Message);
            return null;
      }
}
//------
public void openConnection(string connectionString) {
      conn = new OleDbConnection(connectionString);
}
//------
public OleDbConnection getConnection() {
```

```
        return conn;
    }
    }
```

注意，除了没有构造函数外，它是一个完整的类。接下来创建多个派生类，每个派生类为不同的数据库创建连接字符串。下面是针对Access数据库生成的一个派生类。

```
public class AxsDatabase :DBase   {
      public AxsDatabase(string dbName)          {
            string connectionString =
            "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" +
                  dbName;
            openConnection(connectionString);
      }
 }
```

下面是针对SQL Server的一个派生类。

```
public class SQLServerDatabase:DBase  {
      string connectionString;
      //-----
public SQLServerDatabase(String dbName)          {
      connectionString = "Persist Security Info = False;" +
            "Initial Catalog =" + dbName + ";" +
            "Data Source = myDataServer;User ID = myName;" +
            "password=";
      openConnection(connectionString);
}
//-----
public SQLServerDatabase(string dbName, string serverName,
      string userid, string pwd) {
      connectionString = "Persist Security Info = False;" +
            "Initial Catalog =" + dbName + ";" +
            "Data Source =" + serverName + ";" +
            "User ID =" + userid + ";" +
            "password=" + pwd;
      openConnection(connectionString);
      }
}
```

18.9.1 DBTable类

我们需要的另外一个重要类是DBTable。它封装了打开数据库表、填充数据库表及更新数据库表等操作。在本例中，我们还要用该类添加数据，可以从它派生出食品类和商店类，让它们为各自的类完成这种额外的工作。

```
public class DBTable  {
      protected DBase db;
      protected string tableName;
      private bool filled, opened;
      private DataTable dtable;
      private int rowIndex;
      private Hashtable names;
      private string columnName;
      private DataRow row;
      private OleDbConnection conn;
      private int index;
 //-----
```

```
public DBTable(DBase datab, string tb_Name)      {
      db = datab;
      tableName = tb_Name;
      filled =false;
      opened = false;
      names = new Hashtable();
}
//-----
public void createTable() {
      try {
            dtable = new DataTable(tableName);
            dtable.Clear();
      }
      catch (Exception e) {
            Console.WriteLine (e.Message );
      }
}
//-----
public bool hasMoreElements() {
      if(opened)
            return (rowIndex < dtable.Rows.Count);
      else
            return false;
}
//-----
public int getKey(string nm, string keyname){
      DataRow row;
      int key;
  if(! filled)
            return (int)names[ nm];
  else {
      string query = "select * from " + tableName + " where " +
             columnName + "=\'" + nm + "\'";
      dtable = db.openQuery(query);
      row = dtable.Rows[ 0];
      key = Convert.ToInt32 (row[ keyname] .ToString());
      return key;
  }
}
//-----
public virtual void makeTable(string cName) {
      //show below
//-----
private void closeConn() {
  if( conn.State == ConnectionState.Open) {
            conn.Close();
  }
}
//-----
private void openConn() {
      if(conn.State == ConnectionState.Closed ) {
            conn.Open();
      }
}
//-----
public void openTable() {
      dtable = db.openTable(tableName);
      rowIndex = 0;
```

```
        if(dtable != null)
                opened = true;
    }
    //-----
    public void delete() {
          //show above
    }
}
```

18.10 为每张表创建自己的类

我们从DBTable类派生出Stores类、Foods类和Prices类，并重用该类的大部分代码，在分析输入文件时，类Stores和Foods都要求创建一个名字惟一的表：Stores类中的商店名和Foods类中的食品名都应该是惟一的。

C#提供了一种非常方便的方法来创建这些类：使用Hashtable（哈希表）。Hashtable是一个无边界的数组，其中的每个元素用一个惟一的关键字来表示。Hashtable的一种使用方式是：用一个较短的代号作为关键字向表中添加对象，然后可以通过代号从表中取出对象。对象不必是惟一的，但关键字必须惟一。

Hashtable的另外一个方便之处是生成一个具有惟一名字的表。如果名字是关键字，另外一些数据是表的内容，我们可以将名字添加到Hashtable中，并保证每个关键字是惟一的。由于关键字惟一，Hashtable必须以一种可预知的方式处理试图向表中添加一个重复关键字的情况。例如，Java中的Hashtable是用新的数据项代替原来的、与该关键字对应的数据项。C#中的Hashtable采用另一种方式：当试图添加一个关键字不惟一的数据时抛出一个异常。

现在，考虑到在将名字加入数据库之前，要先在整个列表中收集名字，可以用下面的方法将名字加到Hashtable，并保证它们的惟一性。

```
public void addTableValue(string nm) {
//accumulates names in hash table
      try {
              names.Add(nm, index++);
      }
      catch (ArgumentException) {}
      //do not allow duplicate names to be added
}
```

在所有名字添加完后，就将其加到数据库表中。这里使用了Hashtable的Enumerator属性，用它在输入到列表里的名字之间循环。

```
public virtual void makeTable(string cName) {
      columnName = cName;
      //stores current hash table values in data table
      DataSet dset = new DataSet(tableName);   //create the data set
      dtable = new DataTable(tableName);   //and a datatable
      dset.Tables.Add(dtable);                 //add to collection
      conn = db.getConnection();
      openConn();                              //open the connection
      OleDbDataAdapter adcmd = new OleDbDataAdapter();
      //open the table
      adcmd.SelectCommand =
            new OleDbCommand("Select * from " + tableName, conn);
      OleDbCommandBuilder olecb = new OleDbCommandBuilder(adcmd);
      adcmd.TableMappings.Add("Table", tableName);
```

```
        //load current data into the local table copy
        adcmd.Fill(dset, tableName);
        //get the Enumerator from the Hashtable
        IEnumerator ienum = names.Keys.GetEnumerator();
        //move through the table, adding the names to new rows
        while (ienum.MoveNext()) {
               string name = (string)ienum.Current;
               row = dtable.NewRow();      //get new rows
               row[ columnName] = name;
               dtable.Rows.Add(row);     //add into table
        }
        //Now update the database with this table
        try {
               adcmd.Update(dset);
               closeConn();
               filled = true;
        }
        catch (Exception e) {
               Console.WriteLine (e.Message);
        }
}
```

利用上述方法，可将派生类Stores简化成如下形式：

```
public class Stores :DBTable     {
       public Stores(DBase db):base(db, "Stores"){
       }
     //-----
      public void makeTable() {
            base.makeTable ("Storename");
       }
 }
```

同样可以将Foods简化成下列形式：

```
public class Foods: DBTable        {
       public Foods(DBase db):base(db, "Foods"){
       }
       //-----
       public void makeTable() {
            base.makeTable ("Foodname");
       }
       //-----
       public string getValue() {
            return base.getValue ("FoodName");
       }
 }
```

getValue方法可以列举Stores的名字表或Foods的名字表，我们将该方法放在基类DBTable中。

```
public virtual string getValue(string cname) {
  //returns the next name in the table
  //assumes that openTable has already been called
       if (opened) {
              DataRow row = dtable.Rows[ rowIndex++] ;
              return row[ cname] .ToString().Trim ();
       }
       else
              return "";
}
```

注意，将该方法声明为 virtual，这样在需要的地方可以覆盖它。

18.11 构建 Prices 表

Prices 表稍微有些复杂，因为它包含了另外两个表的关键字。该表完成时类似于表 18.5。

表 18.5 Grocery 数据库中的 Prices 表

Pricekey	Foodkey	StoreKey	Price
1	1	1	0.27
2	2	1	0.36
3	3	1	1.98
4	4	1	2.39
5	5	1	1.98
6	6	1	2.65
7	7	1	2.29
8	1	2	0.29
9	2	2	0.29
10	3	2	2.45
11	4	2	2.99
12	5	2	1.79
13	6	2	3.79
14	7	2	2.19
15	1	3	0.33
16	2	3	0.47
17	3	3	2.29
18	4	3	3.29
19	5	3	1.89
20	6	3	2.99
21	7	3	1.99

为了创建Prices表，我们需要重新读取文件，找到商店名和食品名，查找它们的关键字，将其添加到 Prices 表中。DBTable 接口不包含这个方法，但我们可以向 Prices 类添加额外的特殊方法，这些方法不作为接口的一部分。

Prices类将一系列 StoreFoodPrice 对象存在一个 ArrayList 里，然后一次将它们全部装入数据库中。注意，我们已经重新调用了 DBTable 的派生类，为商店关键字、食品关键字和价格取得参数。

每次向 Prices 类内部的 ArrayList 添加一个商店关键字、食品关键字和价格时，就创建并保存了一个 StoreFoodPrice 对象实例。

```
public class StoreFoodPrice {
      private int storeKey, foodKey;
      private float foodPrice;
      //-----
      public StoreFoodPrice(int sKey, int fKey, float fPrice) {
            storeKey = sKey;
            foodKey = fKey;
            foodPrice = fPrice;
      }
      //-----
      public int getStore() {
            return storeKey;
      }
```

```
        //-----
        public int getFood() {
             return foodKey;
        }
        //-----
        public float getPrice() {
             return foodPrice;
        }
 }
```

所有的内容都得到后，就可以创建具体的数据库表。

```
public class Prices : DBTable    {
        private ArrayList priceList;
        public Prices(DBase db) : base(db, "Prices")    {
             priceList = new ArrayList ();
        }
        //-----
        public void makeTable() {
        //stores current array list values in data table
          OleDbConnection adc = new OleDbConnection();

          DataSet dset = new DataSet(tableName);
          DataTable dtable = new DataTable(tableName);

          dset.Tables.Add(dtable);
          adc = db.getConnection();
          if (adc.State == ConnectionState.Closed)
              adc.Open();
          OleDbDataAdapter adcmd = new OleDbDataAdapter();

          //fill in price table
          adcmd.SelectCommand =
              new OleDbCommand("Select * from " + tableName, adc);
          OleDbCommandBuilder custCB = new
              OleDbCommandBuilder(adcmd);
          adcmd.TableMappings.Add("Table", tableName);
          adcmd.Fill(dset, tableName);
          IEnumerator ienum = priceList.GetEnumerator();
          //add new price entries
          while (ienum.MoveNext() ) {
                   StoreFoodPrice fprice =
                          (StoreFoodPrice)ienum.Current;
                   DataRow row = dtable.NewRow();
                   row[ "foodkey"] = fprice.getFood();
                   row[ "storekey"] = fprice.getStore();
                   row[ "price"] = fprice.getPrice();
                   dtable.Rows.Add(row);     //add to table
          }
          adcmd.Update(dset);       //send back to database
          adc.Close();
          }
        //-----
        public DataTable getPrices(string food) {
        string query=
          "SELECT Stores.StoreName, " +
          "Foods.Foodname, Prices.Price " +
          "FROM (Prices INNER JOIN Foods ON " +
```

```
            "Prices.Foodkey = Foods.Foodkey) " +
            "INNER JOIN Stores ON" +
            "Prices.StoreKey = Stores.StoreKey " +
            "WHERE(((Foods.Foodname) = \'" + food + "\'))" +
            "ORDER BY Prices.Price";
            return db.openQuery(query);
        }
        //-----
        public void addRow(int storeKey, int foodKey, float price) {
             priceList.Add (
                    new StoreFoodPrice (storeKey,
                           foodKey, price));
        }
}
```

18.12 填写数据库表

利用这些派生类，可以编写一个类从数据文件读数据填到表中。该类一旦要读文件，就先构建Stores和Foods数据库表；然后反复读取文件，查找商店关键字和食品关键字，并将它们加到Prices类的数组表中；最后创建Prices表。

```
public class DataLoader     {
       private csFile vfile;
       private Stores store;
       private Foods fods;
       private Prices price;
       private DBase db;
       //-----
       public DataLoader(DBase datab) {
            db = datab;
            store = new Stores(db);
            fods = new Foods (db);
            price = new Prices(db);
       }
       //-----
       public void load(string dataFile) {
              string sline;
              int storekey, foodkey;
              StringTokenizer tok;
              //delete current table contents
              store.delete();
              fods.delete();
              price.delete();
              //now read in new ones
              vfile = new csFile(dataFile);
              vfile.OpenForRead();
              sline = vfile.readLine();
              while (sline != null){
                   tok = new StringTokenizer(sline, ",");
                   store.addTableValue(tok.nextToken());    //store
                   fods.addTableValue(tok.nextToken());     //food
                   sline = vfile.readLine();
              }
              vfile.close();
              //construct store and food tables
              store.makeTable();
              fods.makeTable();
```

```
            vfile.OpenForRead();
            sline = vfile.readLine();
        while (sline != null) {
            //get the gets and add to storefoodprice objects
            tok = new StringTokenizer(sline, ",");
            storekey = store.getKey(tok.nextToken(), "Storekey");
            foodkey = fods.getKey(tok.nextToken(), "Foodkey");
            price.addRow(storekey, foodkey,
                  Convert.ToSingle (tok.nextToken()));
                  sline = vfile.readLine();
        }
         //add all to price table
         price.makeTable();
         vfile.close();
        }
}
```

18.13 最终的应用程序

程序一开始运行，就将一个食品价格表填到一个列表框中。

```
private void loadFoodTable() {
      Foods fods =new Foods(db);
      fods.openTable();
      while (fods.hasMoreElements()){
            lsFoods.Items.Add(fods.getValue());
      }
}
```

单击某个食品名，显示选中食品的价格。

```
private void lsFoods_SelectedIndexChanged(object sender,
            System.EventArgs e) {
      string food  = lsFoods.Text;
      DataTable dtable = prc.getPrices(food);

      lsPrices.Items.Clear();
      foreach (DataRow rw in dtable.Rows) {
            lsPrices.Items.Add(rw["StoreName"].ToString().Trim() +
              "\t" + rw["Price"].ToString());
      }
}
```

最终的程序运行结果如图 18.1 所示。

18.14 Facade 的构成

本例中的Facade包装了下列的类：

- DBase

  包含了OleDbConnection，Database，DataTable，OleDbAdapter，DataSet。

- DBTable

  包含了DataSet，DataRow，Datatable，OleDbConnection，OleDbCommandBuilder。

处理如此复杂的数据对象时，读者就能很快明白Facade方法的优点。这一点可以用图18.2说明。

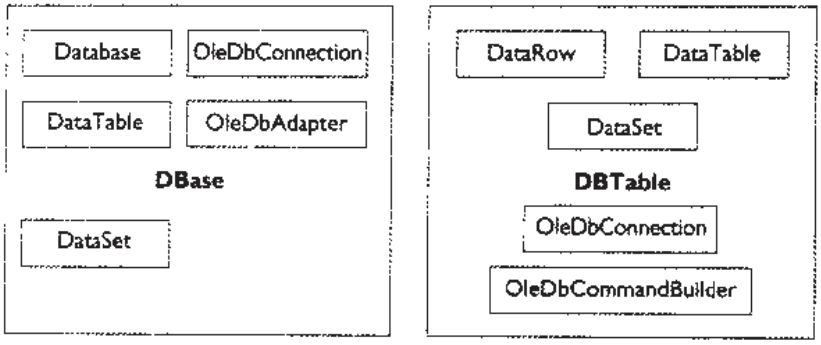


图 18.2　Facade 将类包装，组成了 DBase 类和 DBTable 类

18.15　Facade 的效果

外观模式对客户屏蔽了复杂的子系统组件，并为一般用户提供了一个比较简单的程序设计接口。但是，它并没有限制高级用户在需要时使用深层的、较复杂的类。

另外，Facade 允许读者改变底层的子系统而不需要修改客户代码，降低了编译依赖性。

18.16　思考题

假设读者编写了一个程序，带有一个 FilelOpen 菜单、一个文本域及一些控制字体（黑体和斜体）的按钮。现在假定读者需要用带参数的命令行方式运行该程序，如何使用外观模式解决这一问题。

18.17　随书附带光盘中的程序

C# Database Facade Classes	\Facade\

第19章　享元模式

享元模式（Flyweight Pattern）可以避免大量非常相似的类的开销。在程序设计中会出现这样的情况：看起来似乎需要生成大量细粒度的类实例来表示数据。如果能发现这些实例除了几个参数外基本上都是相同的，有时就能够大幅度地减少需要实例化的类的数量。如果能把那些参数移到类实例外面，在方法调用时将它们传递进来，就可以通过共享大幅度地减少单个实例的数目。

享元设计模式提供了处理这种类的一种方法，它涉及到使实例惟一的内部数据和作为参数传递的外部数据。享元模式适合于小的、细粒度的类（像屏幕上的单个字符或图标）。例如，在一个窗口的屏幕上画一系列的图标，每个图标把一个人或数据文件表示成一个文件夹，如图19.1所示。

这种情况下，若每个文件夹用一个单独的类实例表示，让它记住该人的名字和图标的屏幕位置，是没有意义的。这些图标一般来说都是非常相似的图片，任何情况下它们的绘图位置都可以根据窗口的尺寸自动计算出来。

在“Design Patterns”所给出的另外一个例子里，将文档中的每种字符表示成一个字符类的单个实例，而将屏幕上的字符绘制位置保存成外部数据。在这个例子中，每种字符有且只有一个实例，而不是该字符每出现一次就有一个实例。

图19.1　表示不同人的信息的一系列文件夹。由于它们很相似，适合于使用享元模式

19.1　讨论

Flyweight是一个类的可共享实例。初看起来每个类都是单件的（Singleton），但实际上它可以有少数几个实例，例如每个字符类有一个实例，每个图标类型有一个实例。指定的实例数目必须根据需要的类实例来决定，通常这项工作由享元工厂（FlyweightFactory）来完成，该Factory类通常是单件的，因为它需要记住是否生成了一个特定的实例，然后返回一个新实例或一个指向已生成实例的引用。

为了确定某部分程序是否适合于使用Flyweight，要考虑能否从类中移出某些数据并存储到外部，如果这能大幅度地减少程序需要保存的不同类实例的数量，就适合于使用Flyweight。

19.2 示例代码

假设我们要画一个小文件夹图标，并在图标下面给出机构中每个人的名字，如果该机构很庞大，就会有大量的这类图标，但实际上它们都是相同的图片。即使我们需要两种类型的图标：一种是选中的，一种是未选中的，不同图标的数量也相当少。在这样一个系统中，若为每个人生成一个带有坐标位置、名字和选中状态的图标对象，是非常浪费资源的。我们在图19.2中给出两种这样的图标。

图 19.2 带一个选中文件夹的 Flyweight 显示

我们换一种做法，创建一个FolderFactory类，它返回选中文件夹的绘制类实例或未选中文件夹的绘制类实例，但是，若已经创建一个实例，就不能再创建另外的实例。由于这只是一个简单的例子，所以我们一开始就创建两个 Folder 实例，并返回其中的一个。

```
public class FolderFactory {
      private Folder selFolder, unselFolder;
      //-----
      public FolderFactory()          {
            //create the two folders
            selFolder = new Folder(Color.Brown);
            unselFolder = new Folder(Color.Bisque);
      }
      //-----
      public Folder getFolder(bool selected) {
            if(selected)
                  return selFolder;
            else
                  return unselFolder;
      }
}
```

对于存在多个实例的情况，Factory 可以保存一张已创建实例的表，只有表中不存在的实例才可以被创建。

使用 Flyweight 惟一需要做的事情，就是在绘制图标时把绘制坐标和名字传递到文件夹中。这些坐标都是外部数据，这才能使我们共享文件夹对象，在本例中只创建两个实例就可以了。下面给出的是完整的文件夹类，它使用一种背景颜色或另一种背景颜色创建一个文件夹实例，并有一个公有的 draw 方法，能在指定的位置绘制文件夹。

```
public class Folder   {
       //Draws a folder at the specified coordinates
       private const int w  = 50;
       private const int h = 30;
       private Pen blackPen, whitePen;
       private Pen grayPen;

       private SolidBrush backBrush, blackBrush;
       private Font fnt;
       //------
       public Folder(Color col)        {
            backBrush = new SolidBrush(col);
            blackBrush = new SolidBrush(Color.Black);
            blackPen = new Pen(Color.Black);
            whitePen = new Pen(Color.White);
            grayPen = new Pen(Color.Gray);
            fnt = new Font("Arial", 12);
       }
       //------
       public void draw(Graphics g, int x, int y, string title) {
              //color folder
              g.FillRectangle(backBrush, x, y, w, h);
              //outline in black
              g.DrawRectangle(blackPen, x, y, w, h);
              //left 2 sides have white line
              g.DrawLine(whitePen, x + 1, y + 1, x + w - 1, y + 1);
              g.DrawLine(whitePen, x + 1, y, x + 1, y + h);
              //draw tab
              g.DrawRectangle(blackPen, x + 5, y - 5, 15, 5);
              g.FillRectangle(backBrush, x + 6, y - 4, 13, 6);
              //gray line on right and bottom
              g.DrawLine(grayPen, x, y + h - 1, x + w, y + h - 1);
              g.DrawLine(grayPen, x + w - 1, y, x + w - 1,
               y + h - 1);
              g.DrawString(title, fnt, blackBrush, x, y + h + 5);
       }
}
```

为了使用这样一个Flyweight类，作为paint例程的一部分，主程序必须计算出每个文件夹的位置，然后将坐标位置传递给文件夹实例。这实际上很容易理解，因为读者需要根据窗口的尺寸进行不同的布局，而且不想去通知每个实例它的新位置在哪儿。我们采用的方法是在paint例程中自动计算出坐标位置。

我们注意到，程序一开始就已生成了一个文件夹ArrayList，通过扫描数组就能绘制每个文件夹。这种数组不像一系列不同的实例那样浪费空间，因为它实际上是对两个文件夹实例中的一个实例的引用数组。由于我们想将一个文件夹显示为“选中的”，而且希望文件夹被选中时能自动改变显示，每次选择时可以用FolderFactory给出正确的实例。

在显示例程里，有两个地方需要计算文件夹的位置：一个是在绘制文件夹的时候，一个是在检查鼠标移到文件夹上的时候。这样，将计算位置的代码抽取出来放在Positioner类中比较恰当。

```
public class Positioner    {
       private const int pLeft = 30;
       private const int pTop  = 30;
       private const int HSpace = 70;
       private const int VSpace = 80;
       private const int rowMax = 2;
       private int x, y, cnt;
       //-----
       public Positioner()  {
              reset();
       }
```

```
        //-----
        public void reset() {
                x = pLeft;
                y = pTop;
                cnt = 0;
        }
        //-----
        public int nextX() {
                return x;
        }
        //-----
        public void incr() {
                cnt++;
                if (cnt > rowMax) {      //reset to start new row
                        cnt = 0;
                        x = pLeft;
                        y += VSpace;
                }
                else {
                        x += HSpace;
                }
        }
        //-----
        public int nextY() {
                return y;
        }
 }
```

接下来就能编写出一个比较简单的 paint 例程。

```
private void picPaint(object sender,  PaintEventArgs e ) {
        Graphics g = e.Graphics;
        posn.reset ();
        for(int i = 0; i < names.Count; i++) {
              fol = folFact.getFolder(selectedName.Equals(
                    (string)names[ i] ) );
              fol.draw(g, posn.nextX(), posn.nextY (),
                    (string)names[ i] );
              posn.incr();
        }
}
```

19.2.1 类图

图 19.3 中的类图给出了这些类之间的相互关系。

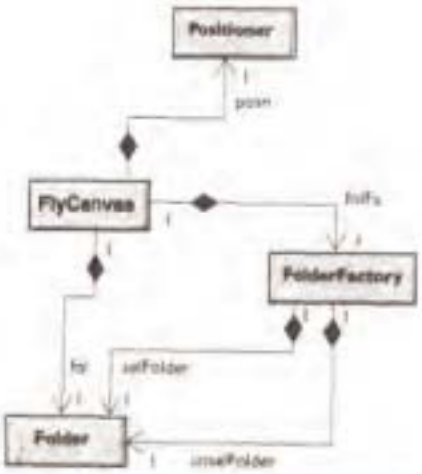


图 19.3 显示了 Flyweight 是如何生成的

FlyCanvas类是主要的UI（用户界面）类，文件夹的排列和绘制都在该类中，它包含了一个FolderFactory实例和一个Folder实例。FolderFactory类包含两个Folder实例：选中的实例和未选中的实例，FolderFactory将其中的一个实例返回给FlyCanvas类。

19.2.2 选择一个文件夹

我们有两个文件夹实例：选中的实例和未选中的实例，我们希望当鼠标移到文件夹上时能够显示出选中了该文件夹。在前面的paint例程里，我们只是记住了被选中的文件夹的名字，并请求工厂为该名字返回一个“选中的”文件夹。由于文件夹不是单独的实例，所以不能在每个文件夹实例中监听鼠标的动作。事实上，即使能在文件夹中监听，还是需要一种方式告诉其他的实例它自己未被选中。

换一种方式，在图片框层次（Picturebox Level）检查鼠标动作，如果在一个矩形范围内发现了鼠标，就将相应的名字作为选中的名字。创建一个Rectangle类的实例，在Rectangle类中检查一个文件夹是否在某时刻包含鼠标。注意，我们将该类放在csPatterns名字空间，这样可以保证不会同System.Drawing名字空间里的Rectangle类相冲突。

```
namespace csPatterns {
       public class Rectangle    {
       private int x1, x2, y1, y2;
       private int w, h;
       public Rectangle()        {               }
       //-----
       public void init(int x, int y) {
              x1 = x;
              y1 = y;
              x2 = x1 + w;
              y2 = y1 + h;
       }
       //-----
       public void setSize(int w_, int h_) {
              w = w_;
              h = h_;
       }
       //-----
       public bool contains(int xp, int yp) {
              return (x1 <= xp) && (xp <= x2) &&
                        (y1 <= yp) && (yp <= y2);
       }
  }
}
```

这样，当我们在需要的地方重画并创建一个“选中的”文件夹实例时，只检查每个文件夹的名字就可以了。

```
private void Pic_MouseMove(object sender, MouseEventArgs e) {
      string oldname = selectedName;  //save old name
      bool found = false;
      posn.reset ();
      int i = 0;
      selectedName = "";
      while (i < names.Count && ! found) {
             rect.init (posn.nextX(), posn.nextY ());
             //see if a rectangle contains the mouse
```

```
                if (rect.contains(e.X, e.Y) ){
                        selectedName = (string)names[ i];
                        found = true;
                }
                posn.incr ();
                i++;
        }
        //only refresh if mouse in new rectangle
        if(  !oldname.Equals ( selectedName)) {
                Pic.Refresh();
        }
}
```

19.3 处理鼠标事件和绘图事件

在C#中，通过添加事件处理句柄来截取绘图事件和鼠标事件。为了能绘制文件夹，我们向图片框添加一个绘图事件处理句柄。

```
Pic.Paint += new PaintEventHandler (picPaint);
```

上面添加的picPaint句柄完成绘制文件夹的工作。可以手工添加该代码，因为我们知道该绘图例程的名字。

```
private void picPaint (object sender, PaintEventArgs e )  {
```

尽管鼠标移动的事件处理程序与此非常类似，我们也可能记不住它的准确格式。因此，用Visual Studio IDE自动生成。在设计模式里显示一个窗体，单击PictureBox，在Properties窗口中单击闪电形按钮，会显示出PictureBox所有的事件，如图19.4所示。

然后双击MouseMove，系统会为鼠标移动事件生成正确的代码，并自动添加事件处理句柄。生成的空方法如下所示：

```
private void Pic_MouseMove ( object sender, MouseEventArgs e )   {
}
```

添加事件处理句柄的代码在Windows Form Designer生成的部分里，如下所示：

```
Pic.MouseMove += new MouseEventHandler ( Pic_MouseMove );
```

图 19.4 在Properties窗口中选择MouseMove事件

19.4 C# 中 Flyweight 的应用

在 C# 的应用层次上，不经常使用 Flyweight。Flyweight 是一种系统资源管理技术，用在比 C# 低的层次上。然而，在Internet程序设计中创建了大量的无状态对象，这种情况有些类似于Flyweight。能认识到这种技术的存在，并在需要的时候使用它，通常还是有帮助的。

C# 语言里的某些对象的实现就隐藏着 Flyweight。例如，两个含相同字符的 String 常量实例，它们指向的可能就是同一存储位置。同样地，数值相同的两个整型常量或浮点型常量也可能是按 Flyweight 实现的，尽管实际上未必是这样。

19.5 共享对象

"Smalltalk Companion" 指出，尽管它们的意图有些不同，但共享对象非常类似于 Flyweight。如果有一个非常大的对象，它包含了大量的复杂数据，比如表或位图，读者会愿意减少该对象的实例数目。这种情况下，应当把一个实例返回给提出请求的各部分程序，并避免创建其他的实例。

当某部分程序想要改变共享对象中的数据时会发生问题。读者在此时必须做出决定：是对所有的用户都改变该对象，还是防止它做任何修改，亦或是创建一个带有改变后数据的新实例。如果是为每个实例而更改对象，必须通知它们对象已经发生改变。

引用 C# 外面的大型数据系统时，比如引用数据库，共享对象也是很有用的。前面在外观模式中开发的DBase类也可以作为一个共享对象，并且不希望在程序的各模块里有太多独立的数据库连接，最好只实例化一个连接。即使这样，不同线程中的几个模块仍可以同时进行查询，Database 类需要将这些查询排队或生成另外的连接。

19.6 Copy-on-Write 对象

享元模式只用几个对象实例表示程序中的多个不同对象，这些实例通常把相同的基本属性作为内部数据；把少数几个属性作为外部数据，它们通常随类实例表现形式不同而不同。然而，有些实例偶尔会接受新的内部属性（如图形或文件夹的 tab 位），需要一个特定的新实例来表示这些属性。不是像创建子类一样事先创建这些实例，而是在程序流程指出需要一个单独的新实例时，拷贝类实例并修改它的内部属性。我们把这个过程称为 "拷贝 - 修改（Copy-on-Write）"，这个过程可用在 Flyweight 和其他许多类中，如我们后面要讨论的代理（Proxy）中。

19.7 思考题

如果按钮可以在 TabDialog 的几个 tab 位上出现，每个按钮控制一到两个相同的任务，那么这种情况适合于使用 Flyweight 吗？

19.8 随书附带光盘中的程序

C# Folders	\Flyweight

第20章 代理模式

当读者需要将一个复杂的对象或创建时比较花费时间的对象表示成一个简单对象时，可以使用代理模式（Proxy Pattern）。如果创建一个对象比较浪费时间或浪费计算机资源，Proxy允许将创建过程推迟到需要该实际对象的时候。Proxy对象通常具有和它所代表的对象一样的方法，一旦对象被调人，就把调用方法从Proxy传递给实际对象。

下列几种情况适合于使用Proxy：

1. 调人一个对象需要花费很长时间，如调人一个大图像。
2. 计算结果需要花费很长时间才能完成，计算过程中需要显示中间结果。
3. 对象位于远程机器上，通过网络调人会很慢，特别是在网络负载高峰期间。
4. 对象限制了访问权限，代理可以使访问许可对用户有效。

Proxy还可以区别是请求一个对象实例，还是真正需要访问该实例。例如，程序初始化时建立了许多对象，但不是所有的对象马上就被使用，这种情况下，代理能在需要时再调人真正的对象。

我们考虑这样一个例子，一个程序需要调人一个大图像并显示出来。程序开始运行时，必须对要显示的图像有些说明，这样才能正确布局屏幕，但真正的图像显示可以推迟到图像完全调人之后。这一点在诸如字处理、Web浏览器等程序中是相当重要的，这类程序在图像可用之前，要在图像周围安排文本。

在图像出现前，图像代理在屏幕上画一个简单的矩形或其他符号来表示该图像的尺寸，同时标明图像并在后台开始调人过程。代理甚至能将图像的调人推迟到它接收一个绘制请求时，只有在那时才开始调人过程。

20.1 示例代码

本例中，我们创建一个简单的程序，调人一个图像时，在一个图像控件上显示另一幅图像。不是直接调人该图像，而是使用一个ImageProxy类推迟调人过程并画出一个矩形，一直到调人完成。

```
private void init() {
        imgProxy = new ImageProxy ();
}
//-----
public Form1() {
        InitializeComponent();
        init();
}
//-----
private void button1_Click(object sender, System.EventArgs e) {
        Pic.Image = imgProxy.getImage ();
}
```

注意，在我们需要一个图像代理时，创建了一个ImageProxy实例。ImageProxy类准备图像的调人，并创建了一个Imager对象跟踪调人过程，它返回一个实现Imager接口的类。

```
public interface Imager   {
     Image getImage ();
}
```

在这个简单的例子里，ImageProxy类只延迟了5秒钟，然后从开始的图像切换到最后的图像，它使用一个Timer类的实例完成上述工作。TimerCallback类中定义了定时器到时间时要调用的方法，我们使用该类处理定时器。这与我们添加其他事件处理句柄的方式非常类似，方法timerCall设置done标志并关闭时钟。

```
public class ImageProxy     {
      private bool done;
      private Timer timer;
      //-----
      public ImageProxy()        {
      //create a timer thread and start it
            timer = new Timer (
                  new TimerCallback (timerCall), this, 5000, 0);
      }
      //-----
      //called when timer completes
      private void timerCall(object obj) {
            done = true;
            timer.Dispose ();
      }
      //-----
      public Image getImage() {
            Imager img;
            if (done)
                  img = new FinalImage ();
            else
                  img = new QuickImage ();
            return img.getImage ();
      }
}
```

我们在两个小类里实现Imager接口，这两个类分别为QuickImage和FinalImage。一个类获得一个小的gif图像，另一个类获得一个较大的（可能较慢）jpeg图像。C#中的Image类是一个抽象类，Bitmap类、Cursor类、Icon类和Metafile类都是由它派生出来的，所以，我们要返回的实际类就派生于Image。QuickImage类由一个gif文件返回一个Bitmap类，FinalImage由一个jpeg文件返回一个Bitmap类。

```
public class QuickImage : Imager {
      public QuickImage()   {}
      public Image getImage() {
            return new Bitmap ("Box.gif");
      }
}
//------------
public class FinalImage :Imager {
      public FinalImage()   {}
      public Image getImage() {
            return new Bitmap("flowrtree.jpg");
      }
}
```

取一幅图像时，首先得到的是快速图像，5秒钟后，若再次调用该方法，会得到最终的图像。图 20.1 说明了程序的这两种状态。

图 20.1　左图显示代理图像直到图像调入完毕，右图显示的是实际图像

20.2　C# 中的 Proxy

C#语言比其他语言有更多的代理行为，因为C#就是为网络和互联网应用开发的。例如，ADO.NET 中的数据库连接类实际上全部都是代理。

20.3　Copy-on-Write

也可以使用代理来保存较大对象的拷贝，该对象可以改变也可以不变。如果读者为一个开销很大的对象创建第二个实例，Proxy能判断出没有理由再生成一个拷贝，只需使用原始对象就可以了。如果程序在一个新拷贝里做了修改，Proxy能拷贝原始对象并在新实例中做出相应的修改。当对象实例化后就不再改变时，这种做法可以节省大量的时间和空间。

20.4　相关模式之间的比较

Adapter 和 Proxy 都是在对象外围构建了一个薄层，但是，Adapter 是为对象提供一个不同的接口，而 Proxy 为对象提供的是相同的接口，该接口可以推迟处理过程或数据转换工作。

Decorator 也具有与它所包含的对象相同的接口，但它的目标是为原对象添加额外的（有时是可视化的）功能，而 Proxy 正相反，它控制对所包含的类的访问。

20.5　思考题

设计一个连接到数据库的服务器，如果有几个客户终端要同时连接到服务器上，如何使用Proxy?

20.6　随书附带光盘中的程序

Image Proxy	`\Proxy`

20.7 结构型模式小结

第三部分包含了下列结构型模式：

适配器模式（Adapter Pattern）用于将一个类的接口转换成另一个类的接口。

桥接模式（Bridge Pattern）可以将一个类的接口与它的实现分离，这样可以不用修改客户端代码就能改变或替换实现过程。

组合模式（Composite Pattern）是一个对象的集合，其中的任何一个对象既可以是一个组合，也可以只是一个叶子对象。

装饰模式（Decorator Pattern）用一个类包装给定的类，并为它添加新的功能，将所有未改动的方法传递给下面的类。

外观模式（Facade Pattern）将一系列复杂的对象放在一起，并提供一个新的、更简单的访问这些数据的接口。

享元模式（Flyweight Pattern）把一部分的类数据移到类外部，在执行方法时将数据传递进来，通过这种方式限制那些又小又相似的实例的增加数量。

代理模式（Proxy Pattern）为一个比较复杂的对象提供一个简单的占位（placeholder）对象，实例化该复杂对象，在某种程度上比较浪费时间或代价较高。

第四部分

行为型模式

行为型模式（Behavioral Pattern）通常和对象之间的通信有关，在本部分，我们研究下列行为模式。

职责链（Chain of Responsibility）把请求从链中的一个对象传递到下一个对象，直到请求被响应为止。通过这种方式在对象之间去除耦合。

命令模式（Command Pattern）用简单的对象表示软件命令的执行，支持登录和取消操作。

解释器模式（Interpreter Pattern）提供一个如何把语言元素包含在程序中的定义。

迭代器模式（Iterator Pattern）提供了一种顺序访问一个类中的一系列数据的方式。

中介者模式（Mediator Pattern）定义了如何用一个对象简化对象之间的通信，使对象之间不必相互了解。

备忘录模式（Memento Pattern）定义了如何保存一个类实例的内容以便以后能恢复它。

观察者模式（Observer Pattern）定义了一种把改动通知给多个对象的方式。

状态模式（State Pattern）允许一个对象在其内部状态改变时修改它的行为。

策略模式（Strategy Pattern）将算法封装到类里。

模板方法模式（Template Method Pattern）提供了算法的一个抽象定义。

访问者模式（Visitor Pattern）在不改变类的前提下，为一个类添加多种操作。

第21章 职 责 链

职责链（Chain of Responsibility）模式允许多个类处理同一个请求，而不需要了解彼此的功能。它在类之间提供了一种松散的耦合：类之间惟一的联系就是相互间传递的请求。请求在类之间传递，直到其中一个类处理它为止。

这种链模式的一个例子就是类似于图21.1所示的帮助系统，这是一个简单的应用程序，其中有不同种类的帮助，在应用程序的每个屏幕区域，都可以找到相应的帮助，但在窗口的背景区域，应该只能找到普通的帮助才比较恰当。

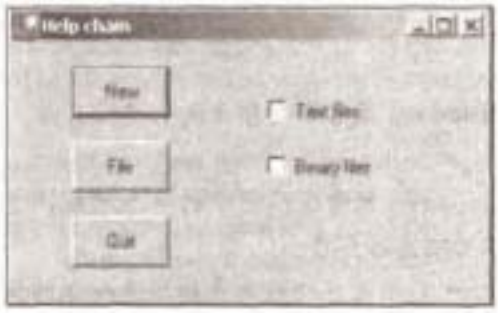


图21.1 一个有几种不同帮助的简单应用程序

当选择一个区域寻求帮助时，对应的可视化控件就将它的ID或名字转发给链。假设选择的是New按钮。如果第一个模块能处理New按钮，它就显示出相应的帮助信息；如果不能，就将请求转发给下一个模块。最后，请求信息转发到All buttons类，它显示一条关于按钮如何工作的通用信息。如果没有通用的按钮帮助模块，请求转发到通用帮助模块，它告诉读者系统一般是如何工作的。如果它也不存在，请求会被丢弃，没有信息显示。这个过程可以用图21.2说明。

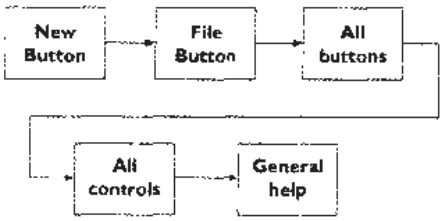


图21.2 一个简单的职责链

我们在这个例子中发现两个要点：第一，链的组织是从最特殊的到最一般的；第二，不能保证请求在任何情况下都会有响应。以后会看到，可以采用观察者（Observer）模式所提供的一种将改动通知给多个类的方式。

21.1 适用范围

职责链将程序中每个对象能做什么的内容隔离，它是设计模式的一个好例子。换句话说，职责链减少了对象之间的耦合，每个对象都能独立操作。职责链也可用于构成主程序的对象和包含其他对象实例的对象。下列几种情况适合于采用这种模式：

- 具有相同方法的几个对象都适合于执行程序请求的操作，但由对象决定由谁去完成操作，比把决策建立在调用代码中更合适。
- 其中某个对象可能最适合于处理请求，但你不想通过一系列if-else语句或switch语句去选择一个特定的对象。
- 程序执行时，需要向处理选项链中添加新的对象。
- 在多个对象都能执行一个请求的情况下，你不想把这些相互作用的内容放在调用程序里。

21.2 示例代码

我们刚才描述的帮助系统与第一个例子关系不大。这里先考虑一个简单的、可视化的命令-解释器程序（图21.3），该程序显示键入命令的结果，它可以说明职责链是如何工作的。尽管我们对该案例做了限定以使其代码更容易处理，但以后会看到，职责链通常都用于分析器和编译器。

本例中，命令可以是下列几种形式：

- 图像的文件名。
- 一般文件名。
- 颜色名。
- 其他命令。

对前三种情况，我们显示出请求的具体结果，对第四种情况，只显示请求文本本身。

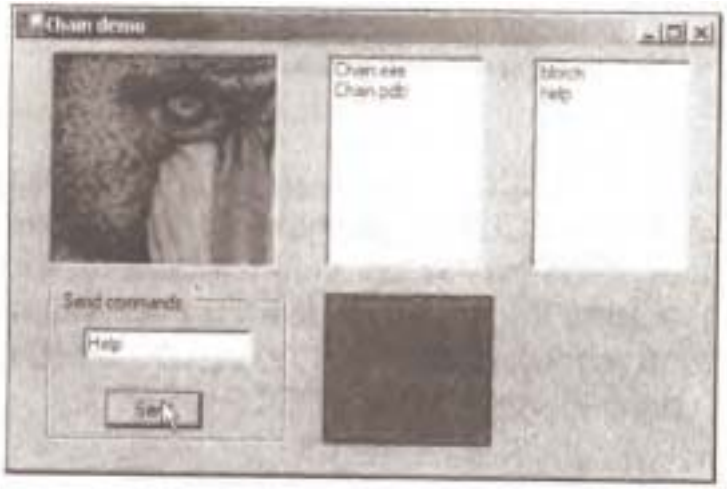


图21.3 一个可视化的命令-解释器程序，根据输入的命令对四个面板中的一个进行操作

在前面的示例系统里，我们做以下事情：

1. 键入“Mandrill”，能看到图像Mandrill.jpg的显示。
2. 然后键入“File”，相应的文件名显示在中央的列表框里。
3. 接下来键入“blue”，对应的颜色显示在下面的中央面板里。

最后，如果键入的既不是文件名也不是颜色，键入的文本会显示在最右边的列表框中。这种处理过程可用图21.4说明。

图21.4 图21.3程序中的命令链的工作方式

为了编写这个简单的职责链程序，先创建一个抽象类 Chain。

```
public abstract class Chain      {
       //describes how all chains work
       private bool hasLink;
       protected Chain chn;
       public Chain()        {
              hasLink = false;
       }
       //you must implement this in derived classes
       public abstract void sendToChain(string mesg);
       //-----
       public void addToChain(Chain c) {
       //add new element to chain
              chn = c;
              hasLink = true;          //flag existence
       }
       //-----
       public Chain getChain() {
              return chn;              //get the chain link
       }
       //-----
       public bool hasChain() {
              return hasLink;          //true if linked to another
       }
       //-----
       protected void sendChain(string mesg) {
       //send message on down the chain
              if(chn != null)
                     chn.sendToChain (mesg);
       }
}
```

其中，addToChain方法向类链添加另外一个类，getChain方法返回的是正转发消息的当前类。这两个方法允许我们动态修改链，以及向一个已存在的链添加另外的类。sendToChain方法将消息转发给链中的下一个对象，而受保护的 sendChain 方法在后面的链接不空时沿着链发送消息。

主程序汇集了 Chain 的所有派生类，并在每个类中设置了一个对相应控件的引用。先创建 ImageChain类，它接收消息字符串并查找具有相应名字的.jpg文件。如果找到，就将其显示在图像控件中，如果找不到，就将命令发送给链中的下一个元素。

```
public class ImageChain :Chain   {
       PictureBox picBox;          //image goes here
       //-----
       public ImageChain(PictureBox pc)      {
              picBox = pc;         //save reference
       }
       //-----
       public override void sendToChain(string mesg) {
              //put image in picture box
              string fname = mesg + ".jpg";
              //assume jpg filename
              csFile fl = new csFile(fname);
              if(fl.exists())
                     picBox.Image = new Bitmap(fname);
              else{
                     if (hasChain()){   //send off down chain
                            chn.sendToChain(mesg);
```

```
                }
            }

        }
    }
```

按照同样的方式创建ColorChain类，它将消息解释为颜色名，如果能解释就显示该颜色。本例中只解释了三种颜色，也可以实现其他颜色。注意，我们用到了一个颜色对象的Hashtable（哈希表），它以字符串名作为关键字，把颜色名（即消息）当做哈希表关键字去解释消息。

```
public class ColorChain : Chain {
        private Hashtable colHash; //color list kept here
        private Panel panel;                //color goes here
        //-----
        public ColorChain(Panel pnl)                        {
                panel = pnl;              //save reference
                //create Hash table to correlate color names
                //with actual Color objects
                colHash = new Hashtable ();
                colHash.Add ("red", Color.Red);
                colHash.Add ("green", Color.Green);
                colHash.Add ("blue", Color.Blue);
        }
        //-----
        public override void sendToChain(string mesg) {
                mesg = mesg.ToLower ();
                try {
                      Color c = (Color)colHash[ mesg];
                      //if this is a color, put it in the panel
                      panel.BackColor =c;
                }
                catch (NullReferenceException e) {
                      //send on if this doesn't work
                      sendChain(mesg);
                }

        }
}
```

21.3 列表框

文件列表和不能识别的命令列表都属于列表框（ListBox）。如果消息和一个文件名相匹配，则文件名显示在file列表框中，如果不匹配，则将消息传递给链中的NoCmd元素。

```
public override void sendToChain( string mesg) {
      //if the string matches any part of a filename
      //put those filenames in the file list box
      string[ ] files;
      string fname = mesg + "*.*";
      files = Directory.GetFiles(
            Directory.GetCurrentDirectory(), fname);
      //add them all to the listbox
      if (files.Length > 0){
            for (int i = 0; i< files.Length; i++) {
                  csFile vbf = new csFile(files[ i]);
                  flist.Items.Add(vbf.getRootName());
```

```
            }
        }
        else {
            if ( hasChain()) {
                chn.sendToChain(mesg);
            }
        }
}
```

NoCmd 类与此很类似，但它没有传递数据的后继类。

```
public class NoCmd :Chain  {
        private ListBox lsNocmd;   //commands go here
        //-----
        public NoCmd(ListBox lb)        {
            lsNocmd = lb;              //copy reference
        }
        //-----
        public override void sendToChain(string mesg) {
            //adds unknown commands to list box
            lsNocmd.Items.Add (mesg);
        }
}
```

最后，我们在窗体加载的子程序（即 init 方法）中将这些类连接在一起创建链。

```
private void init() {
        //set up chains
        ColorChain clrChain = new ColorChain(pnlColor);
        FileChain flChain = new FileChain(lsFiles);
        NoCmd noChain = new NoCmd(lsNocmd);
        //create chain links
        chn = new ImageChain(picImage);
        chn.addToChain(clrChain);
        clrChain.addToChain(flChain);
        flChain.addToChain(noChain);
}
```

单击 Send 按钮，它接收文本框中的当前消息并沿着链向前发送，我们通过这种方式开始访问链。

```
private void btSend_Click(object sender , EventArgs e )  {
                    chn.sendToChain ( txCommand.Text );
}
```

在图 21.5 的 UML 图中可以看出这些类之间的关系。

Sender 类是实现 Chain 接口的初始类，它接收到按钮单击事件，并从文本域中获取文本，将命令传递到 ImageChain 类、FileChain 类、ColorChain 类，最后到 NoCmd 类。

21.4 设计一个帮助系统

正如开始时提到的，帮助系统是讲解如何使用职责链模式的好例子。前面已经概括地给出了编写这种链的方法，接下来设计一个具有若干控件的窗口的帮助系统。程序在用户按下 F1（帮助）键时弹出一个帮助消息对话框（如图 21.6 所示）。按下 F1 键时，根据选择的控件显示不同的消息。

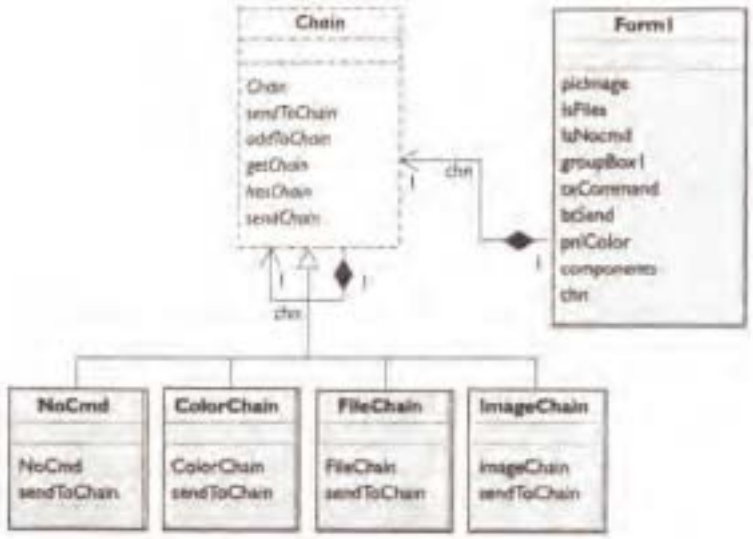


图21.5 职责链程序的类结构

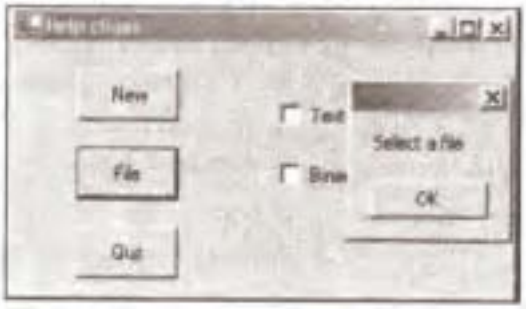


图21.6 一个简单的帮助程序的演示

在前面的例子里，用户若选择了Quit按钮，则没有与之关联的专门信息。链把帮助请求转发给通用按钮帮助对象，它在右边显示出相应的信息。

为了编写该帮助链系统，先创建一个抽象类Chain，它处理的是控件而不是消息。注意，没有消息传递给sendToChain方法，当前控件保存在类里。

```
public abstract class Chain{
       //describes how all chains work
       private bool hasLink;
       protected Control control;
       protected Chain chn;
       protected string message;

       public Chain(Control c, string mesg) {
              hasLink = false;
              control = c;        //save the control
              message = mesg;
       }

       public abstract void sendToChain();
       //-----
       public void addToChain(Chain c) {
              //add new element to chain
              chn = c;
              hasLink = true;          //flag existence
       }
```

```
        //-----
        public Chain getChain() {
               return chn;    //get the chain link
        }
        //-----
        public bool hasChain() {
               return hasLink;          //true if linked to another
               }
        }
        //-----
        protected void sendChain() {
        //send message on down the chain
               if(chn != null)
                      chn.sendToChain ();
        }
}
```

接下来为每类帮助信息创建专门的类。按照先前说明的，有下列几种情况的帮助信息：

- New按钮。
- File按钮。
- 通用按钮。
- 通用可视化控件（包括复选框）。

C#中，一个控件总会具有焦点，实际上并不需要窗口的焦点，但为了完整性还是包含了一个。为每种帮助信息创建单独的类，并将不同的控件分配给它们，这样做并没有太多的好处。我们换个做法，创建一个通用类ControlChain，把控件和帮助信息传递进来，然后在类的内部检查控件是否具有焦点，如果有，显示相关的帮助信息。

```
public class ControlChain:Chain {
       public ControlChain(Control c, string mesg):base(c, mesg) {}
       public override void sendToChain() {
              //if it has the focus display the message
              if (control.Focused ) {
                     MessageBox.Show (message);
              }
              else
                   //otherweise pass on down the chain
                   sendChain();
       }
}
```

最后，还需要处理一种特殊的情况：在链的末尾显示一条信息，而不考虑控件是否具有焦点，这就是EndChain类，其目的是为了程序的完整性，尽管一个控件总有可能获得焦点，但未必将来就一定调用它。

```
public class EndChain:Chain       {
      public EndChain(Control c, string mesg):base(c, mesg){}
      //default message display class
      public override void sendToChain() {
             MessageBox.Show (message);
      }
}
```

在窗体初始化代码里，按如下形式构建链。

```
chn = new ControlChain(btNew, "Create new files");
Chain fl =new ControlChain (btFile, "Select a file");
chn.addToChain (fl);
Chain bq = new ControlChain (btQuit, "Exit from program");
fl.addToChain (bq);
Chain cb =new ControlChain (ckBinary, "Use binary files");
bq.addToChain (cb);
Chain ct =  new ControlChain (ckText, "Use text files");
cb.addToChain (ct);
Chain ce = new EndChain (this, "General message");
ct.addToChain (ce);
```

21.4.1 接收帮助命令

现在需要指派监听器查找F1按键，读者可能首先会想到，我们会需要五个这样的监听器——分别针对三个按钮和两个复选框。可以只用一个KeyDown事件监听器，将它指派给每个控件。

```
KeyEventHandler keyev =  new KeyEventHandler(Form1_KeyDown);
      btNew.KeyDown += keyev;
      btFile.KeyDown += keyev;
      btQuit.KeyDown += keyev;
      ckBinary.KeyDown += keyev;
      ckText.KeyDown += keyev;
```

接下来，KeyDown事件在F1键按下时启动链。

```
private void Form1_KeyDown(object sender, KeyEventArgs e) {
      if(e.KeyCode==Keys.F1)
           chn.sendToChain();
}
```

在图21.7中给出该帮助系统的完整类图。

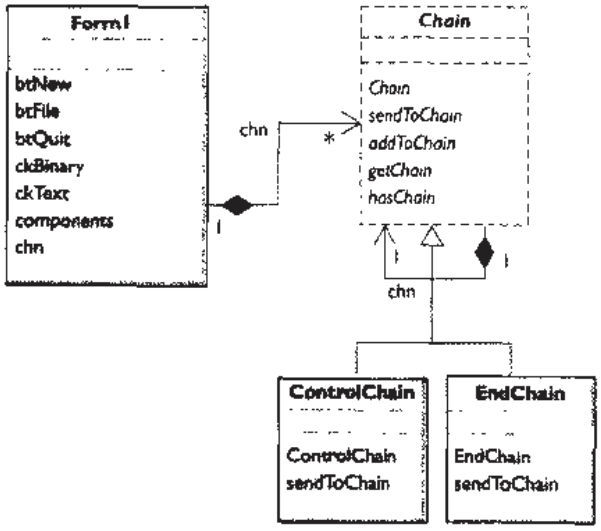


图21.7 帮助系统的类图

21.5 链的树形结构

当然，职责链不一定非是线性的。“Smalltalk Companion”中指出，职责链通常是一种树结构，许多的特殊入口节点都指向上面的最通用节点，如图21.8所示。

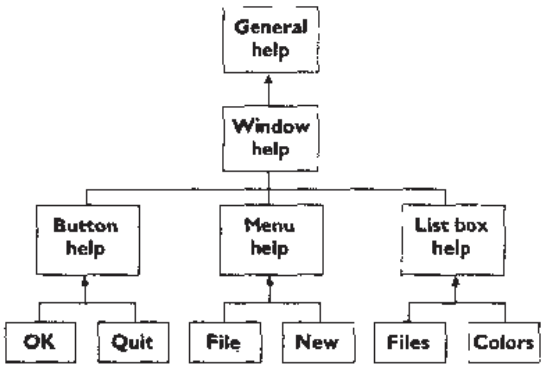


图 21.8 按树形结构实现的职责链

然而，这种结构意味着每个按钮或它的处理句柄都要知道从哪里进入链，在某些情况下这会将设计复杂化，而且可能根本不需要链。

处理树形结构的另一种方法是，只有一个入口点，它分岔到专门的按钮、菜单或其他未命名类型，然后不再分岔，如前所述，直到更通用的帮助实例。这么复杂的做法是毫无道理的——读者可以把所有的类安排到一个链中，从底部开始，由左至右，再到上一行，最后会遍历整个系统，如图 21.9 所示。

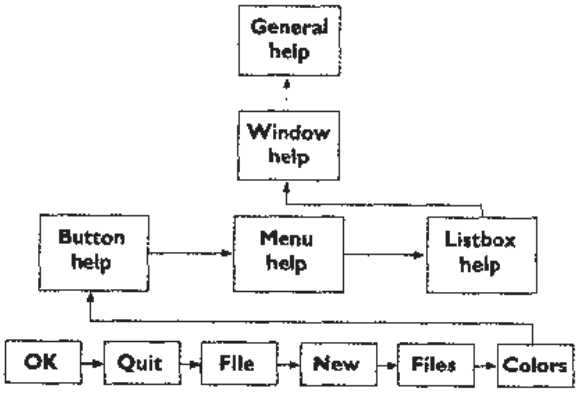


图 21.9 按线性链表实现的同一个职责链

21.6 请求的种类

沿着职责链传递的请求或消息，可以比在例子里惯用的字符串或控件复杂得多。这种信息可以包含各种数据类型，或一个完整的带有多个方法的对象。由于链中的各种类可能会用到请求对象的不同属性，所以可以设计一个抽象的 Request 类，以及多个带有不同方法的派生类。

21.7 C# 中的例子

在 C# 内部，窗口接收各种事件，如 MouseMove 事件，然后将事件转发给窗体所包含的控件。C# 中只有最后的控件接收消息，而在其他语言里，每个内部控件都能接收消息。这是职责链模式的一种典型实现。我们通常都认为，C# 的类继承结构本身就是该模式的一个例子。如果在底层的派生类里调用一个方法，则该方法调用沿着继承链向上传递，直到发现含有该方法的第一个父类为止，而更上层的、包含该方法其他实现的父类不会起作用。

以后还会看到，职责链适合于实现解释器（Interpreter），在后面讨论的解释器模式里会用到一个职责链。

21.8 职责链

同其他许多模式一样，该模式的主要目的是减少对象之间的耦合。一个对象只需知道如何将请求转发给其他对象。

链中的每个C#对象都是“自治”的，它对其他对象一无所知，只需判断它本身能否满足请求。这样，既能独立编写每个对象，又很容易构建链。

你可以决定，链中的最后一个对象是以默认方式处理它收到的所有请求，还是简单地抛弃请求。不管怎样，你必须知道链中最后一个可用的对象是哪一个。

最后，由于C#不提供多继承，有时候需要基类Chain是一个接口，而不是一个抽象类，这样，每个对象可继承另外一个层次结构，如我们前面让对象都继承于Control。这种方法的缺点是，必须在每个模块中分别实现连接、发送、转发的代码，或者正如我们在这一章里所做的那样，子类化一个实现Chain接口的具体类。

21.9 思考题

如何使用职责链实现邮件过滤器。

21.10 随书附带光盘中的程序

Program showing how a help system can be implemented	`\Chain\HelpChain`
Chain of file and image displays	`\Chain\Chain`

第22章　命令模式

职责链沿着类链转发请求，而命令模式（Command Pattern）只将请求转发给一个特定对象。命令模式把一个申请特定操作的请求封装到一个对象中，并给该对象一个众所周知的公共接口，使客户端不用了解实际执行的操作就能产生请求，也可以使你改变该操作而丝毫不影响客户端程序。

22.1　动机

建立一个C#用户界面时，需要提供菜单项、按钮、复选框等控件，让用户告诉程序做什么。用户选择其中的一个控件后，程序收到一个单击事件，放在用户接口的一个专门例程中处理。我们构建一个非常简单的程序，可以选择其中的菜单项FileIOpen和FileIExit，单击一个标题为Red的按钮，能将窗口的背景改为红色。程序如图22.1所示。

图22.1　一个接收按钮事件和菜单项事件的简单程序

该程序包含一个File菜单对象，并添加了mnuOpen和mnuExit菜单项，它还包含一个称为btnRed的按钮。在设计阶段，单击其中的任何一项都会在Form类里产生一个小的方法，控件被单击时调用该方法。

如果只有几个菜单项和按钮，这种方法没有任何问题，但是，当拥有几十个菜单项和多个按钮时，Form代码模块将会变得相当庞大。另外，我们最后还想让按钮和菜单项都能执行Red命令（上述方法不适合这种情况）。

22.2　命令对象

确保每个对象能直接收到自己的命令的一种方法是，使用命令模式并创建单独的Command对象。Command对象中总有一个Execute()方法，在对象上的操作发生时调用它。最简单的Command对象至少要实现下面的接口：

```
public interface Command    {
       void Execute();
}
```

使用该接口的一个目的是，将用户界面代码与程序要完成的操作分离，如下所示：

```
private void commandClick(object sender,EventArgs e)  {
      Command comd=(Command)sender;
      comd.Execute() ;
}
```

这个事件可以关联到每个能被单击的用户界面元素上，每个元素都可以有自己的Execute方法实现，只要从支持该Command接口的Button类和MenuItem类派生出一个新类就可以了。

接下来我们为每个执行预定操作的对象提供一个Execute方法，通过这种方法将关于做什么的细节封装到所属的对象里，而不是让另外一部分程序做这些决定。

命令模式的一个重要目标是，将程序和用户界面对象与它们所引起的操作完全隔离。换句话说，这些程序对象彼此之间应该完全隔离，不应该知道其他对象是如何工作的。用户界面接到一个命令后，告诉Command对象去执行命令所规定的操作。UI不知道也不应该知道要执行什么任务。这使UI类和特定命令的执行相隔离，可以修改或完全改变操作代码而不必改动包含该用户界面的类。

若需要告诉程序在资源可用时去执行命令，而不是立即执行命令，则也可以使用Command对象，这类情况下需要对将要执行的命令排队。最后一点，可以用Command对象记住操作，这样就能支持Undo请求。

22.3 构建Command对象

为一个程序构建Command对象有几种方式，每种方式都有一些优点。我们先从最简单的方式开始：创建新类，在每个类中实现Command接口。对于将背景变红的按钮，我们从Button类中派生出RedButton类，并让它包含一个Execute方法，这样才符合Command接口的要求。

```
public class RedButton : System.Windows.Forms.Button, Command   {
      //A Command button that turns the background red
      private System.ComponentModel.Container components = null;
      //-----
      public  void Execute() {
            Control c = this.Parent;
            c.BackColor =Color.Red;
            this.BackColor =Color.LightGray;
      }
      public RedButton()        {
            InitializeComponent();
      }
```

在这个实现里，找到按钮的父类（Form）并将它的背景置为红色，通过这种方式给出背景窗口。就像把参数传给构造函数一样，这里非常简单地把Form传递进来。

记住，为了创建一个能用在IDE环境中且派生于Button的类，需要创建一个用户控件，并将该控件继承的基类由UserControl改为Button，然后编译它。这会在工具箱上添加一个图标，可以将该图标拖到Form1窗口上。

为了创建一个实现Command接口的MenuItem，要用到工具栏上而的MainMenu控件，将其命名为MenuBar，菜单设计器如图22.2所示。

除了使用菜单设计器外，就像接下来FileExit要看到的，使用代码创建MainMenu也很容易。从MenuItem类派生出FileOpen类和FileExit类。

图 22.2 菜单设计器界面

因为在窗体设计器里无法添加这些类，所以必须用程序代码去添加。

```
private void init() {
       //create a main menu and install it
       MainMenu main = new MainMenu();
       this.Menu =main;

       //create a click-event handler
       EventHandler evh = new EventHandler (commandClick);
       btRed.Click += evh;          //add to existing red button

       //create a "File" top level entry
       MenuItem file = new MenuItem("File");

       //create File Open command
       FileOpen mnflo = new FileOpen ();
       mnflo.Click += evh;        //add same handler
       main.MenuItems.Add ( file );

       //create a File-Exit command
       FileExit fex = new FileExit(this);
       file.MenuItems.AddRange( new MenuItem[]{ mnflo, fex} );
       fex.Click += evh;          //add same handler
}
```

下面是FileExit类：

```
public class FileExit :MenuItem, Command   {
       private Form form;
       //----------
       public FileExit(Form frm) :base ("Exit") {
              form = frm;
       }
       //----------
       public void Execute() {
              form.Close ();
       }
 }
```

当调用File | Exit菜单项的Execute方法时，FilelExit命令会调用该类。这可以使我们简化用户接口代码，但是，它需要为每个要执行的操作创建一个新类并实例化它。

需要特定参数来工作的类要把那些参数传递给构造函数或set方法，例如，FilelExit命令需要你传递给它一个Form对象的实例，这样它才能关闭窗体。

```
//create a File-Exit command
FileExit fex = new FileExit(this);
```

22.4 命令模式的效果

命令模式的主要缺点是，增加了使程序散乱的小类。不过，即使有单独的单击事件，也通常都是调用小的私有方法去完成具体功能。最后的结果是，私有方法和我们这些小类的代码长度几乎一样，因此，构建Command类和编写较多的方法在复杂性上通常没有区别，主要区别是命令模式生成的小类更容易理解。

22.5 CommandHolder接口

尽管将操作封装到一个Command对象里是有利的，但把这个对象绑定到引起操作的元素（例如菜单项或按钮）中不是命令模式的本意。相反，Command对象应该与调用客户程序真正地隔离，这样能分别改变调用程序和命令操作的细节。我们可以让菜单类和按钮类成为独立存在的Command对象的容器，而不是让命令成为菜单或按钮的一部分。这里让UI元素实现一个CommandHolder接口。

```
public interface CommandHolder   {
      Command getCommand();
      void setCommand(Command cmd);
}
```

这个简单的接口说明了两个方法：把一个命令对象放到类中的方法，以及取出命令对象去执行的方法。当我们有几种方式可调用相同操作的时候（比如我们既有Red按钮又有Red菜单项时），这一点是相当重要的。这种情况下，读者当然不想在MenuItem和Button类里都执行同样的代码，相反，应该从两个类中取出同一命令对象的引用，再执行该命令。

接下来我们创建CommandMenu类，它实现了上述接口。

```
public class CommandMenu : MenuItem, CommandHolder    {
      private Command command;
      public CommandMenu(string name):base(name){}
      //-----
      public void setCommand (Command comd) {
            command = comd;
      }
      //-----
      public Command getCommand () {
            return command;
      }
}
```

这实际上简化了我们的程序，这样无需为每个要执行的操作创建一个单独的菜单类，只要创建菜单类的实例，并将不同的标签和Command对象传递给它们。

例如，RedCommand对象在构造函数里接收一个Form对象实例，并在Execute方法中将背景设为红色。

```
public class RedCommand : Command     {
      private Control window;
      //-----
      public RedCommand(Control win)       {
```

```
            window = win;
        }
        //-----
        void Command.Execute () {
               window.BackColor =Color.Red;
        }
 }
```

可以将该命令的实例设在RedButton对象和Red菜单项对象里，如下所示：

```
private void init() {
        //create a main menu and install it
        MainMenu main = new MainMenu();
        this.Menu =main;

        //create a click-event handler
        //note: btRed was added in the IDE
        EventHandler evh = new EventHandler (commandClick);
        btRed.Click += evh;         //add to existing red button
        RedCommand cRed = new RedCommand (this);
        btRed.setCommand (cRed);
        //create a "File" top level entry
        MenuItem file = new CommandMenu("File");
        main.MenuItems.Add ( file );

        //create File Open command
        CommandMenu mnuFlo = new CommandMenu("Open");
        mnuFlo.setCommand (new OpenCommand ());
        mnuFlo.Click += evh;                //add same handler

        //create a Red menu item, too
        CommandMenu mnuRed = new CommandMenu("Red");
        mnuRed.setCommand(cRed);
        mnuRed.Click += evh;                //add same handler

        //create a File-Exit command
        CommandMenu mnuFex = new CommandMenu("Exit");
        mnuFex.setCommand(new ExitCommand(this));
        file.MenuItems.AddRange(
               new CommandMenu[]{mnuFlo, mnuRed, mnuFex} );
        mnuFex.Click += evh;                //add same handler
}
```

使用CommandHolder这种方法，还需要创建各自的Command对象，但它们不再是用户接口类的一部分。例如，OpenCommand类就是下列形式：

```
public class OpenCommand :Command     {
        public OpenCommand()
        {}
        public void Execute() {
               OpenFileDialog fd = new OpenFileDialog ();
               fd.ShowDialog ();
        }
 }
```

接下来，单击事件处理方法需要从引起操作的UI对象获得具体的命令对象，然后执行该命令。

```
private void commandClick(object sender, EventArgs e)  {
      Command comd = ((CommandHolder)sender).getCommand();
      comd.Execute();
}
```

这比原来的程序稍微复杂些，并将操作和用户界面元素隔离开来。图22.3中我们可以看到运行中的程序。

图22.3 使用CommandHolder接口的命令模式的菜单部分

从图22.4的UML图中，可以清楚地看到这些类和接口之间的关系。

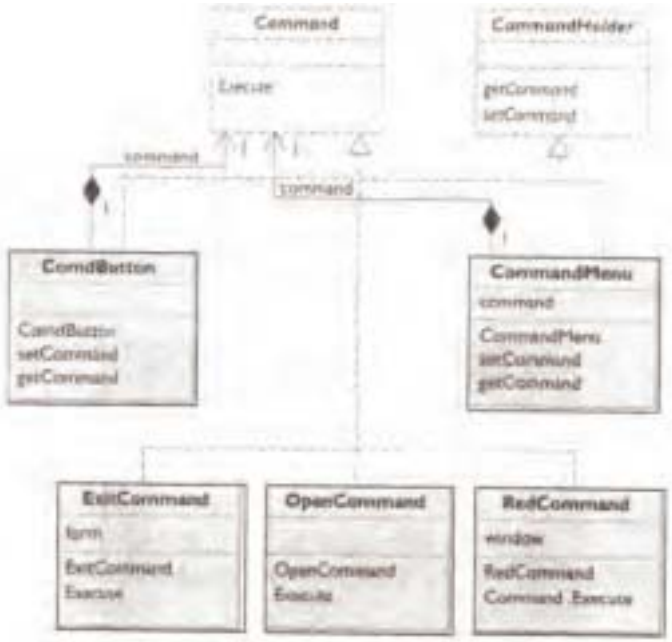


图22.4 采用CommandHolder接口方式的类图

22.6 提供Undo操作

使用命令设计模式的另一个主要原因是，它们提供了一个便捷的存储方法并能完成Undo功能。每个命令对象都记住刚做过的事，并在有Undo请求时，只要计算量和内存需求不太过分，就能恢复到刚才的状态。这里重定义顶层的Command接口，让它有三个方法：

```
public interface Command   {
      void Execute ();
```

```
        void Undo ();
        bool isUndo ();
}
```

接下来，必须让每个命令对象记录上一次的操作，这样它才能取消操作。这比上一个程序复杂些，因为交叉执行了许多命令，取消操作会导致一些滞后现象。另外，每个命令都需要保存有关命令每一次执行情况的足够信息，才能明确地完成取消操作。

取消命令的问题实际上是一个几部分的问题，首先，必须保存一个已执行命令的列表，其次，每个命令必须保持一个自身执行情况的列表。为了说明如何使用命令模式完成取消操作，我们考虑图 22.5 中的程序，它在屏幕上画出连续的红色线或蓝色线，使用两个按钮来分别绘制两种颜色的线。可以用 Undo 按钮取消刚才所画的线。

图 22.5 每次单击 Red 按钮和 Blue 按钮时绘制红色线和蓝色线的程序

如果单击几次 Undo 按钮，会希望不管原来按钮按下的顺序如何，最后画的几条线都消失，如图 22.6 所示。

图 22.6 单击几次 Undo 按钮后，图 22.5 中程序的输出情况

因此，任何支持取消操作的程序都需要一个所有执行过的命令的顺序表，每次单击一个按钮，都要将对应的命令添加到该表中。

```
private void commandClick(object sender, EventArgs e) {
      //get the command
      Command comd = ((CommandHolder)sencer).getCommand ();
      undoC.add (comd);     //add to undo list
      comd.Execute ();//and execute it
}
```

我们把添加Command对象的列表保存在Undo命令对象中，这样它才能很方便地访问该表。

```
public class UndoComd:Command    {
      private ArrayList undoList;
      public UndoComd()     {
            undoList = new ArrayList ();
      }
      //-----
      public void add(Command comd) {
            if(! comd.isUndo ()) {
                  undoList.Add (comd);
            }
      }
      //-----
      public bool isUndo() {
            return true;
      }
      //-----
      public void Undo() {                  }
      //-----
      public void Execute() {
            int index = undoList.Count - 1;
            if (index >= 0) {
                  Command cmd = (Command)undoList[ index] ;
                  cmd.Undo();
                  undoList.RemoveAt(index);
            }
      }
}
```

undoCommand对象保存着Command的列表，而不是具体数据的列表。每个命令对象都调用它自己的Undo方法，用来执行实际的取消操作。注意，因为undoCommand对象实现了Command接口，因此，它也需要有一个Undo方法。但是，对这个简单例子来说，取消连续的Undo操作有点复杂。读者会注意到，由于我们已经把取消Undo命令定义为不做任何事，因此，除了undoCommand以外，add方法将所有的Command都添加到列表中。由于这个原因，新的Command接口包含了一个isUndo方法，它对RedCommand和BlueCommand对象返回false，对undoCommand对象返回true。

尽管RedCommand类和BlueCommand类都实现了修订后的Command接口，但它们是用不同的颜色从窗口的两边开始画线。每个类都在一个集合里保存了一个画过线的列表，集合中是一系列包含每条线坐标的DrawData对象。从红线或蓝线中取消一条线意味着，删除drawList集合中的最后一个DrawData对象。接下来，每个命令都要使屏幕重画。由于除了线的颜色和倾斜方向不同外，这两个类做的都是同样的事情，所以创建一个基类ColorCommand，再由它派生出RedCommand和BlueCommand类。下面是基类ColorCommand。

```
public class ColorCommand :Command
      {
            protected Color color;              //line color
            protected PictureBox pbox;          //box to draw in
            protected ArrayList drawList;       //list of lines
            protected int x, y, dx, dy;         //coordinates
            //------
```

```
        public ColorCommand(PictureBox pict)
    {
        pbox = pict;   //copy in picture box
        drawList = new ArrayList ();  //create list
    }
    //------
    public void Execute() {
        //create a new line to draw
        DrawData dl = new DrawData(x, y, dx, dy);
        drawList.Add(dl);   //add it to the list
        x = x + dx;         //increment the positions
        y = y + dy;
        pbox.Refresh();
    }
    //-----
    public bool isUndo() {
        return false;
    }
    //-----
    public void Undo() {
        DrawData dl;
        int index = drawList.Count - 1;
        if (index >= 0) {
            dl = (DrawData)drawList[ index] ;
            drawList.RemoveAt(index);
            x = dl.getX();
            y = dl.getY();
        }
        pbox.Refresh();
    }
    //-----
    public void draw(Graphics g) {
        Pen rpen = new Pen(color, 1);
        int h = pbox.Height;
        int w = pbox.Width;
    //draw all the lines in the list
        for (int i = 0; i < drawList.Count; i++) {
            DrawData dl = (DrawData)drawList[ i] ;
            g.DrawLine(rpen, dl.getX(), dl.getY(),
                    dl.getX() + dx, dl.getDy() + h);
        }
    }

}
```

注意，ColorCommand类中的draw方法重画了命令对象所存储的全部线段。派生类中的两个draw方法都由窗体的paint处理程序调用。

```
public void paintHandler(object sender, PaintEventArgs e)  {
    Graphics g = e.Graphics;
    blueC.draw(g);
    redC.draw(g);
}
```

接下来可以从ColorCommand类派生出RedCommand和BlueCommand类，下面是BlueCommand类。

```
public class BlueCommand :ColorCommand    {
    public BlueCommand(PictureBox pict):base(pict) {
        color = Color.Blue;
        x = pbox.Width;
        dx = -20;
        y = 0;
        dy = 0;
```

```
        }
    }
```

同样可以很容易地创建出 RedCommand 类。

```
public class RedCommand : ColorCommand {
        public RedCommand(PictureBox pict):base(pict){
                color = color.Red;
                x = 0;
                dx = 20;
                y = 0;
                dy = 0;
        }
}
```

我们在 Undo 程序中使用的类如图 22.7 所示。

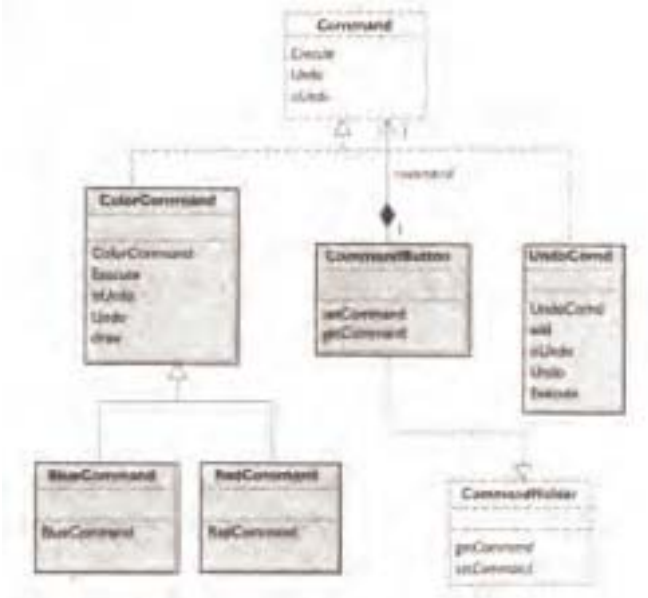


图 22.7 在命令模式中实现 Undo 操作的类图

22.7 思考题

1. 鼠标单击列表框项目和单选按钮可构成命令，单击多选列表框也可以表示为命令。设计一个包含这些特性的程序。
2. 乐天彩票系统使用一个随机数生成器将整数限制在1~50的范围内，号码选择的间隔由一个随机定时器选定，选择的每个号码必须是惟一的。设计一个模式来选出每周的中奖号码。

22.8 随书附带光盘中的程序

Buttons and menus using Command pattern	`\Command\ButtonMenu`
C# program showing line drawing and Undo	`\Command\UndoCommand`
C# program showing CommandHolder interface	`\Command\ComdHolder`

第23章　解释器模式

某些程序受益于用一种语言描述它们能完成的操作。解释器模式（Interpreter Pattern）通常描述的是，为某种语言定义一个文法，并用该文法解释语言中的语句。

23.1　动机

当一个程序要处理许多不同但又有些类似的问题时，用一种简单的语言描述这些问题，然后让程序解释该语言，是非常便利的方法。这种方案要么像许多办公套件程序所提供的宏语言记录工具一样简单，要么像 Visual Basic for Application（VBA）一样复杂。VBA 不但包含在 Microsoft Office 产品中，而且很容易嵌入在任何第三方产品中。

要解决的问题之一是，判别在什么时候采用一种语言比较合适。宏语言记录器能够记录菜单操作和按键操作，以便将来回放这些操作，它很少被当做一种语言，实际上它也不可能具有书写格式或文法。另一方面，VBA 这样的语言太复杂，它们远远超越了一个应用程序开发人员的能力。此外，嵌入商业语言通常都需要许可费用，除了大型的开发组外，对其他人很少具有吸引力。

23.2　适用性

正如“Smalltalk Companion”所提到的，辨别哪一种情况适合于使用解释器是一个相当大的问题，没有正规语言或编译器技能训练的程序员经常会忽略这种方法。通常有三种情形适合于使用语言。

1. **当读者需要用一个命令解释器分析用户命令时。**用户可以通过输入查询获得各种答案。
2. **当程序需要分析一个代数串时。**这种情况非常显而易见。用户输入某个方程式，要求程序根据计算结果执行相应的操作。这种情况通常出现在数学绘图程序中，程序根据它求解的方程式对曲线或曲面进行着色。类似于 Mathematica 的程序和 Origin 的图形绘制软件包都是采用解释器方式。
3. **当程序要生成各种形式的输出时。**这种情况不太明显，但相当有用。考虑一个程序，它可以按任意顺序显示几列数据，并能用不同的方法对它们排序。这类程序通常称做报表生成器（Report Generators），它的底层数据可以存在一个关系数据库中，报表程序的用户接口通常比数据库用的 SQL 语言简单得多。实际上，在某些情况下，简单的报表语言可以由报表程序解释，并翻译成 SQL 语言。

23.3　一个简单的报表例子

考虑一个简单的报表生成器，它对一张表中的五列数据操作，返回这些数据的不同报表。假设我们从一场游泳比赛中获得下列结果：

Amanda	McCarthy	12	WCA	29.28
Jamie	Falco	12	HNHS	29.80
Meaghan	O'Donnell	12	EDST	30.00
Greer	Gibbs	12	CDEV	30.04
Rhiannon	Jeffrey	11	WYW	30.04
Sophie	Connolly	12	WAC	30.05
Dana	Helyer	12	ARAC	30.18

这五列分别是 frname，lname，age，club 和 time。如果考虑 51 名游泳选手的比赛结果，我们会意识到，按俱乐部、姓氏或年龄对这些结果排序是很方便的。由于我们要根据这些数据生成许多报表，而且报表中列的顺序和排序都有变化，所以使用语言处理这些报表是一种有效的方式。

这里为这种排序定义一个非常简单的非递归文法：

```
Print lname frname club time Sortby club Thenby time
```

针对本例的意图，定义下面三个动词：

```
Print
Sortby
Thenby
```

还要定义前面给出的五个列的名称。

```
Frname
Lname
Age
Club
Time
```

为方便起见，假设该语言区分大小写。还注意到，该语言的简单文法没有标点符号，下面是文法的简洁形式：

```
Print var [ var] [ sortby var [ thenby var] ]
```

最后一点，该文法只有一个主要动词，面且每个句子都是一个声明语句，该文法没有赋值语句或计算功能。

23.4　解释语言

解释一种语言分为三个步骤：

1. 将语言中的符号分析成 Token。
2. 将 Token 归约成动作。
3. 执行动作。

使用StringTokenizer扫描每个句子，然后用一个数字代替每个单词，以此将语言的符号分析成 Token。通常，语法分析器将每个分析过的 Token 压入一个栈中，我们这里也采用这种方式。用一个 ArrayList（数组列表）实现 Stack 类，类中有 push，pop，top 和 nextTop 方法，它们检查并处理栈的内容。对前面的句子（排序）分析完成后，栈中内容如下：

```
类型    Token
Var     Time    <-- 栈顶
Verb    Thenby
Var     Club
Verb    Sortby
```

```
Var     Time
Var     Club
Var     Frname
Var     Lname
Verb    Print
```

可是，我们很快意识到，动词（Verb）Thenby并不像说明的那样，它没有实际意义，所以把分析Token和跳过Thenby放在一起更合理。原来的堆栈变成如下形式：

```
Time
Club
Sortby
Time
Club
Frname
Lname
Print
```

23.5 用于分析的对象

在分析过程中，我们不是把Token编号压入栈中，而是把一个具有类型和属性值的ParseObject对象压入栈中。

```
public class ParseObject   {
      public const int VERB=1000;
      public const int VAR=1010;
      public const int MULTVAR=1020;
      protected int value, type;
      //----
      public ParseObject(int val, int typ) {
            value = val;
            type = typ;
      }
      //----
      public int getValue() {
            return value;
      }
      //----
      public int getType() {
            return type;
      }
}
```

这些对象可以是VERB类型或VAR类型。接下来，将该对象扩展为ParseVerb和ParseVar对象，其中的value域对ParseVerb来说可以是PRINT或SORT，对ParseVar来说可以是FRNAME，LNAME等。为了以后归约分析列表，从ParseVerb派生出Print类和Sort类。

图23.1给出了简单的层次结构。

分析过程的代码如下所示，它使用了StringTokenizer对象和ParseObject对象。这里给出的是Parser类的主要部分。

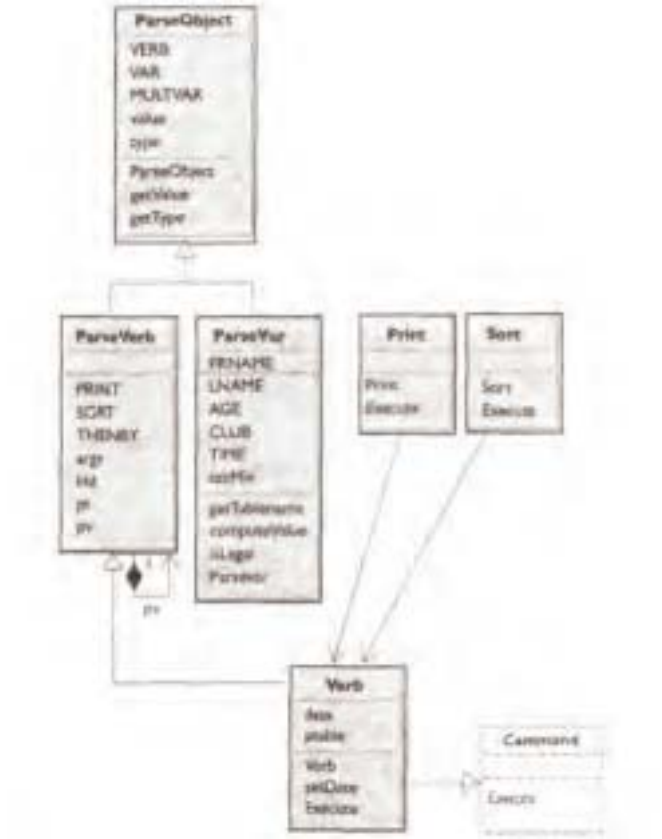


图 23.1　一个说明解释器模式的简单分析程序的层次结构

```
public class Parser   {
      private Stack stk;
      private ArrayList actionList;
      private Data dat;
      private ListBox ptable;
      private Chain chn;
      //-----
      public Parser(string line, KidData kd, ListBox pt) {
            stk = new Stack ();
            //list of verbs accumulates here
            actionList = new ArrayList ();
            setData(kd, pt);
            buildStack(line);    //create token stack
            buildChain();        //create chain of responsibility
      }
      //-----
      private void buildChain() {
            chn = new VarVarParse(); //start of chain
            VarMultvarParse vmvp = new VarMultvarParse();
            chn.addToChain(vmvp);
            MultVarVarParse mvvp = new MultVarVarParse();
            vmvp.addToChain(mvvp);
            VerbMultvarParse vrvp = new VerbMultvarParse();
            mvvp.addToChain(vrvp);
            VerbVarParse vvp = new VerbVarParse();
            vrvp.addToChain(vvp);
            VerbAction va = new VerbAction(actionList);
```

```
            vvp.addToChain(va);
            Nomatch nom = new Nomatch ();      //error handler
            va.addToChain (nom);
        }
        //-----
        public void setData(KidData kd, ListBox pt) {
            dat = new Data(kd.getData ());
            ptable = pt;
        }
        //-----
        private void buildStack(string s) {
            StringTokenizer tok = new StringTokenizer (s);
            while(tok.hasMoreElements () ) {
                ParseObject token = tokenize(tok.nextToken));
                    stk.push (token);
            }
        }
        //-----
        protected ParseObject tokenize(string s) {
            ParseObject obj;
            int type;
            try {
                obj = getVerb(s);
                type = obj.getType ();
            }
            catch(NullReferenceException) {
                obj = getVar(s);
            }
            return obj;
        }
        //-----
        protected ParseVerb getVerb(string s) {
            ParseVerb v = new ParseVerb (s, dat, ptable);
            if(v.isLegal () )
                return v.getVerb (s);
            else
                return null;
        }
        //-----
        protected ParseVar getVar(string s) {
            ParseVar v = new ParseVar (s);
            if( v.isLegal())
                return v;
            else
                return null;
        }
}
```

如果 ParseVerb 类和 ParseVar 类都能够识别单词，就返回将 isLegal 设为 true 的对象。

```
public class ParseVerb:ParseObject          {
    protected const int PRINT = 100;
    protected const int SORT = 110;
    protected const int THENBY = 120;
    protected ArrayList args;
```

```
protected Data kid;
protected ListBox pt;
protected ParseVerb pv;
//-----
public ParseVerb(string s, Data kd, ListBox ls):
              base(-1, VERB)   {
       args = new ArrayList ();
       kid = kd;
       pt = ls;
       if(s.ToLower().Equals ("print")) {
              value = PRINT;
       }
       if(s.ToLower().Equals ("sortby")) {
              value = SORT;
       }
}
//------
public ParseVerb getVerb(string s) {
       pv = null;
       if(s.ToLower ().Equals ("print"))
              pv =new Print(s,kid, pt);
       if(s.ToLower ().Equals ("sortby"))
              pv = new Sort (s, kid, pt);
       return pv;
}
//-----
public void addArgs(MultVar mv) {
       args = mv.getVector ();
}
```

23.6 归约分析栈

对前面的简单文法，栈中的Token如下所示（栈中存储的是分析对象）：

```
Var
Var
Verb
Var
Var
Var
Var
Verb
```

我们一次归约栈中的一个Token，将连续的Var归约成一个MultVar类，直到参数都归约成Verb对象，如图23.2所示。

当把栈归约成一个动词时，该动词及其参数都放在一个动作列表中；当栈空时，执行动作。

创建一个Parser类，它是一个Command对象，当用户界面上的Go按钮按下时执行命令，以此完成全部的分析过程。

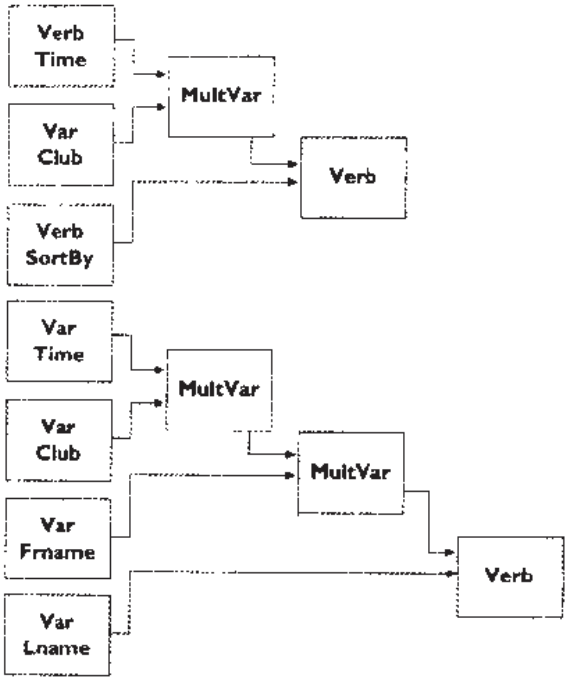


图 23.2 在分析过程中如何归约栈

```
private void btCompute_Click(object sender, EventArgs e) {
      parse();
}
private void parse() {
      Parser par = new Parser (txCommand.Text, kdata, lsResults);
      par.Execute ();
}
```

分析器本身只归约Token，如前面程序所示，它检查栈中的各种Token对，并针对五种情况中的每一种，将每个Token对归约成一个Token。

23.7 实现解释器模式

如果只用一系列的if语句，当然也能为该简单文法编写一个分析器。栈有六种可能的组态，针对每一种组态归约栈，直到栈中只剩下一个动词为止。由于把动词Print类和Sort类都作为Command对象，所以，在枚举动作列表时，可以一个接一个地执行这些命令。

解释器模式的真正优点是其灵活性。通过把每种分析情况作为单个对象，可以把分析树表示成一系列关联的对象，对栈进行连续归约。使用这种方案，不需要对程序做太多的改变，就能很容易修改分析规则，只要创建新的对象并将它们插入到分析树中。

根据 Gang of Four 的描述，解释器模式中有下列参与对象：

- **抽象表达式**（AbstractExpression）：声明一个抽象的解释操作。
- **终结符表达式**（TerminalExpression）：解释由文法中终结符组成的表达式。
- **非终结符表达式**（NonTerminalExpression）：解释文法中所有非终结符表达式。
- **上下文**（Context）：包含了分析器的一部分全局信息，在本例中是指 Token 栈。
- **客户**（Client）：根据前面的表达式类型构建语法树并调用解释操作。

23.7.1　语法树

为完成前面栈的分析而构建出的语法树相当简单。只需要查找前面定义的栈的每一种组态，并将它归约为一种可执行形式。实际上，实现该语法树最好的方法是使用一个职责链，它将栈的组态在类之间传递，直到其中的一个类能够识别该组态并对它执行操作。你可以决定，一次成功的栈归约是结束传递过程还是继续传递下去。在一个传递中，多个继承的链成员都对栈操作，是完全有可能的。栈空时，处理过程结束。在图 23.3 中，看到了单个分析链的元素的类图。

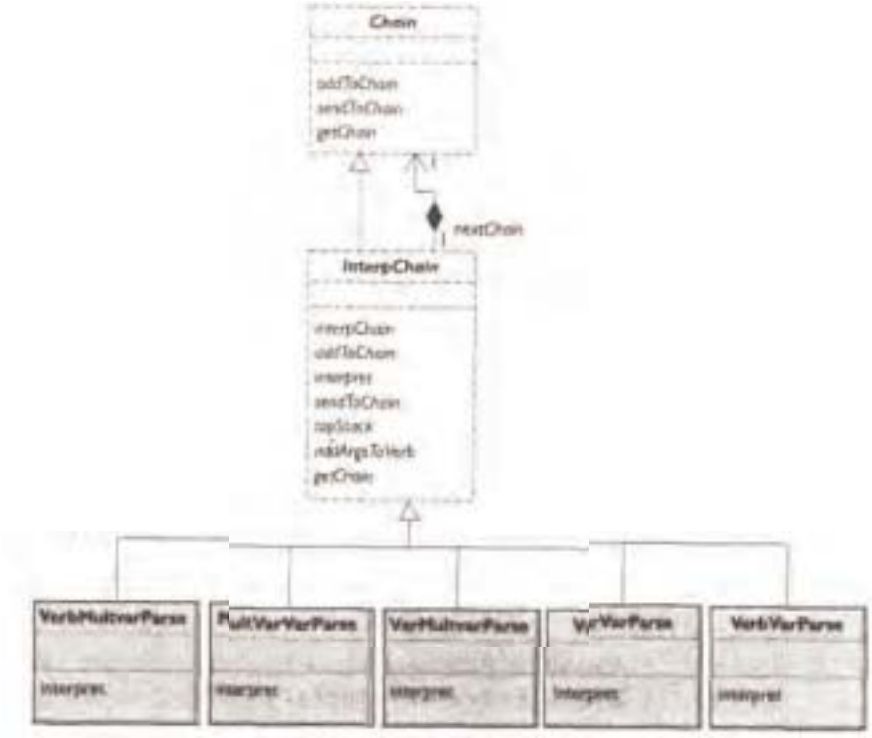


图 23.3　完成分析操作的类之间的相互作用

在该类结构中，先创建 AbstractExpression 解释器类 InterpChain。

```
public abstract class InterpChain:Chain {
       private Chain nextChain;
       protected Stack stk;
       private bool hasChain;
       //-----
       public InterpChain() {
              stk = new Stack ();
              hasChain = false;
       }
       //-----
       public void addToChain(Chain c) {
              nextChain = c;
              hasChain = true;
       }
       //-----
       public abstract bool interpret();
       //-----
       public void sendToChain(Stack stack) {
              stk = stack;
              if(! interpret()   ) {              //interpret stack
```

```
                nextChain.sendToChain (stk); //pass along
            }
        }
        //-----
        public bool topStack(int c1, int c2) {
                ParseObject p1, p2;
                p1 = stk.top ();
                p2 = stk.nextTop ();
                try{
                    return (p1.getType() == c1 && p2.getType() == c2);
                    }
                catch(NullReferenceException) {
                    return false;
                }
        }
        //-----
        public void addArgsToVerb() {
                ParseObject p = (ParseObject) stk.pop();
                ParseVerb v =  (ParseVerb) stk.pop();
                v.addArgs (p);
                stk.push (v);
        }
        //-----
        public Chain getChain() {
                return nextChain;
        }
```

这个类还包含了操作栈对象的方法，它的每个子类分别实现了interpret操作，以此完成对栈的归约。例如，VarVarParse类将栈中的两个连续变量归约成一个MultVar对象。

```
public class VarVarParse : InterpChain {
      public override bool interpret() {
            if(topStack(ParseVar.VAR,ParseVar.VAR)) {
                  //reduces VAR VAR to MULTVAR
                  ParseVar v1=(ParseVar)stk.pop();
                  ParseVar v2=(ParseVar)stk.pop();
                  MultVar mv = new MultVar(v2,v1);
                  stk.push(mv);
                  return true;
            }
            else
                  return false;
    }
}
```

在模式的实现中，栈就是参与对象Context，InterpChain的五个子类是参与对象NonTerminal-Expression，ActionVerb类将整个的动词和动作对象放到actionList中，它是参与对象TerminalExpression。

客户对象是Parser类，它根据键人的命令文本构建栈对象列表，并为各种解释器类构建职责链。我们刚才已经给出了Parser类的大部分内容。除此以外，它还实现命令模式，并沿着职责链发送栈，直到栈变空为止，然后，调用它的Execute方法，执行累积在动作列表中的动词。

```
  //executes parse and interpretation of command line
public void Execute() {
      while(stk.hasMoreElements () ) {
            chn.sendToChain (stk);
      }
```

```
        //now execute the verbs
        for(int i=0; i< actionList.Count; i++ ) {
                Verb v = (Verb)actionList[ i];
                v.setData (dat, ptable);
                v.Execute ();
        }
}
```

最终的可视化程序如图 23.4 所示。

图 23.4　解释器模式对文本域中的简单命令进行操作

23.8　解释器模式的效果

只要在程序中使用了解释器，就要为程序的用户提供一种简单的方法来输入该语言的命令，它既可以像我们前面提到的 Macro 记录按钮那样简单，也可以类似于前面程序中的可编辑文本域。

另外，引进一种语言及其文法，还需要对拼错的单词或放错的文法元素做相当大范围的错误检查，除非有可用的模板代码来实现这种检查，否则要编写大量的程序。此外，将这些错误通知给用户的有效方法也难以设计和实现。

在前面的 Interpreter 例子里，处理的惟一错误是，不识别的关键字没有转换成 ParseObject，也不能压入栈中。这样，由于结果栈中的序列不能被成功分析，所以也不会发生任何事情，如果分析能成功，那么由拼错的关键字所表示的项目也不会被包含进来。

也可以考虑由单选按钮、命令按钮和列表框组成的用户界面自动生成一种语言，尽管这样的界面似乎排除了语言的必要性，但语言对排序和计算还是适用的。当需要用一种方法指定操作顺序时，语言是完成这种工作的一种好方法，即使它由用户界面生成。

解释器模式具有下列优点：一旦构建好通用的分析、归约工具，扩展或修改文法相当容易；规则建立后也可以添加新的动词或变量。然而，随着文法变得越来越复杂，读者也在冒着创建的程序难以维护的风险。

解释器在解决一般程序设计问题时不那么通用，接下来学习的迭代器（Iterator）模式才是以后最常用的一种模式。

23.9　思考题

设计一个系统，让它计算二次方程式的结果，比如：

```
4x ^ 2 + 3x – 4
```

用户可以输入 x 或 x 的范围，也可以输入方程式。

23.10 随书附带光盘中的程序

C# Interpreter	`\Interpreter`

第24章　迭代器模式

迭代器模式（Iterator Pattern）是一种最简单、最常用的设计模式。迭代器模式允许使用一个标准的接口顺序访问一个数据列表或集合，而又不需知道数据的内部表示细节。另外，也可以定义专用迭代器，让它完成一些特殊操作并且只返回数据集合中的特定元素。

24.1　动机

迭代器是很有用的，因为它以一种预先定义的方式顺序访问一系列数据元素，又不需要暴露类内部的行为。由于迭代器是一个接口，可以采用任何便于返回数据的方式实现它。“Design Patterns”中指出，一个恰当的迭代器接口可以是下列形式：

```
public interface Iterator   {
      object First();
      object Next();
      bool isDone();
      object currentItem();
}
```

通过该接口，可以将当前元素指针移列表的第一个元素，顺序访问列表，判断出列表是否还有元素，以及找出列表的当前元素。该接口易于实现，且具有某些优点，也可以使用其他类似的接口。例如，在我们讨论组合（Composite）模式时，引入的 getSubordinates 方法：

```
IEnumerator getSubordinates();      //get subordinates
```

它提供了一个循环遍历一名雇员所有下属的方法。IEnumerator 接口可以用 C# 表示为：

```
bool MoveNext();
void Reset();
object Current{get;}
```

该接口也能循环访问具有某种内部列表结构的、拥有零个或多个元素的列表，而又不需要知道该列表在类内部是如何组织的。

针对相似结构的枚举法可使用C++和Smalltalk语言实现，但对C#则是个劣势，因为C#是一种强类型语言，Current( )属性不能返回与集合中实际的数据类型相一致的对象，必须把返回对象的类型转换成集合中的数据类型。所以，要把 IEnumerator 接口变为多态的，用 C# 不能直接实现。

24.2　迭代器示例代码

我们再次使用以前描述的由游泳选手、俱乐部和时间组成的列表，并向 KidData 类添加一些列举功能，该类实际上是 Kid 对象的一个集合，每个 Kid 对象都具有名字、俱乐部和时间等数据域，这些 Kid 对象保存在一个 ArrayList 中。

```
public class KidData :IEnumerator {
      private ArrayList kids;
      private int index;
```

```
public KidData(string filename)    {
        kids = new ArrayList();
        csFile fl = new csFile(filename);
        fl.openForRead();
        string line = fl.readLine();
        while(line != null) {
              kid kd = new kid(line);
              kids.Add(kd);
              line = fl.readLine();
        }
        fl.close();
        index = 0;
 }
```

为了得到集合中所有Kid对象的枚举量，使用刚才定义的IEnumerator接口中的方法：

```
public bool MoveNext(){
      index++;
      return index < kids.Count;
      }
      //------
      public object Current{
            get {
                    return kids[ index] ;
            }
      }
      //------
      public void Reset(){
            index = 0;
      }
```

读入数据并显示一个名字列表非常容易。用文件名初始化KidData类，让它构建Kid对象的集合，然后，把Kid类看做IEnumerator的一个实例（即实现了该接口的迭代器），顺序访问对象集合，取出各对象并显示他们的名字。

```
private void inti(){
  kids = new KidData("50free.txt");
  while(kids.MoveNext()){
    Kid kd = (Kid)kids.current;
    lsKids.Items.Add(kd.getFrname()+" "+kd.getLname());
  }
}
```

24.2.1 取出一个迭代器

在类中处理迭代器的另外一种稍灵活的方法是，提供一个带有getIterator方法的类，该方法为类中的数据返回迭代器的实例。因为对同样的数据，可以同时运行多个迭代器，所以这种方法更灵活一些。创建一个KidIterator类，它实现了迭代器接口。

```
public class KidIterator :IEnumerator {
      private ArrayList kids;
      private int index;
      public KidIterator(ArrayList kidz) {
            kids = kidz;
            index = 0;
      }
      //------
```

```
        public bool MoveNext(){
                index++;
                return index < kids.Count;
        }
        //------
        public object Current {
                get {
                        return kids[ index] ;
                        }
        }
        //------
        public void Reset() {
                index = 0;
        }
}
```

可根据需要创建迭代器，并从 KidData 类取回来。

```
public KidIterator getIterator() {
       return new KidIterator(kids);
}
```

24.3　过滤迭代器

尽管有一个定义简洁、能顺序访问一个集合的方法是很有帮助的，但还可以定义过滤迭代器（Filtered Iterator），它在返回数据前会对数据执行一些计算。例如，可以返回按特定方式排序的数据，或只返回匹配特定判别式的那些对象。此时，不用为这些过滤迭代器设计许多个类似的接口，只需提供一个能返回每种类型迭代器的方法，而这些迭代器都要具有相同的方法。

24.3.1　一个过滤迭代器

假设只想列举出属于某一特定俱乐部的选手，这就有必要设计一个专门的迭代器类去访问 KidData 类中的数据。这很简单，因为刚才定义的方法就能做到这一点。接下来，只需编写一个具体的迭代器，它只返回属于一个特定俱乐部的选手。

```
public class FilteredIterator : IEnumerator {
      private ArrayList kids;
      private int index;
      private string club;
      public FilteredIterator(ArrayList kidz,string club){
              kids = kidz;
              index = 0;
              this.club = club;
      }
      //------
      public bool MoveNext() {
             bool more = index<kids.Count-1;
             if(more) {
                  Kid kd = (Kid)kids [ ++index] ;
                  more = index < kids.Count;
                  while(more && ! kd.getClub().Equals(club)){
                        kd = (Kid)kids[ index++] ;
                        more = index < kids.Count;
                  }
```

```
            }
            return more;
        }
        //------
        public object current {
            get {
                return kids[ index] ;
            }
        }
        //------
        public void Reset() {
            index = 0;
        }
}
```

所有的工作都在MoveNext( )方法中完成，它扫描数据集合，寻找属于构造函数中指定的那个俱乐部的选手，然后返回 true 或 false。

最后，还需要向 KidData 添加一个方法，它返回这一新的过滤迭代器。

```
public FilteredIterator getFilteredIterator(string club)  {
     return new FilteredIterator(kids,club);
}
```

该方法将数据集合和俱乐部名称的首字母传递给新的迭代器类 FilteredIterator。图 24.1 给出了这一简单程序的运行结果，左边显示的是所有选手。该程序将一个俱乐部列表填到组合框中，用户选择一个俱乐部后，属于该俱乐部的那些选手会填到右边的列表框中。类图如图 24.2 所示。注意，KidData 中的方法提供了一个迭代，而 kidClub 类本身实际上就是一个迭代器类。

图 24.1 一个说明过滤迭代的简单程序

24.4 记录俱乐部

我们需要得到一个俱乐部列表，填到图 24.1 中的组合框中。在读入每名小选手时，将俱乐部名放到一个哈希表（Hashtable）中，这样就能完成上述工作。

```
while(line != null){
      Kid kd = new Kid(line);
      string club = kd.getClub();
      if(! clubs.Contains(club)){
```

```
            clubs.Add(club,club);
        }
        kids.Add(kd);
        line = fl.readLine();
    }
```

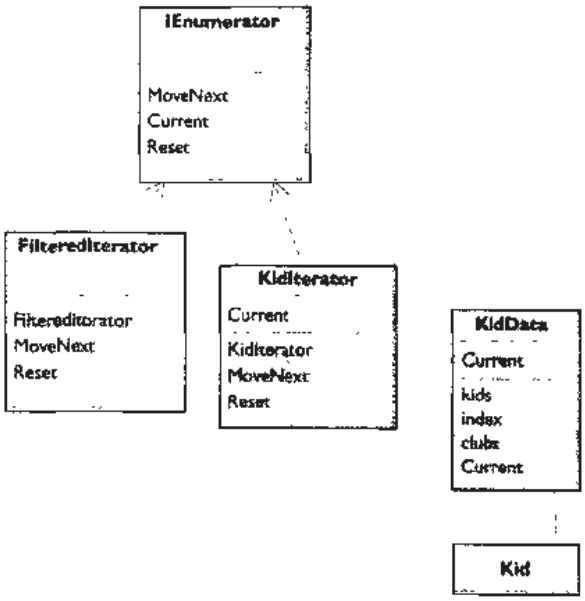


图 24.2　用在过滤迭代中的所有类

接下来，当需要俱乐部列表时，可以让哈希表给出其内容。Hashtable 类有一个方法 getEnumerator，它应该能返回需要的信息。但是，该方法返回的是一个 IdictionaryEnumerator 对象，与我们的目标稍有些差别。由于 IdictionaryEnumerato 派生于 IEnumerator，可以使用它的 Value 方法返回哈希表中的内容。这样，就可以用下列代码填充组合框。

```
IDictionaryEnumerator clubiter = kdata.getClubs();
while(clubiter.MoveNext())   {
      cbClubs.Items.Add((string)clubiter.Value);
}
```

单击组合框，会得到一个选中的俱乐部，生成一个过滤迭代器并填充列表框。

```
private void cbClubs_SelectedIndexChanged(object sender,
                                EventArgs e){
      string club = (string)cbClubs.SelectedItem;
      FilteredIterator iter = kdata.getFilteredIterator(club);
      lsClubKids.Iterms.Clear();
      while(iter.MoveNext()){
        Kid kd = (Kid)iter.current;
        lsClubKids.Items.Add(kd.getFrname()+ "  "+
             kd.getLname());
      }
}
```

24.5　迭代器模式的效果

1. **数据修改。**迭代器中最重要的问题是，遍历数据的同时数据正在被修改。如果读者的代码搜索范围较宽，只是偶尔才移向下一个元素，就有可能在访问集合的同时，底层集合正在添加或删除元素，也可能有另一个线程改变了集合。对这一问题没有简单的解决方案。如果读者

想使用迭代器顺序访问一系列数据并删除特定的项目，要当心这种操作的后果。添加或删除一个元素意味着可能会遗漏掉某一元素或对同一元素访问了两次，这和读者所用的存储机制有关。

2. **特权访问。**迭代器类需要对基本容器类的底层数据结构具有特殊访问权限，这样才能访问数据。如果数据存储在一个 ArrayList 或 Hashtable 中，这很容易做到，但如果是存储在一个类里的其他集合结构里，就只能通过 get 操作使用该结构。另一种方法是，让迭代器成为容器类的一个派生类，直接访问其中的数据。
3. **外部迭代器与内部迭代器。**“Design Patterns”中描述了两种迭代器：外部的和内部的。迄今为止，只介绍了外部迭代器。内部迭代器是能顺序访问整个集合的一些方法，不需要用户发出任何明确的请求就能直接对每个元素进行操作。这在 C# 中比较少见，但可以将这些方法设想成，规范一个集合中的数据值，使之处于 0 到 1 的范围，或者将字符串中的所有字符转换成特定格式（大写或小写）。总的来说，外部迭代器给读者更多的控制权，因为调用程序可直接访问每个元素，并能决定是否对元素执行操作。

24.6 随书附带光盘中的程序

Kid list using Iterator	`\Iterator\SimpleIterator`
Filtered iterator by team name	`\Iterator\FilteredIterator`

第25章　中介者模式

当一个程序由多个类组成时，类之间是按逻辑与算法划分的。然而，程序中使用的孤立类越多，类之间的通信问题就越复杂。每个类对其他类中的方法了解得越多，类结构就会变得越混乱。这会使程序难以阅读、难以维护，而且修改程序也变得困难，因为任何修改都会影响其他几个类的代码。中介者模式（Mediator Pattern）通过促进类之间的松散耦合解决了这一问题。中介者作为惟一知道其他类中方法细节的一个类，在类发生变化时通知中介者，中介者再将这些变化传递给其他需要通知的类。

25.1　一个例子

我们考虑这样一个程序，它有两个按钮、两个列表框和一个文本输入域，如图25.1所示。程序开始运行时，Copy和Clear按钮都是不可用的。

1. 当读者选择左边列表框中的一个名字时，它被拷贝到文本域中进行编辑，此时Copy按钮可用。
2. 单击Copy按钮，文本被加到右边列表框中，此时Clear按钮可用，如图25.2所示。
3. 如果单击Clear按钮，右边列表框和文本域被清空，列表框是未选中状态，两个按钮又都不可用。

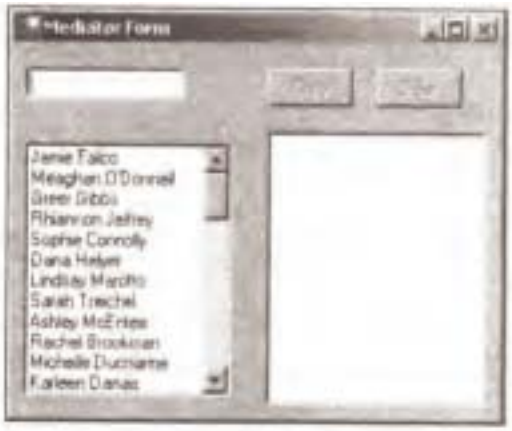


图25.1　一个简单的程序，两个列表、两个按钮和一个文本域相互协作

图25.2　选择一个名字后，按钮变为可用，单击Copy按钮，该名字拷贝到右边的列表框中

像这样的用户界面通常用于在很长的人员列表或产品列表中选择子表，而且通常都要比这个例子复杂得多，还要包括插入、删除和取消等操作。

25.2 控件间的相互协作

即使在这个简单的例子里，可视控件之间的相互协作也很复杂。每个可视对象都要了解其他两个或更多的对象，于是形成了一个相当混乱的关系图，如图 25.3 所示。

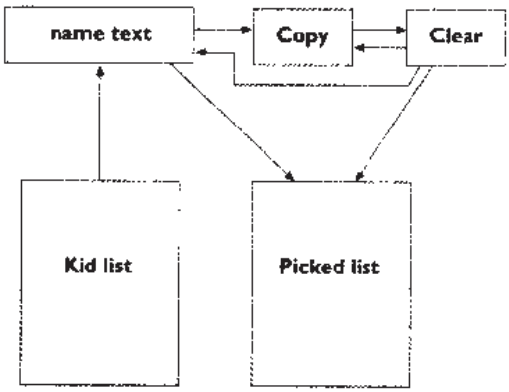


图 25.3 图 25.1 和图 25.2 的可视界面中的类组成了一个混乱的相互协作网

中介者模式可以简化该系统，它让中介者作为惟一了解系统其他类的一个类。每个和中介者通信的控件都称为同事（Colleague）。每个同事收到用户事件时通知中介者，中介者决定应该将该事件通知给其他哪一个类，这种比较简单的相互协作机制可用图 25.4 说明。

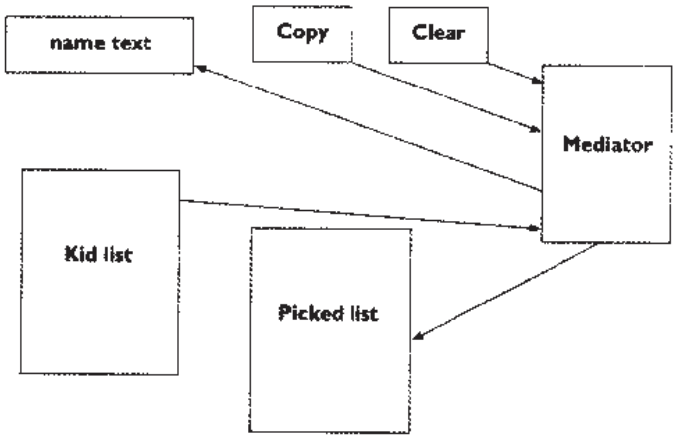


图 25.4 一个 Mediator 类简化了类之间的相互协作

中介者的优点很明显：因为它是惟一了解其他类的类，所以，如果其他某个类发生改变，或者添加了其他接口控件类，它是惟一需要改变的类。

25.3 示例代码

详细考虑这个程序，然后再决定怎样构建每个控件。使用 Mediator 类编写程序的主要差别是，每个类都要知道 Mediator 的存在。可以先创建一个 Mediator 类的实例，然后将该 Mediator 实例传递给每个类。

```
med = new Mediator (btCopy, btClear, lsKids, lsSelected);
btCopy.setMediator (med);  //set mediator ref in each control
btClear.setMediator (med);
lsKids.setMediator (med);
med.setText (txName);      //tell mediator about text box
```

从Button类派生出两个自己的按钮类，这样它们就也能实现Command接口，这些按钮被传递到Mediator的构造函数里。下面是CpyButton类。

```
public class CpyButton : System.Windows.Forms.Button, Command   {
       private Container components = null;
       private Mediator med;
       //-----
       public CpyButton()        {
              InitializeComponent();
       }
       //-----
       public void setMediator(Mediator md) {
              med = md;
       }
       //-----
       public void Execute() {
              med.copyClicked ();
       }
```

该类的Execute方法只是告诉Mediator类“它被单击”，让Mediator决定发生该事件时应该做什么。Clear按钮与此完全类似。

从ListBox类派生出KidList类，在Mediator的构造函数里，将名字加到列表框中。

```
public Mediator(CpyButton cp, ClrButton clr, KidList kl,
                          ListBox pk)        {
       cpButton = cp;                        //copy in buttons
       clrButton = clr;
       klist = kl;                           //copy in list boxes
       pkList = pk;
       kds = new KidData ("50free.txt");     //create data list class
       clearClicked();                       //clear all controls
       KidIterator kiter = kds.getIterator ();
       while(kiter.MoveNext () ) {           //load list box
              Kid kd = (Kid) kiter.Current;
              klist.Items.Add (kd.getFrname() +" "+
                           kd.getLname ());
       }
}
```

不必专门对文本域做任何事情，因为它的操作都发生在Mediator内，正如刚才说明的，只需用setText方法把它传递到Mediator就可以了。

另一个重要的初始化内容是为两个按钮和列表框创建一个事件处理程序。我们不是通过开发环境生成这些单击事件，而是自己创建一个事件，并将它添加到两个按钮的单击处理程序中和列表框的SelectIndexChanged事件中，该事件处理程序的巧妙之处在于：它只调用每个控件的Execute方法，让Execute方法去调用Mediator的方法来完成所有的实际工作。

这些单击事件的处理程序就是下列形式：

```
//each control is a command object
public void clickHandler(object obj, EventArgs e)  {
     Command comd = (Command)obj;        //get command object
```

```
        comd.Execute();                    //and execute command
    }
```

下面给出完整的窗体初始化方法，它创建了事件处理句柄。

```
    private void init() {
           //set up mediator and pass in referencs to controls
           med = new Mediator (btCopy, btClear, lsKids, lsSelected);
           btCopy.setMediator (med); //mediator ref in each control
           btClear.setMediator (med);
           lsKids.setMediator (med);
           med.setText (txName);      //tell mediator about text box

           //create event handler for all command objects
           EventHandler evh = new EventHandler (clickHandler);
           btClear.Click += evh;
           btCopy.Click += evh;
           lsKids.SelectedIndexChanged += evh;
    }
```

这些类的共同点是，每个类都知道Mediator并告诉Mediator它自己的存在，这样，Mediator就可以在适当的时候向它们发送命令。

Mediator本身非常简单，它支持Copy，Clear和Select方法，还有针对TextBox的一个登录（register）方法。两个按钮和一个列表框都传递给Mediator的构造函数。注意，选择setXxx方法而不用构造函数参数传入这些控件的引用，并没有什么真正的原因。在这个例子中使用了这两种方法。

```
    public class Mediator {
          private CpyButton cpButton;              //buttons
          private ClrButton clrButton;
          private TextBox txKids;                  //text box
          private ListBox pkList;                  //list boxes
          private KidList klist;
          private KidData kds;                     //list of data from file

          public Mediator(CpyButton cp, ClrButton clr,
                 KidList kl, ListBox pk)       {
                 cpButton = cp;                    //copy in buttons
                 clrButton = clr;
                 klist = kl;                      //copy in list boxes
                 pkList = pk;
                 kds = new KidData ("50free.txt");//create data list class
                 clearClicked();       //clear all controls
                 KidIterator kiter = kds.getIterator ();
                 while(kiter.MoveNext () ) {       //load list box
                        Kid kd = (Kid) kiter.Current ;
                        klist.Items .Add (kd.getFrname() +
                               " "+kd.getLname ());
                 }
          }
          //-----
          //get text box reference
                 public void setText(TextBox tx) {
                        txKids = tx;
                 }
          //-----
```

```
        //clear lists and set buttons to disabled
        public void clearClicked() {
                //disable buttons and clear list
                cpButton.Enabled = false;
                clrButton.Enabled = false;
                pkList.Items.Clear();
        }
        //-----
        //copy data from text box to list box
        public void copyClicked() {
                //copy name to picked list
                pkList.Items.Add(txKids.Text);
                //clear button enabled
                clrButton.Enabled = true;
                klist.SelectedIndex = -1;
        }
        //-----
        //copy selected kid to text box
        //enable copy button
        public void kidPicked() {
                //copy text from list to textbox
                txK    ids.Text = klist.Text;
                //copy button enabled
                cpButton.Enabled = true;
        }
}
```

25.3.1　系统的初始化

另一个操作最好也由Mediator代理：就是将所有的控件初始化为预期的状态。启动程序时，每个控件都必须处于一个已知的默认状态，由于这些状态会随着程序的运行而发生改变，所以，只在Mediator的构造函数中执行初始化操作，将所有的控件置为预期的状态。在本例中，初始状态就是Clear按钮对应的状态，只调用下面的方法就能完成初始化操作。

```
clearClicked();      //clear all controls
```

25.4　Mediator对象与Command对象

本程序中的两个按钮都使用了命令对象。正如先前提到的，这会使按钮单击事件的处理变得相当简单。

在命令模式那一章中提到一个问题：每个按钮为了执行自己的命令，需要知道许多其他用户接口类的信息。Mediator是该问题的一个解决方案。令那些信息由Mediator代理，这样，Command按钮就不需要了解其他可视对象中的方法内容。本程序的类图如图25.5所示，它既说明了中介者模式，也说明了命令模式的应用。

25.5　中介者模式的效果

1. 当一个类的动作要受另一个类的状态影响时，中介者模式可以防止这些类变得混乱。
2. 使用中介者很容易修改一个程序的行为。对多数形式的修改来说，只需改变或子类化中介者，程序的其他部分保持不变。
3. 可以添加新的控件或类，除了改变中介者外，不用改动任何部分。

4. 中介者解决了每个Command对象需要对用户接口中的其他对象和方法了解太多的问题。
5. 中介者变成了一个“万能类”，知道太多其他部分程序的信息，这会使它难以修改和维护。有时候可以改善这种状态：将大部分的功能放在各个类中，少部分放在中介者中。每个对象应该完成自己的任务，中介者只需管理对象之间的相互协作。
6. 每个中介者都是一个定制的类，它拥有每个同事要调用的方法，并知道每个同事有哪些可用的方法。这使得在不同项目中重用中介者变得很困难。从另一方面来说，绝大多数中介者都相当简单，编写中介者代码总要比管理复杂的相互协作容易得多。

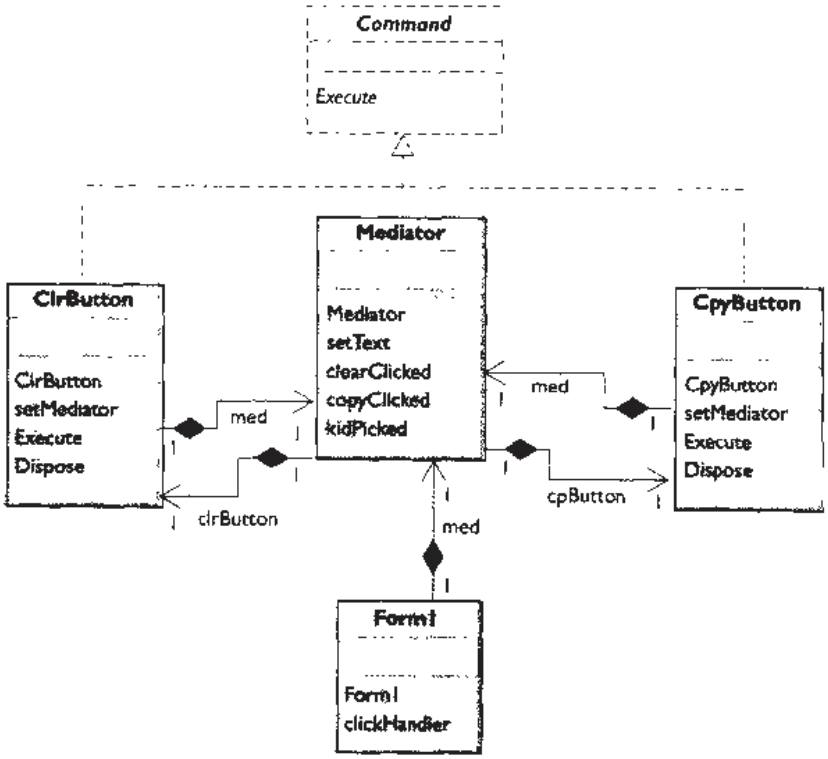


图 25.5 Command对象和Mediator对象间的协作

25.6 单接口的中介者

这里描述的中介者模式相当于一种观察者（Observer）模式，观察每个同事元素的变化。每个元素都对中介者有一个定制的接口。另外一种实现方法是，中介者只有一个接口，通知中介者执行哪种操作的对象都传递给该接口方法。

在这种方法中，我们不用登记运行的(active)组件，只需创建一个action方法，它针对每个动作对象具有不同的多态参数。

```
public void action (MoveButton mv );
public void action (clrButton clr );
public void action (KidList klist );
```

这样，就不必登记所有的动作对象，如按钮和源列表框，因为可以将其作为参数传递给action方法。

按照同样的方法，可以只有一个Colleague接口，让每个Colleague都实现它，这样，每个Colleague都可以决定它要执行什么操作。

25.7　实现问题

中介者模式不只限于应用在可视界面程序中，但这的确是它最常见的应用。当面临多个对象之间的复杂通信问题时，可以使用中介者模式。

25.8　随书附带光盘中的程序

Mediator	\Mediator

第26章　备忘录模式

本章中，我们讨论如何使用备忘录模式（Memento Pattern）保存对象的数据以便以后能够恢复它。例如，若想要保存一个绘图程序中的对象颜色、尺寸、模式或形状，理想情况是，不需要对象负责具体的任务，也不破坏封装，就能保存和恢复该状态。这就是备忘录模式的目的。

26.1　动机

对象通常不应该用公有方法暴露太多的内部状态，但是因为以后可能需要恢复对象，还是希望能保存它的整个状态。在某些情况下，可以从公有接口（例如，图形对象的绘图位置）获得足够的信息来保存和恢复数据，而在另外一些情况下，需要保存颜色、明暗、角度以及与其他对象的连接关系，这些信息不容易获得。这种需要保存和恢复的信息在支持Undo命令的系统中是很常见的。

如果在公有变量里可以得到一个对象的所有描述信息，那么将这些信息以某种外部形式存储起来并不困难。然而，使这些数据公有化会使整个系统易受外部代码的改变而破坏，而通常都希望对象内部的数据是私有的，与外部世界相隔离。

备忘录模式可以通过特权访问要保存对象的状态，采用某些语言解决这一问题。其他对象对该对象只具有比较受限的访问权限，从而保持了它们的封装特性。然而，在C#中，关于特权访问的内容并不多，我们在例子里会用到。

该模式为对象定义了三个角色。

1. 发起人（Originator）是一个对象，我们要保存它的状态。
2. 备忘录（Memento）是另外一个对象，它保存了发起人的状态。
3. 负责人（Caretaker）管理状态保存的时机，保存备忘录，并且如果需要的话，使用备忘录恢复发起人的状态。

26.2　实现

保存一个对象的状态而又不让它的全部变量都公开是比较棘手的，不同语言对其实现的程度也不同。“Design Patterns”建议使用C++的友元结构来实现这种特权访问，“Smalltalk Companion”中提到，Smalltalk语言不能直接实现这种访问。在Java中，保护模式的包可具有特权访问。C#中可以使用internal关键字，其含义是任何标记了internal的类方法只能在该项目内访问。如果由这样的类组成一个库，标记internal的方法不能导出使用。我们换种方法，定义一个用于取出和存储重要内部数据的属性，不再为其他目的而使用该类中的其他属性。为了保持一致性，我们对这些属性使用internal关键字，但是记住，这种对internal的使用在语言上没有严格限制。

26.3　示例代码

考虑一个绘图程序的简单原型，它可以绘制矩形，选择矩形，并能用鼠标拖动它四处移动。该程序拥有一个工具栏，包括三个按钮：Rectangle，Undo和Clear，如图26.1所示。

Rectangle 按钮是一个工具栏 ToggleButton，它一直是选中状态，直到单击鼠标画了一个新矩形。画完矩形后，可以单击矩形内部选择它，如图 26.2 所示。

选择一个矩形后，可以用鼠标拖动它到一个新的位置，如图 26.3 所示。

图 26.1　一个简单的图形绘制程序，能绘制矩形、取消绘图和清除屏幕

图 26.2　选择一个矩形会显示出“处理”形状，表示该矩形被选中并可以移动

图 26.3　拖动被选中的矩形

Undo按钮可以取消一系列操作，特别是能取消一个矩形的移动，还能取消矩形的创建。下面是程序必须响应的五个动作。

1. 单击 Rectangle 按钮。
2. 单击 Undo 按钮。
3. 单击 Clear 按钮。
4. 单击鼠标。
5. 拖动鼠标。

三个按钮都可构建成Command对象，也可以把鼠标单击和拖动看做命令。因为我们有多个可视化对象控制屏幕对象显示，这种情况适合于使用中介者模式，实际上那也正是我们构建本程序的途径。

这里要创建一个Caretaker类来管理Undo动作列表，它保存了前n个操作，便于以后取消它们。Mediator类保存了绘图对象的列表，它还要和Caretaker对象通信。事实上，由于这类程序中会有多个要保存和取消的操作，所以实际上也需要中介者，只有它可以发送命令给Caretaker中的Undo列表。

本程序中，只保存和取消两个操作：创建新矩形和改变矩形位置。先创建VisRectangle类，每一个矩形实例都由它绘制。

```
public class VisRectangle  {
       private int x, y, w, h;
       private const int SIZE=30;
       private CsharpPats.Rectangle rect;
       private bool selected;
       private Pen bPen;
       private SolidBrush bBrush;
       //-----
       public VisRectangle(int xp, int yp)       {
              x = xp;                          y = yp;
              w = SIZE;                        h = SIZE;
              saveAsRect();
              bPen = new Pen(Color.Black);
              bBrush = new SolidBrush(Color.Black);
       }
       //-----
       //used by Memento for saving and restoring state
       internal CsharpPats.Rectangle rects {
              get {
                     return rect;
              }
              set {
                     x=value.x;
                     y=value.y;
                     w=value.w;
                     h=value.h;
                     saveAsRect();
            }
       }
       //------
       public void setSelected(bool b) {
              selected = b;
       }
       //-----
       //move to new position
```

```
public void move(int xp, int yp) {
        x = xp;
        y = yp;
        saveAsRect();
}
//-----
public void draw(Graphics g) {
        //draw rectangle
        g.DrawRectangle(bPen, x, y, w, h);

if (selected) {    //draw handles
g.FillRectangle(bBrush, x + w / 2, y - 2, ,4);
g.FillRectangle(bBrush, x - 2, y + h / 2, 4, 4);
g.FillRectangle(bBrush, x + (w / 2), y + h - 2, 4, );
g.FillRectangle(bBrush, x + (w - 2),
                    y + (h / 2), 4, 4);
}
}
//-----
//return whether point is inside rectangle
public bool contains(int x, int y) {
       return rect.contains (x, y);
}
//------
//create Rectangle object from new position
private void saveAsRect() {
        rect = new CsharpPats.Rectangle (x,y,w,h);
}
```

还用到了前面开发的Rectangle类，它包含了针对数据域x、y、w、h的Get和Set属性以及一个contains方法。

绘制矩形相当简单。接下来创建Memento类，它很简单，我们用它保存一个矩形的状态。

```
public class Memento  {
      private int x, y, w, h;
      private CsharpPats.Rectangle rect;
      private VisRectangle visRect;
      //------
      public Memento(VisRectangle vrect)        {
            visRect = vrect;
            rect = visRect.rects;
            x = rect.x;
            y = rect.y;
            w = rect.w;
            h = rect.h;
      }
      //------
      public void restore() {
            rect.x = x;
            rect.y = y;
            rect.h = h;
            rect.w = w;
            visRect.rects = rect;
      }
}
```

当创建一个Memento实例时，将需要保存的VisRectangle实例传递给它的初始化方法（即构造器）。Memento拷贝尺寸和位置参数，并保存了一个VisRectangle实例的副本。当我们以后需要恢复

这些参数时，Memento知道要恢复的数据对应了哪个实例，它可以直接进行恢复，在restore()方法里可以看到这一点。

其余的动作发生在Mediator类里，把图形列表的先前状态作为整数（即图形数）保存到取消列表里。

```
public void createRect(int x, int y) {
      unpick();          //make sure no rectangle is selected
      if (startRect) {   //if rect button is depressed
            int count = drawings.Count;
            caretaker.Add(count); //Save list size
            //create a rectangle
            VisRectangle v = new VisRectangle(x,y);
                        drawings.Add(v); //add element to list
            startRect = false;    //done with rectangle
            rect.setSelected(false);     //unclick button
            canvas.Refresh();
      }
      else
            //if not pressed look for rect to select
            pickRect(x, y);
      }
}
```

另外，如果在没选择Rectangle按钮时单击了面板，则认为你在试图选择一个已存在的矩形，下面是检测代码。

```
public void pickRect(int x, int y) {
      //save current selected rectangle
      //to avoid double save of undo
      int lastPick = -1;
      if (selectedIndex >= 0) {
            lastPick = selectedIndex;
      }
      unpick();  //undo any selection
      //see if one is being selected
      for (int i = 0; i< drawings.Count; i++) {
            VisRectangle v = (VisRectangle)drawings[ i];
            if (v.contains(x, y)) {
                  //did click inside a rectangle
                  selectedIndex = i;      //save it
                  rectSelected = true;
                  if (selectedIndex != lastPick) {
                        //but don't save twice
                        caretakr.rememberPosition(v);
                  }
                  v.setSelected(true);     //turn on handles
                  repaint();           //and redraw
            }
      }
}
```

Caretaker类将矩形的原始位置记录在一个Memento对象里，并将该对象添加到取消列表中。

```
public void rememberPosition (VisRectangle vr )  {
       Memento mem = new Memento (vr);
       undoList.Add (mem);
}
```

Caretaker 类管理取消列表，该表是整数对象和 Memento 对象的集合。如果取消列表的某个元素是一个整数，它表示在那一时刻绘制的图形数目；如果是一个 Memento 对象，它表示一个要被恢复的 VisRectangle 实例的初始状态。换句话说，该取消列表能取消添加的新矩形和已存在矩形的移动。

这里的 Undo 方法只是决定，是从绘图列表中减去一个元素还是调用 Memento 的 restore 方法。由于取消列表包含了整数对象和 Memento 对象，所以先把某个列表元素转换成 Memento 类型，如果转换失败，则捕获该异常并能认识到它是一个要被删除的图形列表元素。

```
public void undo() {
      if(undoList.Count > 0) {
            int last = undoList.Count -1;
            object obj = undoList[ last];
            try{
                  Memento mem = (Memento)obj;
                  remove(mem);
            }
            catch (Exception) {
                  removeDrawing();
            }
            undoList.RemoveAt (last);
      }
}
```

两个删除方法中，一个是让图形的数目减 1，一个是恢复矩形的位置。

```
public void removeDrawing()  {
       drawing.RemoveAt (drawings.Count -1 );
}
public void remove (Memento mem)  {
      mem.restore ();
}
```

26.3.1　重要提示

尽管在本例中，提取出表示矩形位置的 Memento 对象和表示添加新图形的整数对象两者之间的差别很有用，但是按通常意义讲，这绝对是一个很糟糕的 OO 程序设计例子。不应该通过检查对象的类型来决定对它的操作；相反，应该能针对对象调用正确的方法，让它完成正确的事情。

另一种更好的编写该示例代码的方法是，让图形元素和 Memento 类拥有各自的恢复方法，并让它们都作为一个通用 Memento 类（或接口）的成员。我们在第 28 章“状态模式”的例子中会采用这种方法。

26.4　用户界面中的命令对象

也可以用命令模式帮助简化用户界面代码。可以在 C# 中通过 IDE 构建一个工具栏并创建 ToolbarButtons，但是，如果采用这种方法，很难子类化它们使之成为命令对象。有两种解决方案：第一种，为 RectButton，UndoButton 和 ClearButton 保留一个并行的命令对象数组，在工具栏的单击例程里调用它们。

然而，读者应该知道，工具栏按钮没有 Index 属性，不能通过索引来确定哪一个对象被单击，再将其关联到相应的命令数组上。但是，可以用每个工具按钮的 GetHashCode 属性得到按钮的惟一

标识，并将对应的命令对象保存到一个以按钮哈希码为关键字的哈希表（Hashtable）中。构造哈希表的代码如下所示：

```
private void init() {
       med = new Mediator(pic);      //create Mediator
       commands = new Hashtable();  //and Hash table
       //create the command objectsb
       RectButton rbutn = new RectButton(med, tbar.Buttons[ 0] );
       UndoButton ubutn = new UndoButton(med, tbar.Buttons[ 1] );
       ClrButton clrbutn = new ClrButton(med);
       med.registerRectButton (rbutn);
       //add them to the hashtable using the button hash values
       commands.Add(btRect.GetHashCode(), rbutn);
       commands.Add(btUndo.GetHashCode(), ubutn);
       commands.Add(btClear.GetHashCode(), clrbutn);
       pic.Paint += new PaintEventHandler (paintHandler);
}
```

接下来的命令解释工作只需要几行代码，因为所有的按钮都已经调用了相同的单击事件程序。单击按钮时，可以用这些哈希码取出正确的命令对象。

```
private void tbar_ButtonClick(object sender,
                    ToolBarButtonClickEventArgs e) {
       ToolBarButton tbutn = e.Button;
       Command comd = (Command)commands[ tbutn.GetHashCode ()] ;
       comd.Execute ();
}
```

另一种方法是，可以在IDE控制下创建工具栏，但要通过编程添加工具按钮，还要用到实现Command接口的派生按钮类。在状态模式那一章中说明这种方法。

RectButton类是发生大部分操作的地方。

```
public class RectButton :  Command    {
       private ToolBarButton bt;
       private Mediator med;
       //------
       public RectButton(Mediator md, ToolBarButton tb)     {
              med = md;
              bt = tb;
       }
       //------
       public void setSelected(bool sel) {
              bt.Pushed  = sel;
       }
       //------
       public void Execute() {
              if(bt.Pushed  )
                     med.startRectangle ();
       }
 }
```

26.5 处理鼠标事件和Paint事件

我们还需要捕获鼠标按下、松开和移动事件，并将其传递给Mediator去处理。

```
private void pic_MouseDown(object sender, MouseEventArgs e) {
      mouse_down = true;
```

```
        med.createRect (e.X, e.Y);
}
//------
private void pic_MouseUp(object sender, MouseEventArgs e) {
        mouse_down = false;
}
//------
private void pic_MouseMove(object sender, MouseEventArgs e) {
        if(mouse_down)
                med.drag(e.X, e.Y);
}
```

任何时候Mediator做出改变，都要求刷新图片框，接下来，图片框的刷新会调用Paint事件，在Paint事件处理程序中再回头调用Mediator的方法，在新的位置画出矩形。

```
private void paintHandler(object sender, PaintEventArgs e)  {
        Graphics g = e.Graphics;
        med.reDraw (g);
}
```

图26.4给出了完整的类结构图。

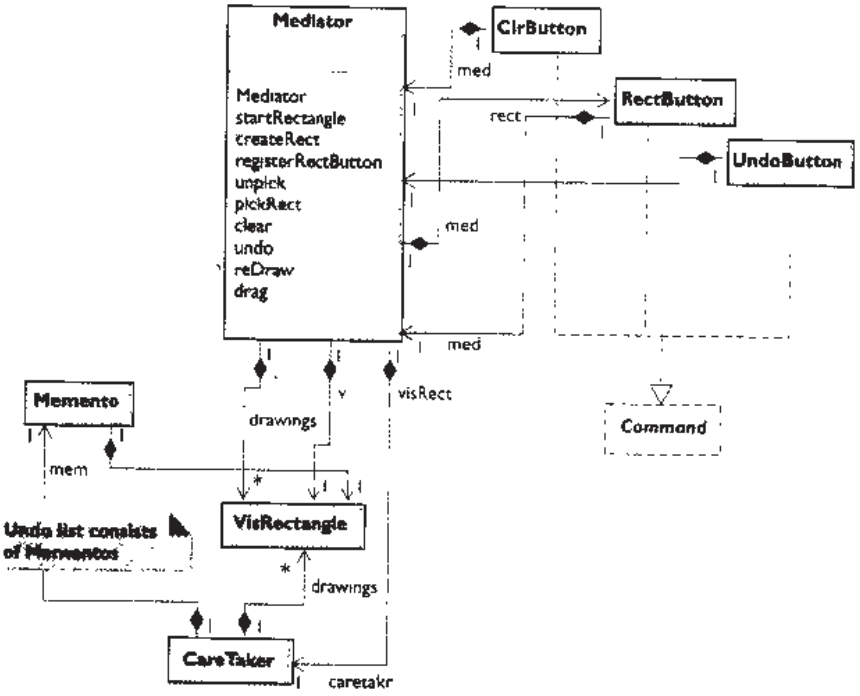


图 26.4　使用备忘录模式的绘图程序的 UML 图

26.6　备忘录模式的效果

备忘录提供了一种保存一个对象状态的方法，并保持了语言的封装特性。只能访问Originator类中的数据，有效地保持了数据的私有性。把信息的保存和恢复指派给Memento类，从而简化了Originator类。

另一方面，Memento需要保存的信息量可能非常庞大，要占用相当多的存储空间。这会进一步影响Caretaker类，需要设计一些策略来限制需要保存状态的对象数量。在这个简单的例子里，我

们没有给出这样的限制。在有些情况下，对象是以一种可预测的方式改变，每个 Memento 应该能只保存一个对象状态的相对变化部分。

在本章的示例代码里，不仅用到了备忘录模式，还使用了命令模式和中介者模式。将几种模式组合在一起使用是很常见的，对优秀的 OO 程序研究得越多，会发现模式组合使用的情况就越多。

26.7 思考题

备忘录也可用于处理过程失败时恢复对象的状态。如果由于网络连接断掉而使数据库更新失败，应该能将缓冲区中的数据恢复到初始状态。重新编写外观模式一章中的 Database 类，使之能处理这种失败情况。

26.8 随书附带光盘中的程序

Memento Example	\Memento

第27章　观察者模式

本章中，我们讨论如何用观察者模式（Observer Pattern）同时以几种形式表示数据。在新奇而复杂的窗口世界里，经常希望能同时以多种形式显示数据，并且所有的显示都能反映出数据的变化。例如，用图形、表格或列表框表示股票价格的变化，每次价格发生变化时，所有的表示形式应该能同时改变，而不需要做任何操作。

我们期望实现这样的行为是因为，在许多Windows应用程序中（如Excel）都能看到它。Windows目前还没有提供实现这种操作的内部细节，而且正如读者所知道的，使用C或C++直接进行Windows程序设计相当复杂。然而在C#中，可以用观察者设计模式使程序轻易地实现这一行为。

观察者模式假定，包含数据的对象和显示数据的对象相隔离，这些显示对象要“观察”数据的变化，说明这一点很容易，如图27.1所示。

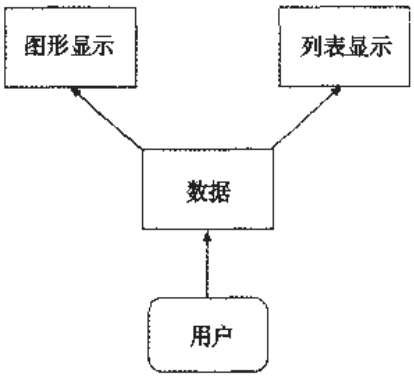


图27.1　用列表方式和图形方式显示数据

我们实观观察者模式时，通常把数据称为目标（Subject），把每种显示称为一个观察者（Observer）。每个观察者通过调用目标中的公有方法注册它所感兴趣的数据。每个观察者中都有一个已知的接口，目标在该数据发生变化时会调用它。将这些接口定义成如下形式：

```
public interface Observer  {
       void sendNotify(string message);
//--------
public interface Subject    {
       void registerInterest(Observer obs);
}
```

定义这些抽象接口的好处是：（1）可以写出多种类型的对象，只要它们都实现这些接口；（2）可以将这些对象声明为Subject类型和Observer类型，不管它们是做什么的。

27.1　观察颜色变化

现在编写一个简单程序来说明怎样使用这一有用的思想。我们的程序显示一个窗体，它包含三个单选按钮，分别为Red，Blue和Green，如图27.2所示。

图 27.2 一个能生成红色、绿色或蓝色“数据”的控制面板

主窗体类实现了Subject接口，也就是说，它提供了一个公有方法来注册该类中感兴趣的数据，该方法就是 registerInterest，它将 Observer 对象添加到一个 ArrayList 中。

```
public void registerInterest(Observer obs )   {
       observers.Add (obs);
}
```

接下来创建两个观察者：一个显示颜色（和颜色名），另一个将当前颜色添加到一个列表框中。每个观察者实际上都是 Windows 窗体，都实现了 Observer 接口。创建这些窗体实例时，将父窗体或启动窗体作为一个参数传递给它们。由于启动窗体实际上就是Subject，这两个观察者可以将它的事件作为感兴趣的数据注册。主窗体的初始化程序创建这些窗体实例，并将一个自身的引用传递给它们。

```
ListObs lobs = new ListObs (this);
lobs.Show();
ColObserver colObs = new ColObserver (this);
colObs.Show();
```

接下来在创建 ListObs 窗口时，在主程序中注册它感兴趣的数据。

```
public ListObs(Subject subj)          {
      InitializeComponent();
      init(subj);
}
//------
public void init(Subject subj) {
      subj.registerInterest (this);
}
```

当它收到一条来自目标（主窗体）的 sendNotify 消息时，所做的工作就是将颜色名加到列表中。

```
public void sendNotify (string message )  {
      lsColors.Items.Add(message);
}
```

这里的颜色窗口也是一个观察者，它需要改变图片框的背景颜色，并用一支画刷画出颜色名。注意，在sendNotify事件处理程序里改变图片框的背景颜色，在paint事件处理程序里改变文本。完整的类如下所示：

```
public class ColObserver : Form, Observer  {
      private Container components = null;
      private Brush bBrush;
```

```
private System.Windows.Forms.PictureBox pic;
private Font fnt;
private Hashtable colors;
private string colName;
//-----
public ColObserver(Subject subj)      {
       InitializeComponent();
       init(subj);
}
//-----
private void init(Subject subj) {
       subj.registerInterest (this);
       fnt = new Font("arial", 18, FontStyle.Bold);
       bBrush = new SolidBrush(Color.Black);
       pic.Paint+= new PaintEventHandler (paintHandler);
       //make Hashtable for converting color strings
       colors = new Hashtable ();
       colors.Add("red", Color.Red );
       colors.Add ("blue", Color.Blue );
       colors.Add ("green", Color.Green );
       colName = "";
}
//-----
public void sendNotify(string message) {
       colName = message;
       message = message.ToLower ();
       //convert color string to color object
       Color col = (Color)colors[ message] ;
       pic.BackColor = col;
}
//-----
private void paintHandler(object sender,
              PaintEventArgs e) {
       Graphics g = e.Graphics;
       g.DrawString(colName, fnt, bBrush, 20, 40)
}
```

注意，sendNotify事件处理程序接收一个表示颜色名的字符串，用一个哈希表（Hashtable）将这些字符串转换成实际的颜色对象。

在主程序里，每次有人单击一个单选按钮时，它就通过遍历观察者集合中的所有对象，调用每个对这些变化感兴趣并进行注册的观察者的 sendNotify 方法。

```
private void opButton_Click(object sender, EventArgs e) {
      RadioButton but = (RadioButton)sender;
      for(int i=0; i< observers.Count; i++ ) {
            Observer obs = (Observer)observers[ i] ;
            obs.sendNotify (but.Text );
      }
}
```

在颜色窗口观察者中，sendNotify方法改变了图片框窗体的背景颜色和文本串。在列表窗口观察者中，sendNotify方法只是将新颜色的名字加到列表框中。在图27.3中看到的是最终程序的运行情况，图27.4是这些接口的UML表示。

27.2　发给观察者的消息

目标应该把什么样的通知发送给它的观察者？在这个精心限定的例子里，通知消息是表示颜色的字符串。单击一个单选按钮，得到该按钮的标题并将其发送给观察者。这当然要假定所有的观察

者都能处理字符串表示。在比较实际的应用环境里，不会总是这种情况，特别是如果观察者还被用来观察其他数据对象的话。在我们这个例子中采取了两种简单的数据转换。

1. 取得单选按钮标签，将其发送给观察者。
2. 在 ColObserver 里将标签转换成一种实际的颜色。

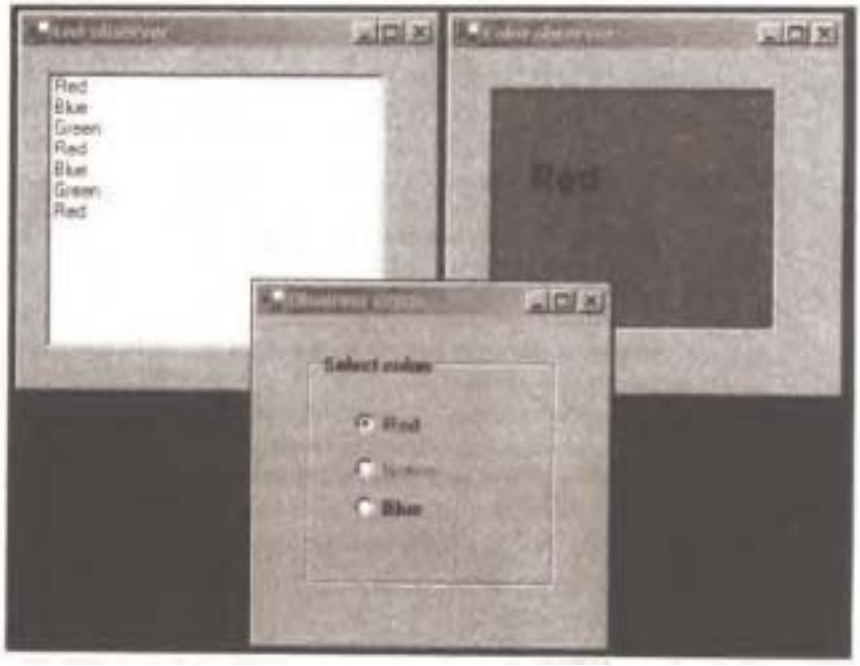


图 27.3 数据控制面板产生的数据同时以一个带颜色的面板和一个列表框显示。这种情况适合于使用观察者模式

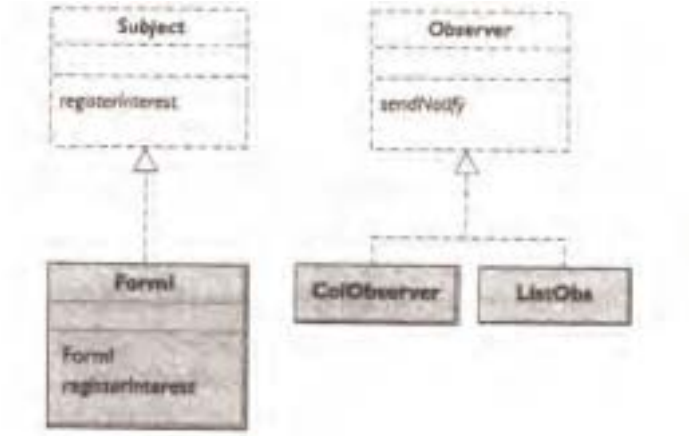


图 27.4 观察者模式中观察者接口和目标接口的实现

在比较复杂的系统里，观察者可能要求各种不同类型的数据。我们不是让各个观察者把消息转换成正确的数据类型，而是用一个中间的 Adapter 类完成这种转换工作。

观察者需要处理的另一个问题是目标类的数据能以几种方式改变的情况。例如，可以从一个数据列表中删除若干节点，编辑它们的数据或改变我们观测的数据的范围，在这些情况下，或者把各种不同的改变消息都发送给观察者，或者只发送一个消息，然后让观察者询问发生的是哪一种改变。

27.3　观察者模式的效果

Observer促进了目标的抽象耦合，目标不知道任何一个观察者的详细内容。但这也具有潜在的缺点：当目标中的数据发生了一系列的递增变化时，要持续或反复地更新观察者。如果更新的代价很高，就有必要引进某种管理更改的策略，这样就不会多次或频繁地通知观察者。

当一个客户（观察者）对底层数据做了修改，你要决定由哪一个对象去触发送给其他观察者的更新通知。如果是由目标在它被更改后去通知所有的观察者，每个客户就不需要记住去触发通知。但另一方面，这会导致多个连续的更新被触发。如果由客户告诉目标何时去通知其他的客户，会避免这种连续的通知。但是客户增加了告诉目标何时发送通知的责任。如果一个客户“忘记”了，程序将不能正常运行。

最后一点，依据所发生的变化的类型或范围，为观察者定义多个接收通知的更新方法，以此指定要选择发送的通知类型，在有些情况下，客户能由此忽略掉一些更新通知。

27.4　随书附带光盘中的程序

Observer Example	`\Observer`

第28章 状态模式

当读者想用一个对象表示应用程序的状态，并通过转换对象来转换程序的状态时，可以使用状态模式（State Pattern）。例如，让一个包装类在多个相关的内部类之间转换，并将方法调用传递给当前的内部类。“Design Patterns”指出，状态模式在内部类之间转换，使得包装对象看起来似乎是修改了它的类。这在C#中有点夸大其辞，但却能大大地改变使用这些类的真正目的。

许多程序员都有这样的经验，即创建一个类，让它根据传进来的参数执行不同的计算或显示不同的信息。通常在类内部使用一些select-case或if-else语句来决定执行哪一种行为，这正是状态模式要力图取代的粗糙之处。

28.1 示例代码

我们考虑一个绘图程序的例子，它类似于我们为备忘录模式开发的程序。该程序拥有几个工具栏按钮：Select，Rectangle，Circle，Fill，Undo和Clear。在图28.1中显示了这个程序的运行情况。

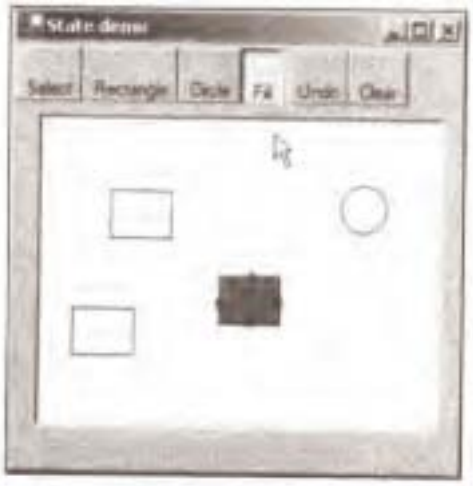


图28.1 一个说明状态模式的简单绘图程序

当选中一个按钮并单击鼠标或在屏幕上拖动鼠标时，每个按钮所做的事情都不一样。也就是说，图形编辑器的状态会影响程序要执行的动作。这种情况适合于使用状态模式进行设计。

我们一开始将程序设计成这样：用一个中介者管理五个命令按钮的操作，如图28.2所示。

这种设计将维护程序状态的全部负担都加在了中介者上，而我们知道，使用中介者的主要目的是协调各种控件（比如按钮）之间的操作。将按钮的状态和预期的鼠标操作都放在中介者中，会使它变得过度复杂，也会导出多个if或select测试语句，使程序难以理解和维护。

此外，这种大量单一的条件语句还需要在中介者解释的每个动作里反复出现，例如mouseUp，mouseDrag，rightClick等动作。这会使程序的理解和维护变得非常困难。

我们换种方法，先分析一下每个按钮的预期行为：

1. 如果选择了Select按扭，单击一个图形元素内部会使它高亮显示或显示成“处理”形式。如果已选择了一个图形元素且拖动鼠标，该图形会在屏幕上移动。
2. 如果选择了Rect按钮，单击屏幕会创建一个新的矩形图形元素。
3. 如果选择了Fill按钮且已选择了一个图形元素，将以当前颜色填充该图形。如果没有选择图形，单击一个图形的内部区域会以当前颜色填充它。
4. 如果选择了Circle按钮，单击屏幕会创建一个新的圆形图形元素。
5. 如果选择了Clear按钮，所有图形元素都被清除。

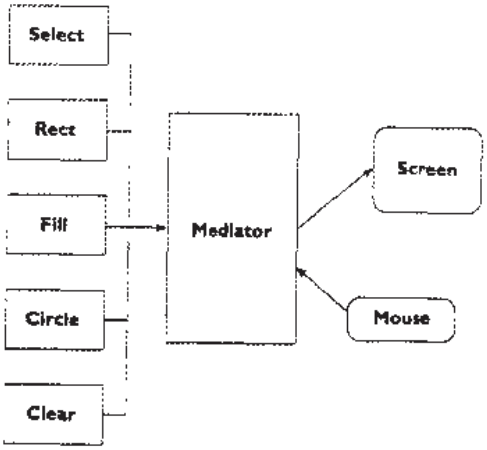


图28.2 支持简单绘图程序的类之间的一种协作方式

在这几个动作之间，还有一些常用的线程需要我们仔细考虑。其中四个线程使用鼠标单击事件引起动作，一个线程使用鼠标拖动事件引起一个动作。因此，需要创建一个系统来根据当前选择的按钮管理这些事件。

先创建一个State对象来处理鼠标操作。

```
public class State     {
       //keeps state of each button
       protected Mediator med;
       public State(Mediator md) {
              med = md;      //save reference to mediator
       }
       public virtual void mouseDown(int x, int y) {}
       public virtual void mouseUp(int x, int y) {     }
       public virtual void mouseDrag(int x, int y) {}
 }
```

注意，创建的是一个带有空方法的真实类，而不是一个接口。这可以使我们从该类派生出新的State对象，并且只需根据具体情况下要做的实际工作填写（重定义）鼠标操作。接下来为Pick，Rect，Circle和Fill四个按钮创建四个派生的State类，并将它们的实例放在StateManager类中，StateManager设置当前的状态并针对该状态对象执行相应的方法。在“Design Patterns”里，StateManager称为Context。该类可用图28.3来说明。

一个典型的State对象只需要重定义它必须专门处理的事件方法。例如，下面是一个完整的Rectangle状态对象，由于它只需要响应mouseDown事件，所以不必为其他事件编写任何代码。

```
public class RectState :State    {
       public RectState(Mediator md):base (md) {}
       //-----
```

```
        public override void mouseDown(int x, int y) {
                VisRectangle vr = new VisRectangle(x, y);
                med.addDrawing (vr);
        }
}
```

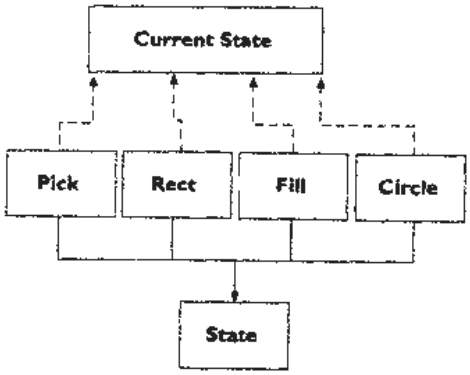


图 28.3 管理当前状态的 StateManager 类

RectState对象告诉中介者向图形列表中添加一个矩形。同样地，Circle状态对象告诉中介者向图形列表中添加一个圆形。

```
public class CircleState : State {
        public CircleState(Mediator md):base (md){ }
        //-----
        public override void mouseDown(int x, int y) {
                VisCircle c = new VisCircle(x, y);
                med.addDrawing (c);
        }
}
```

惟一比较棘手的按钮是Fill，因为我们为它定义了两个动作：

1. 如果已经选择了一个对象，填充它。
2. 如果在一个对象内部单击鼠标，填充该对象。

为了完成这些任务，需要在基类State中添加一个selectOne方法，选择每个工具按钮时都调用该方法。

```
public class State      {
        //keeps state of each button
        protected Mediator med;
        public State(Mediator md) {
                med = md;       //save reference to mediator
        }
        public virtual void mouseDown(int x, int y) {}
        public virtual void mouseUp(int x, int y) {}
        public virtual void mouseDrag(int x, int y) {}
        public virtual void selectOne(Drawing d) {}
}
```

Drawing参数或者是当前选中的图形，或者是在没有图形选中时的空值。在这个简单程序里，我们将填充颜色设置为红色，所以FillState类就变成了下面这种形式：

```
public class FillState : State   {
        public FillState(Mediator md): base(md)    { }
```

```
        //-----
        public override void mouseDown(int x, int y) {
               //Fill drawing if you click inside one
               int i = med.findDrawing(x, y);
               if (i >= 0) {
                    Drawing d = med.getDrawing(i);
                    d.setFill(true);  //fill drawing
               }
        }
        //-----
        public override void selectOne(Drawing d) {
               //fill drawing if selected
               d.setFill (true);
        }
  }
```

28.2 状态之间的转换

我们已经定义了在鼠标事件发送给每个状态时，各状态会产生什么样的动作，现在考虑StateManager如何在状态之间转换。创建每个状态的实例，然后将currentState变量设成由选中的按钮指示的状态。

```
public class StateManager  {
       private State currentState;
       private RectState rState;
       private ArrowState aState;
       private CircleState cState;
       private FillState fState;

       public StateManager(Mediator med)          {
              //create an instance of each state
              rState = new RectState(med);
              cState = new CircleState(med);
              aState = new ArrowState(med);
              fState = new FillState(med);
              //and initialize them
              //set default state
              currentState = aState;
       }
```

注意，在这一版本的StateManager中，在构造函数中创建了每个状态的一个实例，并在调用设置（set）方法时将一个正确的状态复制到状态变量中。也可以根据需要创建这些状态，在状态数目很大而每个状态又消耗很多资源时，建议采用这种形式。

StateManager的其余代码只是调用当前状态对象的方法，其关键之处在于没有条件检查。正确的状态已经设置好，就等待调用它的方法。

```
       public void mouseDown(int x, int y) {
              currentState.mouseDown (x, y);
       }
       public void mouseUp(int x, int y) {
              currentState.mouseUp (x, y);
       }
       public void mouseDrag(int x, int y) {
              currentState.mouseDrag (x, y);
```

```
    }
    public void selectOne(Drawing d) {
          currentState.selectOne (d);
    }
```

28.3 Mediator如何与StateManager互相协作

曾经提到过，把状态管理与Mediator的按钮事件和鼠标事件管理分开是比较明智的。Mediator是比较重要的类，它要在当前程序状态发生改变时告诉StateManager。Mediator的开始部分说明状态的改变是如何发生的。注意，每个按钮单击事件会调用Mediator的一个方法并改变程序的状态。每个方法的剩余语句都关闭了其他按钮的触发，这样一次只能有一个按钮被按下。

```
public class Mediator       {
      private bool startRect;
      private int selectedIndex;
      private RectButton rectb;
      private bool dSelected;
      private ArrayList drawings;
      private ArrayList undoList;
      private RectButton rButton;
      private FillButton filButton;
      private CircleButton circButton;
      private PickButton arrowButton;
      private PictureBox canvas;
      private int selectedDrawing;
      private StateManager stMgr;
      //-----
      public Mediator(PictureBox pic)      {
            startRect = false;
            dSelected = false;
            drawings = new ArrayList();
            undoList = new ArrayList();
            stMgr = new StateManager(this);
            canvas = pic;
            selectedDrawing = -1;
      }
      //-----
      public void startRectangle() {
            stMgr.setRect();
            arrowButton.setSelected(false);
            circButton.setSelected(false);
            filButton.setSelected(false);
      }
      //-----
      public void startCircle() {
            stMgr.setCircle();
            rectb.setSelected(false);
            arrowButton.setSelected(false);
            filButton.setSelected(false);
      }
```

28.3.1 ComdToolBarButton类

在讨论备忘录模式时，我们创建了几个与工具栏按钮并列的按钮命令对象，并将它们保存在一个哈希表（Hashtable）中，在工具栏按钮单击事件发生时会调用该表。另外一种很有效的方式是创

建一个ComdToolBarButton类，它实现了Command接口并继承了ToolBarButton类，这样，每个按钮都能有一个定义其目标的Execute方法。下面给出的是基类ComdToolBarButton。

```
public class ComdToolBarButton : ToolBarButton, Command    {
      private System.ComponentModel.Container components = null;
      protected Mediator med;
      protected  bool selected;
      public ComdToolBarButton(string caption, Mediator md)
      {
            InitializeComponent();
            med = md;
            this.Text =caption;
      }
      //------
      public void setSelected(bool b) {
            selected = b;
            if(!selected)
                  this.Pushed =false;
      }
      //-----
      public virtual void Execute() {
      }
```

注意，该基类中的Execute方法是空的，但它是virtual方法，这样就能在每个派生类里重定义它。这种情况下不能用IDE创建工具栏，但可通过简单的编程向工具栏添加按钮。

```
private void init() {
            //create a Mediator
            med = new Mediator(pic);
            //create the buttons
            rctButton = new RectButton(med);
            arowButton = new PickButton(med);
            circButton = new CircleButton(med);
            flButton = new FillButton(med);
            undoB = new UndoButton(med);
            clrb = new ClearButton(med);
            //add the buttons into the toolbar
            tBar.Buttons.Add(arowButton);
            tBar.Buttons.Add(rctButton);
            tBar.Buttons.Add(circButton);
            tBar.Buttons.Add(flButton);
            //include a separator
            ToolBarButton sep =new ToolBarButton();
            sep.Style = ToolBarButtonStyle.Separator;
            tBar.Buttons.Add(sep);
            tBar.Buttons.Add(undoB);
            tBar.Buttons.Add(clrb);
}
```

然后，就可以在一个方法里捕获所有的工具栏按钮单击事件，并调用每个按钮的Execute方法。

```
private void tBar_ButtonClick(object sender,
            ToolBarButtonClickEvnetArgs e) {
      Command comd = (Command) e.Button;
      comd.Execute ();
}
```

说明该程序中状态模式的程序类图被分成两部分。State部分如图28.4所示。

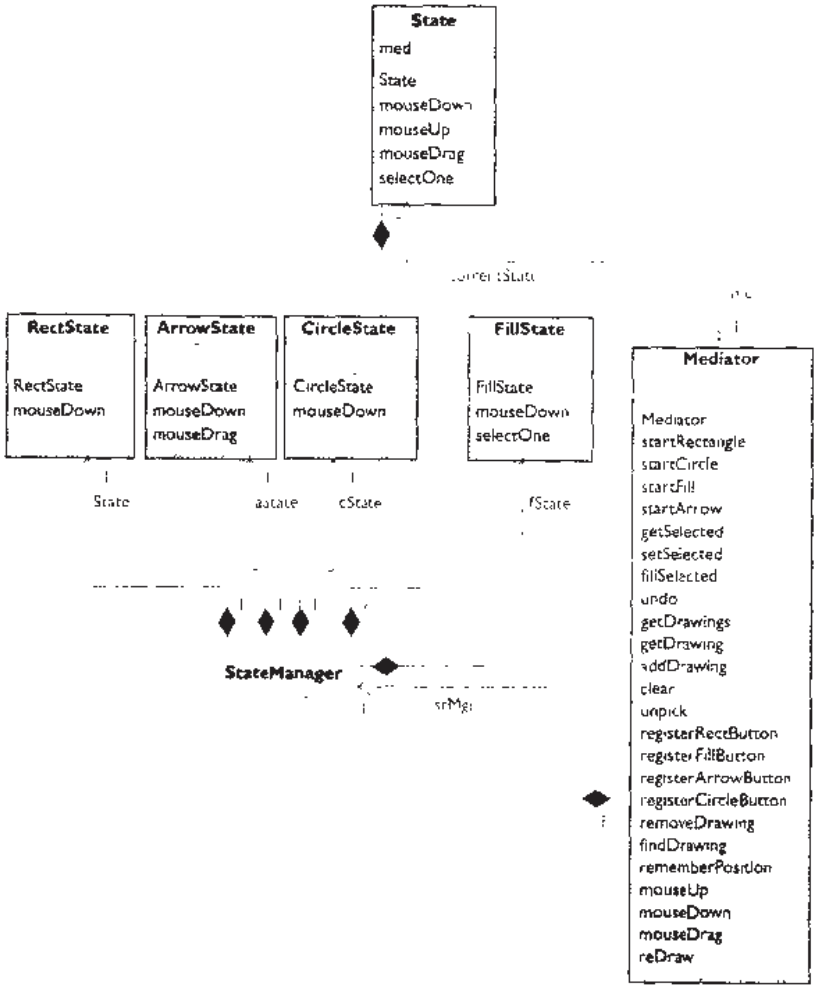


图 28.4 StateManager 类和 Mediator 类

Mediator 与按钮的连接如图 28.5 所示。

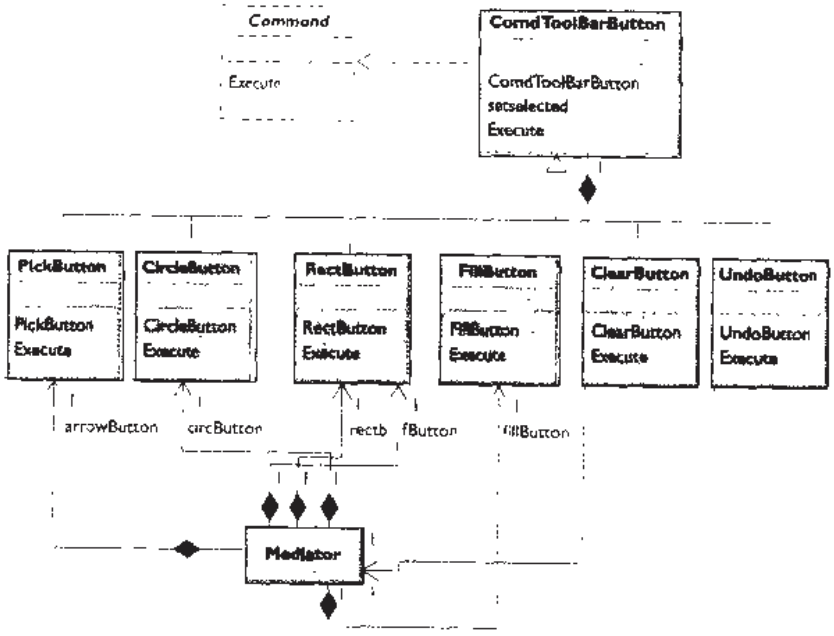


图 28.5 按钮与 Mediator 类之间的相互协作

28.4 处理 Fill 状态

因为要处理两种情况，FillState对象略微复杂些。如果已经选中一个对象，程序会填充它，或者单击一个对象时，程序填充该对象。这就表示：针对这两种情况，FillState要填充（重定义）两个方法，如下所示：

```
public class FillState : State  {
      public FillState(Mediator md): base(md)    { }
      //-----
      public override void mouseDown(int x, int y) {
            //Fill drawing if you click inside one
            int i = med.findDrawing(x, y);
            if (i >= 0) {
                  Drawing d = med.getDrawing(i);
                  d.setFill(true);  //fill drawing
            }
      }
      //-----
      public override void selectOne(Drawing d) {
            //fill drawing if selected
            d.setFill (true);
      }
}
```

28.5 处理 Undo 列表

现在，我们要在绘图程序中取消所执行的操作，也就是说，应该把这些操作保存到一个取消列表中。下面给出我们能执行和取消的操作。

1. 创建一个矩形。
2. 创建一个圆形。
3. 移动一个矩形或圆形。
4. 填充一个矩形或圆形。

在我们讨论备忘录模式时，用一个Memento对象保存矩形对象的状态，并在需要时从Memento中恢复它的位置。这种方法对矩形和圆形都是正确的，因为需要保存和恢复同样的位置信息。但是，矩形或圆形的添加以及对不同图形的填充也是我们希望能取消的操作，而且正如我们在前面备忘录模式的讨论中指出的那样，通过检查取消列表中对象的类型来执行正确的取消操作，是一个很糟糕的做法。

```
//really terrible programming approach
    object obj = undoList[ last];
    try{
          Memento mem = (Memento)obj;
          remove(mem);
    }
    catch(Exception)
          removeDrawing();
    }
```

不妨换一种方法，将Memento定义成一个接口。

```
public interface Memento   {
      void restore ();
}
```

然后，让所有添加到取消列表中的对象都实现 Memento 接口，并且拥有一个执行某种操作的 restore 方法。一些 Memento 对象保存和恢复图形的坐标，另外一些 Memento 对象只是删除图形或取消填充状态。

首先让圆形和矩形对象都实现 Drawing 接口。

```
public interface Drawing   {
       void setSelected(bool b);
       void draw(Graphics g);
       void move(int xpt, int ypt );
       bool contains(int x, int y);
       void setFill(bool b);
       CsharpPats.Rectangle getRects();
       void setRects(CsharpPats.Rectangle rect);
 }
```

用来保存一个图形状态的 Memento 类和我们在备忘录模式一章中使用的类相似，除了要指定它去实现 Memento 接口外。

```
public class DrawMemento : Memento    {
      private int x, y, w, h;
      private Rectangle  rect;
      private Drawing visDraw;
      //------
      public DrawMemento(Drawing d)   {
             visDraw = d;
             rect = visDraw.getRects ();
             x = rect.x;
             y = rect.y;
             w = rect.w;
             h = rect.h;
      }
      //-----
      public void restore() {
             //restore the state of a drawing object
             rect.x = x;
             rect.y = y;
             rect.h = h;
             rect.w = w;
             visDraw.setRects( rect);
      }
}
```

对于只想从列表中删除一个图形的情况，创建一个类记住该图形的索引，并在调用 restore 方法时删除它。

```
public class DrawInstance :Memento {
      private int intg;
      private Mediator med;
      //-----
      public DrawInstance(int intg, Mediator md) {
             this.intg = intg;
             med = md;
      }
```

```
        //-----
        public int integ {
                get { return intg;        }
        }
        //-----
        public void restore() {
                med.removeDrawing(intg);
        }
}
```

以同样的方式处理FillMemento，除了它的restore方法关闭了图形元素的填充标志。

```
public class FillMemento : Memento    {
        private int index;
        private Mediator med;
        //-----
        public FillMemento(int dindex, Mediator md) {
                index = dindex;
                med = md;
        }
        //-----
        public void restore() {
                Drawing d = med.getDrawing(index);
                d.setFill(false);
        }
}
```

28.6 VisRectangle类和VisCircle类

在设计VisRectangle类和VisCircle类时，可以利用继承的一些优点。我们让VisRectangle类实现Drawing接口，然后让VisCircle类继承VisRectangle。这样可以重用setSelected，setFill，move等方法，以及矩形的一些属性。另外，分离出drawHandle方法，让它能用在两个类里。新的VisRectangle类如下所示：

```
public class VisRectangle : Drawing   {
        protected int x, y, w, h;
        private const int SIZE=30;
        private CsharpPats.Rectangle rect;
        protected bool selected;
        protected bool filled;
        protected Pen bPen;
        protected SolidBrush bBrush, rBrush;
        //-----
        public VisRectangle(int xp, int yp)       {
                x = xp;              y = yp;
                w = SIZE;            h = SIZE;
                saveAsRect();
                bPen = new Pen(Color.Black);
                bBrush = new SolidBrush(Color.Black);
                rBrush = new SolidBrush (Color.Red );
        }
        //-----
        //used by Memento for saving and restoring state
         public CsharpPats.Rectangle getRects() {
              return rect;
        }
```

```
        //-----
        public void setRects(CsharpPats.Rectangle value) {
                        x=value.x;              y=value.y;
                        w=value.w;              h=value.h;
                        saveAsRect();
        }
        //------
        public void setSelected(bool b) {
               selected = b;
        }
        //-----
        //move to new position
        public void move(int xp, int yp) {
               x = xp;       y = yp;
               saveAsRect();
        }
        //-----
        public virtual void draw(Graphics g) {
               //draw rectangle
               g.DrawRectangle(bPen, x, y, w, h);
               if(filled)
                    g.FillRectangle (rBrush, x, y, w, h);
               drawHandles(g);
        }
        //-----
        public void drawHandles(Graphics g) {
        if (selected) {    //draw handles
               g.FillRectangle(bBrush, x + w / 2, y - 2, 4, );
               g.FillRectangle(bBrush, x - 2, y + h / 2, 4, );
               g.FillRectangle(bBrush, x + (w / 2),
                      y + h - 2, 4, 4);
               g.FillRectangle(bBrush, x + (w - 2),
               y + (h / 2), 4, 4);
        }
        }
        //-----
        //return whether point is inside rectangle
        public bool contains(int x, int y) {
               return rect.contains (x, y);
        }
        //------
        //create Rectangle object from new position
        protected void saveAsRect() {
               rect = new CsharpPats.Rectangle (x,y,w,h);
        }
        public void setFill(bool b) {
               filled = b;
        }
```

VisCircle类只需要重定义draw方法，并给出一个略微不同的构造函数。

```
public class VisCircle : VisRectangle {
      private int r;
      public VisCircle(int x, int y):base(x, y)       {
            r = 15; w = 30; h = 30;
            saveAsRect();
      }
      //-----
```

```
        public override void draw(Graphics g) {
                if (filled) {
                        g.FillEllipse(rBrush, x, y, w, h);
                }
        g.DrawEllipse(bPen, x, y, w, h);
        if (selected ){
                drawHandles(g);
                }
        }
}
```

注意，由于我们将x，y和filled变量声明为保护的，所以，在派生类VisCircle里不用再声明就可以直接引用它们。

28.7 Mediator类和“万能类”

当有许多对象相互协作时，程序的一个大问题是，太多的系统内容放在了Mediator里，使它成了一个“万能类”。在前面的例子里，Mediator要同六个按钮、一个图形列表和StateManager通信。我们可以用另外一种方式编写这个程序，让按钮命令对象同StateManager通信，而Mediator只处理按钮和图形列表。其中，每个按钮创建一个它所需的状态的实例，并将其发送给StateManager。将它作为练习留给读者。

28.8 状态模式的效果

1. 状态模式为应用程序所具有的每一个状态创建了一个基类State的子类，并在应用程序改变状态时在这些状态之间转换。
2. 没有与各状态相关联的一长串的if或switch条件语句，因为每个状态都封装在一个类中。
3. 由于在别处没有指明程序所处状态的变量，减少了因程序员忘记检查状态变量而引起的错误。
4. 可以在几部分应用程序之间共享状态对象，比如独立的窗口之间，只要没有一个状态对象具有专门的实例变量。在本例中，只有FillState类拥有一个实例变量，我们很容易将它改成一个每次调用时被传递的参数。
5. 这种方法会产生许多小的类对象，但其处理过程会使程序简单清楚。
6. C#中，所有的State都必须实现一个公共接口，因而必然有共同的方法，尽管某些方法可能是空的。在其他语言里，可以用函数指针实现状态，它需要较少的类型检查，但出现错误的可能性较大。

28.9 状态转换

状态之间的转换可以内部指定或外部指定。在我们的例子里，Mediator告诉StateManager何时在状态之间转换，也可以让每个状态自动决定其后继状态是什么。例如，当创建完一个矩形或圆形图形对象后，程序能自动转回到Arrow状态。

28.10 思考题

1. 重写StateManager，用工厂模式根据需要产生状态。

2. 尽管可视化图形程序是显而易见的应用状态模式的好例子，服务器程序也能从这种方法中受益。概述一个简单的使用状态模式的服务器程序。

28.11 随书附带光盘中的程序

State Drawing Program	\State

第29章 策略模式

策略模式（Strategy Pattern）在外形上与状态模式很相似，但在意图上有些不同。策略模式由多个封装在一个称为Context的驱动器类里的相关算法组成。客户程序可以从这些不同的算法中选择一个，或者在某些情况下，由Context替你选择一个最好的算法。策略模式的意图是使这些算法可交换，并提供一种方法来选择最合适的算法。状态模式和策略模式之间的区别是：用户通常选择一种策略使用，在Context类中一次只能有一个策略实例化和运行；与此相反，正如我们前面所看到的，所有不同的状态可以同时都是活动的（active），在它们之间经常进行转换。另外，策略模式封装的算法所做的工作或多或少是相同的，而状态模式封装的相关类所做的事情多少有些不同。最后一点，在不同状态之间转换的概念在策略模式中已经完全没有了。

29.1 动机

如果一个程序需要一种特定的服务或功能，而且该程序有多种实现该功能的方式，此时适合于使用策略模式。程序可根据运算效率或用户选项在这些算法之间选择。程序中可以有任意数量的策略，可以添加策略，也可以在任何时候修改策略。

程序中有很多这样的情形：希望能以几种不同的方式做同样的事情。“Smalltalk Companion”中列举出了这样的一些情形。

- 以不同的格式保存文件。
- 用不同的算法压缩文件。
- 用不同的压缩机制捕获视频数据。
- 用不同的行分割策略显示文本数据。
- 以不同的形式对同样的数据绘图：线状图、条形图或饼图。

对每种情形，都让客户程序告诉驱动器模块（Context）使用哪一个策略，然后让该策略完成相应的操作。

策略模式的基本思想是：将不同的策略封装在一个模块中，并提供一个简单的接口在这些策略之间进行选择。尽管这些策略不必都是同一类层次结构中的成员，但每个策略都应该有相同的程序设计接口。

29.2 示例代码

我们考虑一个简化的绘图程序，它能将数据表示成一个线状图或条形图。先创建一个抽象类PlotStrategy，然后从它派生出两个绘图类，如图29.1所示。

基类PlotStrategy是一个抽象类，它包含了一个plot例程，在派生的策略类中要为它填充代码。

```
public abstract class PlotStrategy  {
      public abstract void plot( float[] x, float[] y);
}
```

plot方法用两个浮点数组作为参数，每个派生策略类都必须实现该方法，这样才能绘制合适的图形。

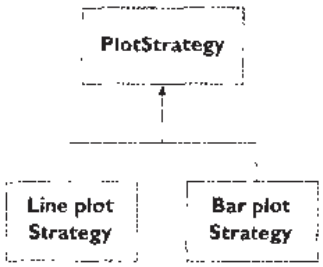


图 29.1 PlotStrategy 的两个派生类

29.3 Context 类

Context类充当交通警察的角色，它决定调用哪一种策略。这种决策通常要依据客户程序的请求，而 Context 所要做的就是将一个变量设置成对具体策略的一个引用。

```
public class Context  {
      float[] x, y;
      PlotStrategy plts;   //strategy selected goes here
      //-----
      public void plot() {
            readFile();        //read in data
            plts.plot (x, y);
      }
      //-----
      //select bar plot
      public void setBarPlot() {
            plts = new BarPlotStrategy ();
      }
      //-----
      //select line plot
      public void setLinePlot() {
            plts = new LinePlotStrategy();
      }
      //-----
      public void readFile()          {
      //reads data in from data file
      }
}
```

Context类还需要负责处理数据。它要么是从文件或数据库获取数据，要么是在创建Context时传递进来数据。根据数据量的多少，或者将数据传递给绘图策略，或者是Context将本身的一个实例传给绘图策略，并提供一个取出数据的公有方法。

29.4 程序中的 Command 对象

这个简单程序（图29.2）只是一个带有两个按钮的面板，它们调用两个绘图程序。每个按钮都是一个派生的Button类，都实现了Command接口。按钮选择正确的策略，然后调用Context的plot例程。例如，下面就是一个完整的线状图的命令 Button 类。

```
public class LineButton : System.Windows.Forms.Button, Command
{
      private System.ComponentModel.Container components = null;
      private Context contxt;
      public LineButton()   {
            InitializeComponent();
            this.Text = "Line plot";
      }
      public void setContext(Context ctx) {
            contxt = ctx;
      }
      public void Execute() {
            contxt.setLinePlot();
            contxt.plot();
      }
```

图29.2 一个调用两个不同绘图程序的面板

29.5 线状图和条形图策略

两个Strategy类非常相似：它们都要为绘图设置窗口尺寸，都要为相应的显示面板调用专门的plot方法。下面给出的是LinePlotStrategy类。

```
public class LinePlotStrategy : PlotStrategy    {
      public override void plot(float[] x, float[] y) {
            LinePlot lplt = new LinePlot();
            lplt.Show ();
            lplt.plot (x, y);
      }
}
```

BarPlotStrategy类或多或少与此相同。

绘图过程就是：拷贝参数给x，y数组，调用换算函数，然后使Picturebox控件刷新，它再接下来调用paint程序绘出条形图。

```
public void plot(float[] xp, float[] yp) {
      x = xp;
      y = yp;
      setPlotBounds();    //compute scaling factors
      hasData = true;
      pic.Refresh();
}
```

29.6 用C#绘制图形

注意，LinePlot窗口和BarPlot窗口都有plot方法，分别被LinePlotStrategy和BarPlotStrategy类中的plot方法调用。两个绘图窗口都有一个setBounds方法，用于完成窗口坐标和x-y坐标之间的

换算。因为它们使用的是同样的换算函数，可以只在BarPlot窗口中书写一次，然后从BarPlot派生出LinePlot窗口，它就也能使用这一方法。

```
public virtual void setPlotBounds() {
      findBounds();
      //compute scaling factors
      h = pic.Height;
      w = pic.Width;
      xfactor = 0.8F * w / (xmax - xmin);
      xpmin = 0.05F * w;
      xpmax = w - xpmin;
      yfactor = 0.9F * h / (ymax - ymin);
      ypmin = 0.05F * h;
      ypmax = h - ypmin;
      //create array of colors for bars
      colors = new ArrayList();
      colors.Add(new SolidBrush(Color.Red));
      colors.Add(new SolidBrush(Color.Green));
       colors.Add(new SolidBrush(Color.Blue));
      colors.Add(new SolidBrush(Color.Magenta));
      colors.Add(new SolidBrush(Color.Yellow));
}
//-----
public int calcx(float xp) {
      int ix = (int)((xp - xmin) * xfactor + xpmin);
      return ix;
}
//-----
public int calcy(float yp) {
      yp = ((yp - ymin) * yfactor);
      int iy = h - (int)(ypmax - yp);
      return iy;
}
```

29.6.1 绘制条形图

条形图的实际绘制工作是在Paint例程中，在paint事件发生时调用该程序。

```
protected virtual void pic_Paint(object sender, PaintEventArgs e)
{
      Graphics g = e.Graphics;
      if (hasData) {
      for (int i = 0; i< x.Length; i++){
            int ix = calcx(x[ i] );
            int iy = calcy(y[ i] );
            Brush br = (Brush)colors[ i] ;
            g.FillRectangle(br, ix, h - iy, 20, iy);
      }
}
```

29.6.2 绘制线状图

因为是从BarPlot派生出来的，所以LinePlot类非常简单，只需为它编写一个新的Paint方法。

```
public class LinePlot :BarPlot {
      public LinePlot() {
            bPen = new Pen(Color.Black);
            this.Text = "Line Plot";
```

```
        }
        protected override void pic_Paint(object sender,
                    PaintEventArgs e) {
                Graphics g= e.Graphics;
                if (hasData) {
                        for (int i = 1; i< x.Length; i++) {
                                int ix = calcx(x[ i - 1] );
                                int iy = calcy(y[ i - 1] );
                                int ix1 = calcx(x[ i] );
                                int iy1 = calcy(y[ i] );
                                g.DrawLine(bPen, ix, iy, ix1, iy1);
                        }
                }
        }
}
```

说明类之间关系的UML图如图29.3所示。最终绘制出的两个图形如图29.4所示。

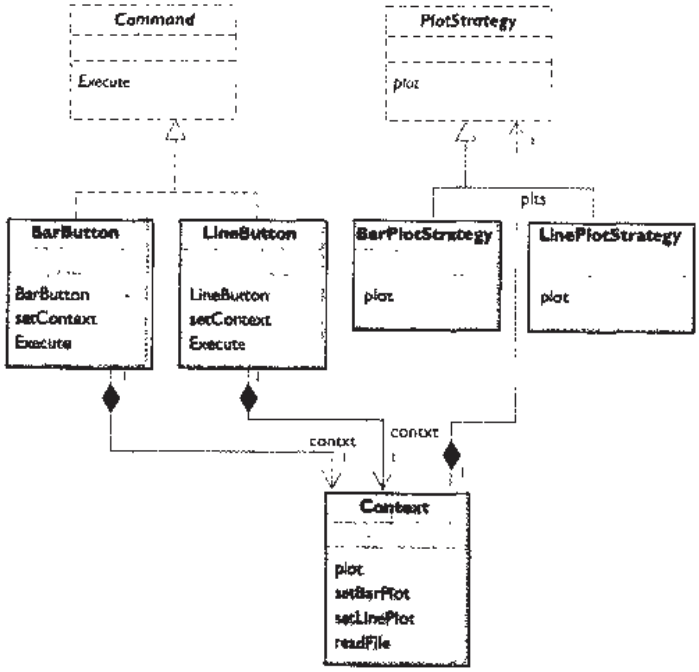


图29.3 策略模式的UML图

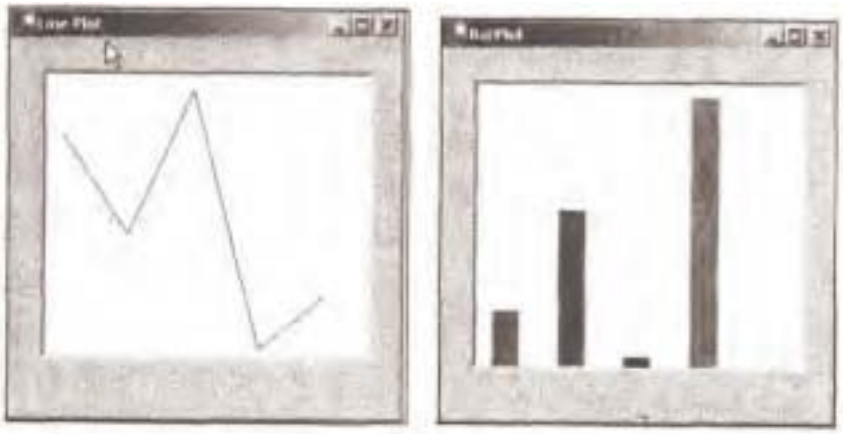


图29.4 线状图（左）和条形图（右）

29.7 策略模式的效果

策略模式允许从几个算法中自动选择一个算法。这些算法可以是在一个继承层次结构中相互关联的，也可以是毫无关联但实现了一个共同的接口。因为Context能根据请求在各策略间转换，所以它比调用相应的派生类有更大的灵活性。这种方法也避免了使用条件语句，那只会使代码难以理解和维护。

但另一方面，策略模式没有把每件事都隐藏起来。客户代码通常都会知道有多少个可替换的算法，还要拥有一些选择算法的准则。这就把算法的选择转移到客户端程序员或用户身上了。

因为有多个不同的参数要传递给不同的算法，所以需要开发一个Context接口和若干策略方法，这些方法的范围要足够大，允许传进来的参数是某个算法用不到的。例如，PlotStrategy 中的setPenColor方法实际上只被LineGraph策略使用，BarGraph策略用不到它，因为它为绘制后续的条形图建立了自己的颜色列表。

29.8 随书附带光盘中的程序

Plot Strategy	\Strategy

第30章　模板方法模式

模板方法模式（Template Method Pattern）是一种非常简单而又经常使用的模式。先创建一个父类，把其中的一个或多个方法留给派生类实现，这实际上就是在使用模板模式。模板模式形式化了下列想法：在类中定义一个算法，但将算法的某些细节留到子类中实现。换句话说，如果基类是一个抽象类（这在设计模式里经常出现），那么你就是在使用一种简单形式的模板模式。

30.1　动机

模板模式非常基本，以至于可能没有经过考虑就已经使用了几十次。模板模式的基本思想是：一个算法的某些部分已经有完善的定义，可以在基类中实现，而其他部分会有几种实现形式，最好留给派生类实现。模板的另一个主题是：将一个类的基本部分抽取出来放到一个基类中，这样它就不必重复出现在几个派生类里。

例如，在前一章策略模式的例子中用到的BarPlot类和LinePlot类里，我们发现在绘制线状图和条形图时，要用相同的代码去换算数据和计算x，y的像素位置。

```
public abstract class PlotWindow : Form {
        protected float ymin, ymax, xfactor, yfactor;
        protected float xpmin, xpmax, ypmin, ypmax, xp, yp;
        private float xmin, xmax;
        protected int w, h;
        protected float[] x, y;
        protected Pen bPen;
        protected bool hasData;
        protected const float max = 1.0e38f;
        protected PictureBox pic;
        //-----
        protected virtual void init() {
                pic.Paint += new PaintEventHandler (pic_Paint);
        }
        //-----
        public void setPenColor(Color c){
                bPen = new Pen(c);
        }
        //-----
        public void plot(float[] xp, float[] yp) {
                x = xp;
                y = yp;
                setPlotBounds();     //compute scaling factors
                hasData = true;
        }
        //-----
        public void findBounds() {
                xmin = max;
                xmax = -max;
                ymin = max;
                ymax = -max;
```

```
            for (int i = 0; i<  x.Length; i++) {
                if (x[ i] > xmax) xmax = x[ i];
                if (x[ i] < xmin) xmin = x[ i];
                if (y[ i] > ymax) ymax = y[ i];
                if (y[ i] < ymin) ymin = y[ i];
            }
        }
        //-----
        public virtual void setPlotBounds() {
            findBounds();
            //compute scaling factors
            h = pic.Height;
            w = pic.Width;
            xfactor = 0.8F * w / (xmax - xmin);
            xpmin = 0.05F * w;
            xpmax = w - xpmin;
            yfactor = 0.9F * h / (ymax - ymin);
            ypmin = 0.05F * h;
            ypmax = h - ypmin;
        }
        //-----
        public int calcx(float xp) {
            int ix = (int)((xp - xmin) * xfactor + xpmin);
            return ix;
        }
        //-----
        public int calcy(float yp) {
            yp = ((yp - ymin) * yfactor);
            int iy = h - (int)(ypmax - yp);
            return iy;
        }
        //-----
        public abstract void repaint(Graphics g);
        //-----
        protected virtual void pic_Paint(object sender,
                            PaintEventArgs e) {
            Graphics g = e.Graphics;
            repaint(g);
        }
    }
```

这些方法都没有实际的绘图功能，都归入到基类 PlotWindow 里。注意，pic_Paint 事件处理程序只是调用了抽象的 repaint 方法，实际的 repaint 方法延迟到了派生类中。派生类的这种扩展就是模板方法的一个例证。

30.2 Template 类中的方法种类

正如“Design Patterns”中讨论的那样，模板方法模式有四种方法可用在派生类中。

1. 完整的方法，它完成了所有子类都要用到的一些基本功能，例如，前一个例子中的 calcx 和 calcy。这些方法被称为具体的方法。
2. 方法中根本没有内容，必须在派生类中实现的方法。在 C# 中，可以将这种方法声明为 virtual 方法。

3. 方法里包含了某些操作的一种默认实现，派生类里可以重新定义这些方法。这些方法称为钩子（hook）方法。当然，这有些任意，因为在 C# 里，本来就可以在派生类重定义任何一个公有方法或保护方法，但钩子方法是有待重定义的，而具体的方法不是这样。
4. 最后，一个 Template 类可以包含这样一些方法：它们自己调用抽象方法、钩子方法和具体方法的各种组合。这些方法不仅有待重定义，还描述了一个没有实现细节的算法。“Design Patterns”称这些方法为模板方法。

30.3　示例代码

考虑一个简单的、在屏幕上绘制三角形的程序。先创建一个抽象类 Triangle，然后从它派生出一些特殊的三角形类型，如图 30.1 所示。

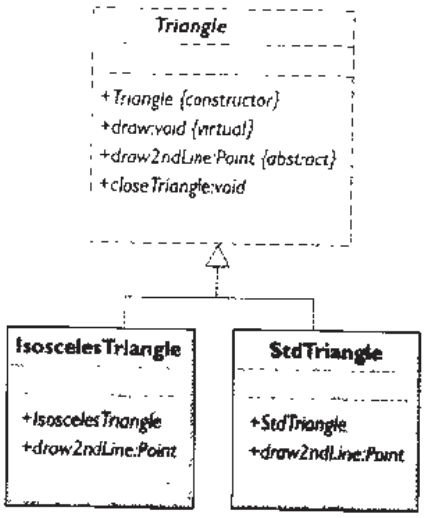


图 30.1　抽象类 Triangle 和它的两个子类

抽象类 Triangle 举例说明了模板模式。

```
public abstract class Triangle   {
       private Point p1, p2, p3;
       protected Pen pen;
       //-----
       public Triangle(Point a, Point b, Point c)      {
              p1 = a;
              p2 =  b;
              p3 = c;
              pen = new Pen(Color.Black, 1);
       }
       //-----
       public virtual void draw(Graphics g) {
              g.DrawLine (pen, p1, p2);
              Point c = draw2ndLine(g, p2, p3);
              closeTriangle(g, c);
       }
       //-----
       public abstract Point draw2ndLine(Graphics g,
                           Point a, Point b);
       //-----
```

```
        public void closeTriangle(Graphics g, Point c) {
             g.DrawLine (pen, c, p1);
        }
    }
```

Triangle类保存了三条线的坐标，但draw程序只画了第一条线和最后一条线。最重要的、画一条到第三个点的线的draw2ndLine方法被声明为抽象方法。这样，派生类可以移动第三个点来创建想画的那种三角形。这是使用模板模式的一个普通例子。draw方法调用了两个具体的基类方法和一个抽象方法，该抽象方法必须在任何派生自 Triangle 的具体类中重定义。

实现 Triangle 类的另外一种非常类似的方法是，让 draw2ndLine 方法包含默认的代码。

```
public virtual void draw2ndLine(Graphics g,
                  Point a, Point b)   {
      g.drawLine(a, b);
}
```

这种情况下，draw2ndLine 方法就成了一个钩子方法，可以被其他类重定义。

30.3.1 绘制一个一般三角形

要绘制一个没有形状限制的一般三角形，只需在派生类 StdTriangle 中实现 draw2ndLine 方法。

```
public class StdTriangle :Triangle    {
      public StdTriangle(Point a, Point b, Point c)
                  : base(a, b, c) {}
      //------
      public override Point draw2ndLine(Graphics g,
                  Point a, Point b) {
            g.DrawLine (pen, a, b);
            return b;
      }
 }
```

30.3.2 绘制一个等腰三角形

该类要计算出第三个点的数据，它要使两边的长度相等，并将新的点保存在类里。

```
public class IsoscelesTriangle : Triangle  {
      private Point newc;
      private int newcx, newcy;
      //-----
      public IsoscelesTriangle(Point a, Point b, Point c) :
                  base(a, b, c) {
            float dx1, dy1, dx2, dy2, side1, side2;
            float slope, intercept;
            int incr;
            dx1 = b.X - a.X;
            dy1 = b.Y - a.Y;
            dx2 = c.X - b.X;
            dy2 = c.Y - b.Y;
            side1 = calcSide(dx1, dy1);
            side2 = calcSide(dx2, dy2);

            if (side2 < side1)
                  incr = -1;
            else
                  incr = 1;
```

```
            slope = dy2 / dx2;
            intercept = c.Y - slope * c.X;

            //move point c so that this is an isosceles triangle
            newcx = c.X;
            newcy = c.Y;
            while (Math.Abs (side1 - side2) > 1) {
                  //iterate a pixel at a time until close
                  newcx = newcx + incr;
                  newcy = (int)(slope * newcx + intercept);
                  dx2 = newcx - b.X;
                  dy2 = newcy - b.Y;
                  side2 = calcSide(dx2, dy2);
            }
            newc = new Point(newcx, newcy);
      }
      //-----
      private float calcSide(float a, float b) {
            return (float)Math.Sqrt (a*a +  b*b);
      }
}
```

Triangle类调用draw方法时，draw方法调用这一新版本的draw2ndLine方法，绘制一条到第三个新点的线。进一步来说，draw2ndLine方法将新的点返回给draw方法，这样就能正确绘制出封闭形状的三角形。

```
public override Point draw2ndLine(Graphics g,
                Point b, Point c ) {
            g.DrawLine (pen, b, newc);
            return newc;
      }
```

30.4 三角形绘图程序

主程序只创建需要绘制的三角形的实例，然后把它们添加到TriangleForm类中的一个ArrayList中。

```
private void init() {
      triangles = new ArrayList();
      StdTriangle t1 = new StdTriangle(new Point(10, 10),
            new Point(150, 50),
            new Point(100, 75));
      IsoscelesTriangle t2 = new IsoscelesTriangle(
            new Point(150, 100), new Point(240, 40),
            new Point(175, 150));
      triangles.Add(t1);
      triangles.Add(t2);
      Pic.Paint+= new PaintEventHandler (TPaint);
}
```

正是该类中的TPaint方法调用Triangle的draw方法绘制出三角形。

```
private void TPaint (object sender,
            System.Windows.Forms.PaintEventArgs e ) {
      Graphics g = e.Graphics;
      for (int i = 0; i<triangles.Count; i++) {
            Triangle t = (Triangle)triangles[ i];
```

```
                t.draw(g);
            }
        }
```

图 30.2 显示了一个一般三角形和一个等腰三角形。

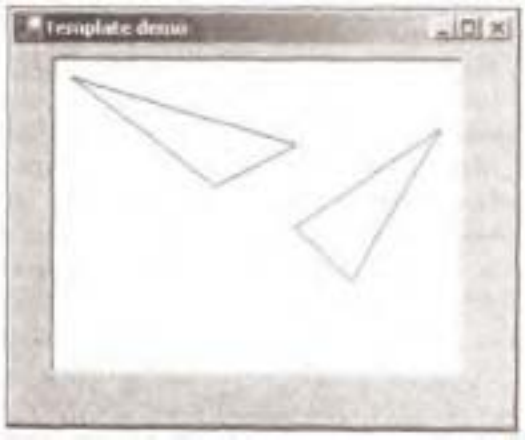


图 30.2 一个一般三角形和一个等腰三角形

30.5 模板与反向调用

"Design Patterns" 指出，模板模式是 "好莱坞法则" 的一个例证，即 "别找我们，我们找你"。在这里它指的是，基类中的方法似乎调用了派生类中的方法（注意这里使用的词是 "似乎"）。如果考虑一下基类 Triangle 中 draw 的代码，就会看到三个方法调用。

```
g.DrawLine (pen, p1, p2);
Point c = draw2ndLine ( g, p2, p3);
closeTriangle (g, c);
```

drawLine 和 closeTriangle 都是在基类中实现的，而 draw2ndLine 方法，正如我们看到的，在基类中根本没有实现，各派生类对它有不同的实现。由于实际被调用的方法是在派生类里，所以看起来像是从基类中调用它们。

如果这种想法使你觉得不舒服，可从下面的结论中获得安慰：所有的方法调用都来自于派生类，这些调用沿着继承链向上传递，直到找到第一个实现这些方法的类为止。如果这个类是基类，很好；如果不是，可能是其他任何一个中间类。当读者调用 draw 方法时，派生类沿着继承树向上传递，直到找到该方法的实现为止。同样地，对 draw 方法内的每个方法调用，派生类都要从当前类开始，沿着树向上寻找实现的方法。当它调用 draw2ndLine 方法时，可以在当前类中立即找到它的实现。所以，它不是 "真正地" 来自于基类的调用，只是看起来像而已。

30.6 小结与效果

模板模式一直存在于 OO 软件中，它既不复杂也不难理解，是 OO 程序设计中一个很普通的部分，不应该将它考虑得过于复杂。

模板模式的第一个要点是，基类可以只定义一些它自己要使用的方法，把其余部分留到派生类中去实现。第二个要点是，基类中的方法可以调用一系列的方法，其中有些方法在基类中实现，有些方法在派生类中实现。该模板方法定义了一个通用的算法，尽管细节部分在基类中没有完全给出来。

Template 类经常会有一些派生类必须重新定义的抽象方法，还会有一些只具有一种简单“占位式”实现的方法，你可以在合适的地方重新定义它。如果这些占位式方法被基类中其他方法调用，我们就称这些可重定义的方法为“钩子”方法。

30.7　随书附带光盘中的程序

Plot strategy using Template method pattern	`\Template\Stragegy`
Plot of triangles	`\Template\Template`

第31章　访问者模式

访问者模式（Visitor Pattern）能将面向对象模型的劣势转化为优势，它可以创建一个外部类来操作其他类中的数据。当一个多形态的操作由于某种原因不能放在类层次中时（例如，因为在层次设计时没考虑到该操作，或者因为没有必要弄乱类之间的接口），使用访问者模式是很有帮助的。

31.1　动机

将本该是另外一个类中的操作放在这个类中，初看起来似乎“含糊不清”，但这样做还是有很充分的理由的。假设很多的绘图对象类都有相似的绘制代码，绘制方法可能不同，但它们都用到基本的、在每个类中都要重复的功能函数。进一步来说，就是一系列本是紧密相关的函数被分散到了多个不同的类中，如图31.1所示。

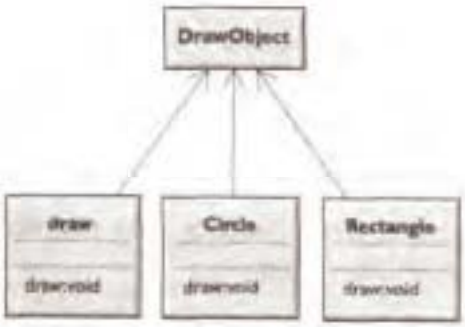


图31.1　DrawObject类和它的三个子类

我们换一种做法，编写一个包含所有相关draw方法的类Visitor，让它接连访问每一个对象（图31.2）。

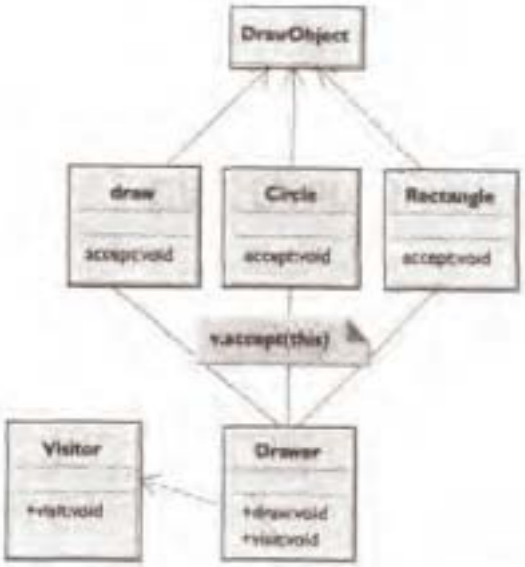


图31.2　Visitor类（Drawer）可访问三个绘图类中的每一个类

大多数人对该模式提出的第一个问题是“访问是什么意思”。一个外部类只有一种途径能访问其他的类，就是调用它的公有方法。在访问者模式中，访问每个类的含义是调用一个为此而存在的方法，即 accept 方法。accept 方法有一个参数是访问者实例，被访问的实例可以根据该参数回过来再调用访问者的 visit 方法，并在调用时将自身作为一个参数传递过去，如图 31.3 所示。

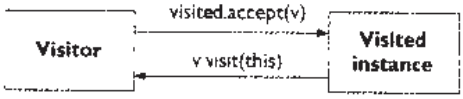


图 31.3　visit 方法和 accept 方法相互配合

用简单的编码来表示，就是要访问的每个对象都必须具备下面的方法：

```
public virtual void accept(Visitor v )  {
          v.visit(this );
}
```

使用这种方式，访问者对象可以收到每个被访问实例的一个引用，接下来就可以调用它们的公有方法获取数据、执行计算、生成报表，或在屏幕中绘制对象。如果某个类没有 accept 方法，可以子类化它并添加上该方法。

31.2　何时使用访问者模式

当想对数据执行一个操作，而该数据包含在不同接口的多个对象中时，应该考虑使用访问者模式。如果要对这些类执行很多不相关的操作，使用访问者模式也是有帮助的。在因为某种技术（或策略的）原因而无法获得源代码或不能更改源代码时，若想要向类库或框架中添加功能，访问者模式是一种有用的方法。在后面这些情形中，只需子类化框架中的类，并向每个子类添加 accept 方法。

另一方面，正如我们将要看到的，若不希望向程序添加很多个新类，访问者模式是一个很好的选择。

31.3　示例代码

这里考虑一个简单的、在组合模式中讨论过的 Employee 问题的“子集”。我们拥有一个简单的 Employee 对象，它保存着雇员的记录：雇员名、薪水、休假和病假天数。下面是该类的一个简单版本：

```
public class Employee        {
      int sickDays, vacDays;
      float salary;
      string name;
      public Employee(string name, float salary,
            int vDays, int sDays) {
      this.name = name;
      this.salary = salary;
      sickDays = sDays;
      vacDays = vDays;
}
//-----
public string getName() {
      return name;
```

```
}
public int getSickDays() {
      return sickDays;
}
public int getVacDays() {
      return vacDays;
}
public float getSalary() {
      return salary;
}
public virtual void accept(Visitor v) {
      v.visit(this);
}
}
```

注意，把accept方法放在该类中。假设要准备一份报表，统计今年迄今为止所有员工休假的天数。可以在客户端编写一些代码，统计每个雇员的getVacDays函数返回的结果，也可以把这一函数放到Visitor中。

由于C#是一种强类型的语言，基类Visitor需要为程序中每一种类型的类准备一个合适的抽象方法visit。在这个例子里，我们只有Employee类，所以抽象基类Visitor只是下列这种形式：

```
public abstract class Visitor   {
            public abstract void visit (Employee emp);
}
```

注意，这里对访问者如何使用每个客户类或抽象类，Visitor没有任何指示。实际上，可以编写出很多的访问者，它们对程序中的各个类做不同的事情，这里首先编写一个具体的Visitor类，它统计所有雇员的休假天数。

```
public class VacationVisitor :Visitor    {
      private int totalDays;
      //-----
      public VacationVisitor()   {
            totalDays = 0;
      }
      //-----
      public int getTotalDays() {
            return totalDays;
       }
      //-----
      public override void visit(Employee emp) {
           totalDays += emp.getVacDays();
      }
      //-----
      public override void visit(Boss bos){
            totalDays += bos.getVacDays();
      }
}
```

31.4 访问类

为了要计算总的休假天数，我们接下来要做的就是遍历一个雇员列表，访问每个雇员，让具体的Visitor给出总的天数。

```
for (int i = 0;  i< empls.Length; i++) {
            empls[ i] .accept(vac);       //get the employee
```

```
}
lsVac.Items.Add("Total vacation days=" +
      vac.getTotalDays().ToString());
```

这里重申一下每次访问要做的事情。

1. 循环遍历所有的雇员。
2. 访问者调用每个雇员的 accept 方法。
3. 雇员的实例调用访问者的 visit 方法。
4. 访问者取出休假天数并加到总数中。
5. 循环结束后主程序打印出总数。

31.5　访问几个类

有多个不同的类，它们都具有不同的接口，我们想在保持封装性的前提下从这些类中取出数据，这种情况下访问者模式变得非常有用。扩展我们的休假天数模型，引进一个新的 Employee 类型：Boss。这里进一步假设，在该公司中，Boss 还享受奖励休假（替代金钱）。这样，Boss 类就需要有两个额外的方法，分别设置和取得奖励休假的天数。

```
public class Boss : Employee     {
       private int bonusDays;
       public Boss(string name, float salary,
              int vdays, int sdays):
                     base(name, salary, vdays, sdays) { }
       public void setBonusDays(int bdays) {
              bonusDays = bdays;
       }
       public int getBonusDays() {
              return bonusDays;
       }
       public override void accept(Visitor v ) {
              v.visit(this);
       }
 }
```

向程序添加一个类时，也要把它添加到 Visitor 中，这样 Visitor 抽象模板就成了下面这种形式：

```
public abstract class Visitor   {
      public abstract void visit (Employee emp);
      public abstract void visit (Boss bos );
}
```

也就是说，编写的任何具体 Visitor 类都必须为 Employee 类和 Boss 类提供多态的 visit 方法。在计算休假天数时，要让 Boss 给出常规休假天数和奖励休假天数，所以这两个访问方法是不一样的。创建一个新类 bVacationVisitor，让它考虑这种差别。

```
public class bVacationVisitor :Visitor  {
       private int totalDays;
       public bVacationVisitor() {
              totalDays = 0;
       }
       //-----
       public override void visit(Employee emp) {
              totalDays += emp.getVacDays();
              try {
```

```
                    Manager mgr = (Manager)emp;
                    totalDays += mgr.getBonusDays();
                }
                catch(Exception ){}
            }
            //-----
            public override void visit(Boss bos) {
                totalDays += bos.getVacDays();
                totalDays += bos.getBonusDays();
            }
            //-----
            public int getTotalDays() {
                return totalDays;
            }
        }
```

注意，尽管本例中Boss类派生于Employee，但对Visitor类来说，它可以不必与Employee类相关，只要拥有accept方法就行了。这一点相当重要，在Visitor中为每个要访问的类实现了一个visit方法，而不能指望继承该方法，因为父类的visit方法是Employee中的方法，而不是Boss需要的visit方法，可以用图31.4说明这一点。

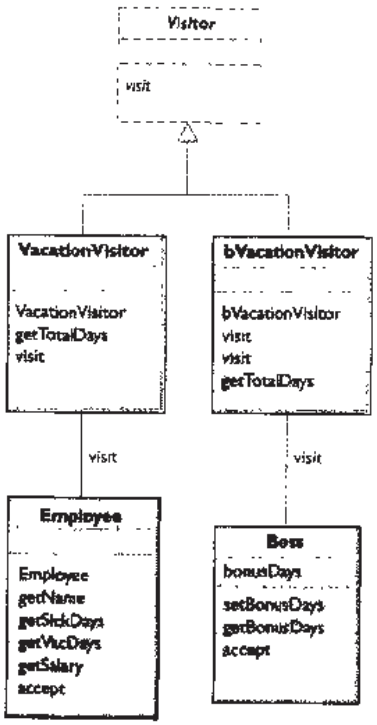


图31.4 两个Visitor类分别访问Boss类和Employee类

31.6 Boss也是Employee

图31.5显示了一个简单的应用程序，它分别对Employee集合和Boss集合访问。原来的VacationVisitor将Boss看做Employee，只给出普通休假天数，bVacationVisitor能求出两种休假的总天数。

```
for (int i = 0;  i< empls.Length; i++) {
        empls[ i] .accept(vac);       //get the employee
```

```
        empls[ i] .accept(bvac);
}
lsVac.Items.Add("Total vacation days=" +
            vac.getTotalDays().ToString());
lsVac.Items.Add("Total boss vacation days=" +
            bvac.getTotalDays().ToString());
```

用户点击 Compute 按钮时，显示出两行数据，表示计算出的两个总天数。

图 31.5　一个执行假期统计的应用程序

31.7　Visitor 的杂物箱操作

在前面的例子里，Visitor 类为每个要访问的类定义了一个 visit 方法，如下所示。

```
public abstract void visit (Employee emp);
public abstract void visit (Boss bos);
```

但是，如果子类化访问者类，并添加新的被访问类时，应该认识到有些 visit 方法没有满足派生类中的方法。这不是识别的对象类型的父类中方法的失败，而是提供了一种说明访问者默认行为的方式。

每个类必须以自己的实现方式重新定义 accept（v），这样，回程调用 v.visit（this）返回的 this 对象才具有正确的类型，而不是超类的类型。

假设在公司中引进另外一层管理：Manager。Manager 是 Employee 的子类，也具有以前 Boss 所保留的权利：拥有额外的假期。Boss 现在享有另一种奖励：股权。现在，如果我们运行同样的程序去计算休假天数，而不修正 Visitor 去查找 Manager，该程序会将 Manager 当做单纯的 Employee，只计算他们的常规假期，而不考虑额外的假期天数。但是，如果要不时地向应用程序添加子类，又想让访问者的操作能不加修改地继续运行，杂物箱式（Catch-All）的父类是一种好的解决办法。

有三种方法可以将新的 Manager 类加到访问者系统中。可以定义一个 ManagerVisitor 类，或者让 BossVisitor 同时处理 Boss 和 Manager 两个类。然而，有的情况下是不想频繁地改变 Visitor 的结构，这时候，可以在 EmployeeVisitor 类里为这种特殊情况检查 Employee。

```
public override void visit(Employee emp) {
      totalDays += emp.getVacDays();
      try {
            Manager mgr = (Manager)emp;
            totalDays += mgr.getBonusDays();
```

```
        }
        catch(Exception ){}
    }
```

尽管与正常的定义类相比，这种方式乍看起来有些“含混不清”，但它提供了一种在派生类里捕获特殊情况的方法，而且不需要编写一整套新的访问者程序体系，这种“杂物箱”方法在“Pattern Hatching”（Vlissides 1998）一书中有详细的讨论。

31.8 双分派

为了让Visitor能够运行，有一个方法实际上分派了两次，没有提到这一点，对访问者模式的讨论就是不完整的。某个Visitor调用了一个给定对象的多态方法accept，该accept方法回头调用Visitor的多态方法visit。这种双向调用允许向任何一个具有accept方法的类中添加更多的操作，因为我们编写的每个新的Visitor类都能使用这些类中的可用数据执行任何想要的操作。

31.9 为什么这样做

读者可能会问自己：“既然能直接调用getVacationDays方法，为什么还要如此麻烦？”通过使用这种“回调”方法，可以实现“双分派”，就不必要求被访问的对象是相同的或相关的类型。此外，使用这种回调方法，可以根据实际类的类型，调用Visitor中不同的访问方法。这是很难直接实现的。

另外，如果要访问的ArrayList对象列表是不同类型的集合，实际Visitor类中的不同版本的访问方法是解决该问题的惟一途径，而且不用专门检查每个类的类型。

31.10 访问一系列的类

将类实例传递给Visitor的调用程序必须知道所有已存在的要被访问的类实例，应该将它们保存在一个简单的结构里，比如数组或集合。另外一种方法是创建这些类的一个迭代器，并将它传递给Visitor。最后一种方法是，Visitor自己保存一个它要访问的对象列表。在我们这个简单的例子里，使用的是一个对象数组，使用其他的方法也同样能使程序很好地运行。

31.11 访问者模式的效果

当读者想封装从多个类实例中取出数据的操作，使用访问者模式是有帮助的。“Design Patterns”指出，访问者模式不需要修改类就能给它提供额外的功能。我们更愿意把它说成：访问者模式能对类集合添加功能，并封装了它所使用的方法。

然而，访问者不是“魔法”，不能从类中取得私有数据，它只限于从公有方法获得数据。这会迫使你提供原本没有的公有方法。但是，访问者能从不相关类的集合中获取数据，用这些数据向用户程序提供一个全局范围的计算结果。

使用访问者模式很容易向程序添加新操作，因为Visitor包含的是代码而不是一个个单独的类。另外，Visitor能将相关的操作收集到一个独立的类中，你不用去修改类（或派生类）来添加这些操作，这会使程序容易编写和维护。

在程序的发展阶段，使用Visitor没有太大的帮助，因为每次添加被访问的新类，都要向抽向的Visitor类添加一个抽象的visit操作，而且要在每个已完成的具体Visitor类中为该类添加一个实现。当程序发展到一定程度，不再增加多个新类时，使用Visitor可显示出强大的添加功能。

访问者模式用在组合系统里也很有效，刚才举例的老板－雇员系统就是一个很好的组合例子，就像我们在组合模式那一章用到的那样。

31.12　思考题

一个投资公司的客户记录包括每个投资人拥有的每支股票或其他投资手段所对应的一个对象。对象包含了一个股票的买入、卖出和股息分配的历史记录。设计一个访问者模式，生成一年中关于股票买卖的年终纯收益或损失的一份报告。

31.13　随书附带光盘中的程序

Visitor Example	\Visitor\

参考文献

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahtking, L., and Angel, S. *A Pattern Language*, Oxford University Press, New York, 1977.

Alpert, S. R., Brown, K., and Woolf, B. *The Design Patterns Smalltalk Companion*, Addison-Wesley, Reading, MA, 1998.

Arnold, K., and Gosling, J. *The Java Programming Language*, Addison-Wesley, Reading, MA, 1996.

Booch, G., Jacobson, I., and Rumbaugh, J. *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA, 1999.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *A System of Patterns*, John Wiley and Sons, New York, 1996.

Cooper, J. W. *Java Design Patterns: A Tutorial*, Addison-Wesley, Boston, MA, 2000.

_____.*Principles of Object-Oriented Programming in Java 1.1*, Coriolis (Ventana), 1997.

_____.*Visual Basic Design Patterns: VB6 and VB.NET*, Addison-Wesley, Boston, MA, 2002.

Coplien, J.O. *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading, MA, 1992.

Coplien, J.O., and Schmidt, D. C. *Pattern Languages of Program Design*, Addison-Wesley, Reading, MA, 1995.

Fowler, M., with K. Scott. *UML Distilled*, Addison-Wesley, Reading, MA, 1997.

Gamma, E. *Object-Oriented Software Development Based on BT+: Design Patterns, ClassLibrary, Tools* (in German), Springer-Verlag, Berlin, 1992.

Gamma, E., Helm, T., Johnson, R., and Vlissides, J. *Design Patterns: Abstraction and Reuse of Object Oriented Design*. Proceedings of ECOOP' 93, 405-431.

_____. *Design Patterns. Elements of Reusable Software*, Addison-Wesley, Reading, MA, 1995.

Grand, M. *Patterns in Java*, Volume 1, John Wiley & Sons, New York, 1998.

Krasner, G. E., and Pope, S. T. "A cookbok for using the Model-View-Controller user interface paradigm in Smalltalk-80." *Journal of ObjectOriented Programming* I(3),1988.

Kurata, D. "Programming with objects." *Visual Basic Programmer's Journal*, June 1998.

Pree, W. *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

Riel, A. J. *Object-Oriented Design Heuristics*, Addison-Wesley, Reading, MA, 1996.

Vlissides, J. *Pattern Hatching: Design Patterns Applied*, Addison-Wesley, Reading, MA. 1998.

参考文献

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahtking, L., and Angel, S. *A Pattern Language*, Oxford University Press, New York, 1977.

Alpert, S. R., Brown, K., and Woolf, B. *The Design Patterns Smalltalk Companion*, Addison-Wesley, Reading, MA, 1998.

Arnold, K., and Gosling, J. *The Java Programming Language*, Addison-Wesley, Reading, MA, 1996.

Booch, G., Jacobson, I., and Rumbaugh, J. *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA, 1999.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *A System of Patterns*, John Wiley and Sons, New York, 1996.

Cooper, J. W. *Java Design Patterns: A Tutorial*, Addison-Wesley, Boston, MA, 2000.

_____.*Principles of Object-Oriented Programming in Java 1.1*, Coriolis (Ventana), 1997.

_____.*Visual Basic Design Patterns: VB6 and VB.NET*, Addison-Wesley, Boston, MA, 2002.

Coplien, J.O. *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading, MA, 1992.

Coplien, J.O., and Schmidt, D. C. *Pattern Languages of Program Design*, Addison-Wesley, Reading, MA, 1995.

Fowler, M., with K. Scott. *UML Distilled*, Addison-Wesley, Reading, MA, 1997.

Gamma, E. *Object-Oriented Software Development Based on BT+: Design Patterns, ClassLibrary, Tools* (in German), Springer-Verlag, Berlin, 1992.

Gamma, E., Helm, T., Johnson, R., and Vlissides, J. *Design Patterns: Abstraction and Reuse of Object Oriented Design*. Proceedings of ECOOP' 93, 405-431.

_____. *Design Patterns. Elements of Reusable Software*, Addison-Wesley, Reading, MA, 1995.

Grand, M. *Patterns in Java*, Volume 1, John Wiley & Sons, New York, 1998.

Krasner, G. E., and Pope, S. T. "A cookbok for using the Model-View-Controller user interface paradigm in Smalltalk-80." *Journal of ObjectOriented Programming* I(3),1988.

Kurata, D. "Programming with objects." *Visual Basic Programmer's Journal*, June 1998.

Pree, W. *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

Riel, A. J. *Object-Oriented Design Heuristics*, Addison-Wesley, Reading, MA, 1996.

Vlissides, J. *Pattern Hatching: Design Patterns Applied*, Addison-Wesley, Reading, MA. 1998.