Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

Professional ASP.NET Design Patterns

ASP.NET设计模式

Microsoft首席Program Manager Scott Hanselman (Microsoft RD, MVP)推荐作序

(美) Scott Millett 著
杨明军 译

清华大学出版社

ASP.NET 设计模式

(美) Scott Millett 著
杨明军 译

清华大学出版社
北 京

Professional ASP.NET Design Patterns
EISBN：978-0-470-29278-5
Copyright © 2010 by Wiley Publishing, Inc.
All Rights Reserved. This translation published under license.

本书中文简体字版由 Wiley Publishing, Inc.授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字：

本书封面贴有 Wiley 公司防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目(CIP)数据

ASP.NET 设计模式/(美)米里特(Millett, S.)著；杨明军 译 —北京：清华大学出版社，2011.11
书名原文：Professional ASP.NET Design Patterns
ISBN 978-7-302-26702-7
I. A… Ⅱ. ①米… ②杨… Ⅲ. 网页制作工具－程序设计 Ⅳ. TP393.092

中国版本图书馆 CIP 数据核字(2011)第 181074 号

责任编辑：王 军 于 平
装帧设计：孔祥丰
责任校对：成凤进
责任印制：
出版发行：清华大学出版社 地 址：北京清华大学学研大厦 A 座
http://www.tup.com.cn 邮 编：100084
社 总 机：010-62770175 邮 购：010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经 销：
开 本：185×260 印 张：43.5 字 数：1167 千字
版 次：2011 年 11 月第 1 版 印 次：2011 年 11 月第 1 次印刷
印 数：1~4000
定 价：79.80 元

产品编号：

前　　言

本书将向您展示如何在实际的 ASP.NET 应用程序中发挥设计模式和核心设计原则的威力。本书的目标是向开发者讲解能够帮助其成为更好的程序员的面向对象编程基础、设计模式、原则及方法学。设计模式和原则支持松散耦合、高内聚的代码，而这将提升代码的可读性、灵活性和可维护性。每一章内容关注企业 ASP.NET 应用程序中的一层，并展示如何利用那些经过实践证明的模式、原则和最佳实践来解决问题并改进代码设计。此外，本书使用一个专业级的、完整的研究案例来讲解如何在实际的网站中运用最佳实践设计模式和原则。

读者对象

本书是为那些熟悉.NET 框架但希望了解如何改进编码方式以及如何运用设计模式、设计原则和最佳实践来提高代码的可维护性和适应性的 ASP.NET 开发者而写的。那些以前已经体验过设计模式的读者可能希望跳过本书的第Ⅰ部分，这部分介绍了 GoF 提出的设计模式以及其他常见设计原则，包括 S.O.L.I.D 原则和 Martin Fowler 的企业设计模式。所有的代码示例均采用 C#语言编写，但这些概念可以非常轻松地用于 VB.NET。

主要内容

本书涵盖了开发企业级 ASP.NET 应用程序的知名模式和最佳实践。本书用到的模式可以用于从 ASP.NET 1.0 到 ASP.NET 4.0 的任何版本。不必管模式本身所用的语言，可以将模式用于任何面向对象编程语言。

结构安排

本书既可以作为一个分步推进的指南，也可以作为闲暇时随意翻阅的常备参考书。本书分为 3 个部分。第Ⅰ部分介绍了模式和设计原则。第Ⅱ部分讲解如何在 ASP.NET 应用程序的不同层中使用模式和原则。第Ⅲ部分展示了一个完整的研究案例，用来演示本书涵盖的多个模式。既可以在阅读研究案例之前通读各章内容；也可以首先阅读研究案例，然后在涉及具体的模式和原则时再回过头来查阅第Ⅱ部分以获取更详细的内容，这种紧密结合实际的方法可能会让学习过程变得更加轻松。

第Ⅰ部分　模式与设计原则

本书第Ⅰ部分首先介绍设计模式概念、企业模式及设计原则，包括 S.O.L.I.D.设计原则。

第 1 章：成功应用程序的模式

该章研究了专业开发者为何需要理解设计模式和原则，以及(更重要的是)如何在实际的企业级应用程序中加以利用。该章讲解了 GoF 设计模式的起源、它们在当今世界中的关联性以及与具体编程语言脱钩。然后浏览了一些常见设计原则和 S.O.L.I.D.设计原则，最后描述了 Fowler 的企业模式。

第 2 章：剖析模式的模式

该章介绍了使用模式模板所需的实用知识以及如何利用设计模板来阅读 GoF 设计模式。然后将教您如何理解设计模式分类，并讲解如何选择和应用设计模式。最后给出了一个示例来演示如何重构现有的代码，以便使用设计模式和原则来提升可维护性。

第Ⅱ部分　剖析 ASP.NET 应用程序：学习并应用模式

本书的第Ⅱ部分演示如何将前两章介绍的模式和原则运用到企业级 ASP.NET 应用程序的各个层次中。

第 3 章：应用程序分层与关注点分离

该章描述了分层设计与传统的 ASP.NET Web 表单代码隐藏模型相比所具有的优势。该章继续讲解逻辑分层和应用程序关注点分离的概念。然后定义了企业级 ASP.NET 应用程序中各个不同层次的职责，这部分的其他几章将讨论这些层次。该章最后是一个练习，将一个 Smart UI 反模式按照分层体系结构方法进行重构。

第 4 章：业务逻辑层：组织

该章涵盖了为组织业务逻辑层而设计的模式。该章首先描述了 Transaction Script 模式；然后是 Active Record 模式，并利用一个使用 Castle Windsor 项目的练习来演示该模式；最后一个模式是 Domain Model 模式，用 NHibernate 练习进行演示。该章最后评论了领域驱动设计(domain-driven design，DDD)方法学，以及如何运用该方法学将精力集中在业务逻辑而非基础设施。

第 5 章：业务逻辑层：模式

第 5 章与第 4 章类似，仍然介绍业务层，但该章关注的是构建对象时可以使用的模式和原则，以及如何确保构建可伸缩、可维护的应用程序。该章涉及的模式包括 Factory、Decorator、Template、State、Strategy 和 Composite。所涉及的企业模式包括 Specification 和 Layer Supertype。最后介绍了一些能够改进代码可维护性和灵活性的设计原则，包括 Dependency Injection、Interface Segregation 和 Liskov Substitution 原则。

第 6 章：服务层

该章介绍了服务层在企业 ASP.NET 应用程序中扮演的角色。该章首先简要地介绍了 Service

Oriented Architecture 及其必要性。然后讨论了 Façade 设计模式。接着讨论了 Document Message、Request-Response、Reservation 和 Idempotent 模式等 Messaging 模式。最后给出了一个 WCF 实体的练习，来演示本章涵盖的所有模式。

第 7 章：数据访问层

如何利用数据存储来使业务对象状态持久化是应用程序体系结构的关键部分。该章将学习该层中使用的设计模式，以及如何将它们整合在一起。这里演示了两种帮助组织持久层的数据访问策略：Repository 和 Data Access Object。接着，介绍了一些有助于优雅地满足数据访问需求的企业模式和原则，包括 Lazy Loading、Identity Map、Unit of Work 和 Query Object。之后介绍了 Object Relational Mapper 以及它们解决的问题。最后，给出了一个企业 Domain Driven 练习，它的 POCO 业务实体同时使用了 NHibernate 和 MS Entity Framework。

第 8 章：表示层

该章介绍了一些为了组织表示层逻辑并让其与应用程序中的其他层保持分离状态的模式。该章首先解释如何使用 Structure Map 和 Inversion of Control 容器将松散耦合的代码连接起来。然后描述了几种表示层模式，包括利用 Model-View-Presenter 模式和 ASP.NET Web 表单实现视图，利用 Command 和 Chain of Responsibility 模式实现 Front Controller 表示模式，以及利用 ASP.NET MVC 框架和 Windsor 的 Castle Monorail 框架来实现 Model-View-Controller 模式。最后讨论的表示层模式是 ASP.NET Web 表单中使用的 PageController。该章最后讲解一种可与组织模式一起使用的 ViewModel 模式，以及如何利用 AutoMapper 自动地把领域实体映射到 ViewModel。

第 9 章：用户体验层

在第Ⅱ部分的最后一章介绍用户体验层。该章首先讲解什么是 Ajax 及其所依赖的技术。然后讲解了 JavaScript 库，说明如何使用 jQuery 等强大代码库来简化 JavaScript 处理。该章主要描述了一些常见的 Ajax 模式：Ajax Periodic Refresh 和 Timeout 模式，用于维护历史的 Unique URL 模式，用于实现客户端数据绑定的 Jtemplate 模式，以及 Ajax Predictive Fetch 模式。

第Ⅲ部分　案例研究：在线电子商务商店

本书最后一部分使用一个完整的示例应用程序来演示在第Ⅱ部分中介绍的众多模式。

第 10 章：需求和基础设施

这是关于案例研究的第 1 章，介绍了将在其余 4 章中构建的 Agatha 电子商务商店。该章描述了该网站的需求，以及将要用到的基础设施和总体架构。表示层采用 ASP.MVC，中间层的组织采用领域模型，利用 NHibernate 使业务实体持久化并从数据库中检索业务实体。

第 11 章：创建商品目录

该章构建了商店的商品目录浏览功能。大量使用 jQuery 来提供丰富的 Web 2.0 观感。利用 JSON 来实现控件和 ASPX 视图之间的通信以提供 Ajax 功能。使用 ViewModel 为控件提供扁平的领域视图。采用 AutoMapper 自动地把领域实体转换成 ViewModel。

第 12 章：实现购物车

该章实现了顾客购物车。使用顾客 Cookie 来存放购物车内容摘要，创建一项服务将访问 Cookie 存储的逻辑抽象出来。为了保持 Web 2.0 观感，购物车上执行的所有动作均通过 Ajax 调用进行。

第 13 章：顾客会员

该章解决顾客会员资格和身份验证问题。使用 ASP.NET 用户凭据提供者进行就地身份验证；但也使用了第二种身份验证方法，让顾客使用其现有的基于 Web 的账号(比如 Facebook 和 Google 账号)来进行身份验证。本章还会开发顾客账号屏幕。

第 14 章：订购与支付

在案例研究练习的最后一章中，将研究该网站的支付和结账功能。本章选中的支付机制是 PayPal，但把逻辑代码抽象出来，这样就可以轻易地将其替换成任何其他支付手段。最后向顾客的账号部分中添加订单历史记录功能。

源代码

在阅读本书示例的过程中，可以选择采用手工方式录入所有代码，也可以选择本书所附的源代码文件。本书中所用的所有源代码均可以在 www.wrox.com 和 http://www.tupwk.com.cn/downpage 下载。在该网站，搜索本书的书名(可以通过搜索栏或使用书名列表)，然后在本书详细信息页面中单击 Download Code 链接来获取本书的源代码。

> 因为许多书籍都有着相似的书名，所以最简单的方式是按照 ISBN 搜索。本书的 ISBN 是 978-0-470-29278-5。

在下载代码之后，使用压缩工具解压。或者，可以打开 Wrox 主代码下载页面 http://www.wrox.com/dynamic/books/download.aspx，查看本书以及所有其他 Wrox 书籍的代码。

勘误表

我们尽力确保本书正文及代码正确无误。但人无完人，错误在所难免。如果发现本书中有错误之处，如拼写错误或代码错误，我们将非常感谢您的反馈。通过提交勘误信息，可以让其他读者免于枉费数小时宝贵时间，还能帮助我们提供更高品质的信息。

要查找本书的勘误页，请打开 www.wrox.com 网站，并使用搜索框或书名列表查找本书的书名。然后，在本书详细信息页面上，单击 Book Errata 链接。在打开的页面上，可以浏览已经由读者为本书提交的以及由 Wrox 编辑通告的所有勘误信息。还可以在 http://www.wrox.com/misc-pages/booklist.shtml 找到完整的包含每本书勘误信息链接的书籍列表。

如果没有在 Book Errata 页找到自己发现的错误，请打开 http://www.wrox.com/contact/techsupport.shtml 页面，填写其中的表单，将您发现的错误发送给我们。我们将检查该信息，如果确认存在该错误，我们将会向本书的勘误页发送一条消息并在本书后续版中纠正该问题。

P2P.wrox.com 网站

如果希望与作者和同行讨论，请加入 P2P 论坛 http://p2p.wrox.com。论坛属于基于 Web 的系统，可供提交关于 Wrox 书籍和相关技术的消息，并与其他读者和技术用户交流。论坛提供了一项订阅功能，当论坛上您所感兴趣的主题有新帖时，系统会通过电子邮件通知您。Wrox 作者、编辑、其他行业专家及同行读者都会出现在这些论坛上。

在 http://p2p.wrox.com 网站上，可以找到多个不同的论坛，不仅能够帮助您阅读本书，还能帮助您开发自己的应用程序。要加入论坛，只需要遵循以下步骤即可：

(1) 打开 http://p2p.wrox.com 并单击 Register 链接。

(2) 阅读使用条款并单击 Agree。

(3) 填写加入论坛时必需的信息以及任何您希望提供的可选信息，然后单击 Submit。

(4) 您将收到一封电子邮件，里面描述了如何验证自己的账号并完成加入过程。

虽然不加入 P2P 论坛也能阅读论坛中的消息，但要提交您自己的消息，必须加入。

一旦加入论坛，就可以提交新信息并回复其他用户提交的消息。可以在任何时间阅读 Web 上的消息。如果希望系统将特定论坛的新消息通过电子邮件发送给您，那么在论坛列表中的论坛名称旁单击 Subscribe to this Forum 图标。

有关如何使用 Wrox P2P 的更多信息，阅读 P2P FAQ 以获得有关论坛软件如何运行以及许多有关 P2P 和 Wrox 书籍常见问题的答案。要阅读 FAQ，在 P2P 页面上单击 FAQ 链接。

目　　录

第Ⅰ部分　模式与设计原则

第 1 章　成功应用程序的模式 ······3
1.1　设计模式释义 ······3
1.1.1　起源 ······4
1.1.2　必要性 ······4
1.1.3　有效性 ······4
1.1.4　局限性 ······5
1.2　设计原则 ······5
1.2.1　常见设计原则 ······5
1.2.2　S.O.L.I.D.设计原则 ······6
1.3　Fowler 的企业设计模式 ······7
1.3.1　分层 ······7
1.3.2　领域逻辑模式 ······7
1.3.3　对象关系映射 ······8
1.3.4　Web 表示模式 ······9
1.3.5　基本模式、行为模式和结构模式 ······9
1.4　其他有名的设计实践 ······10
1.4.1　测试驱动设计 ······10
1.4.2　领域驱动设计 ······10
1.4.3　行为驱动设计 ······10
1.5　小结 ······11

第 2 章　剖析模式的模式 ······13
2.1　如何阅读设计模式 ······13
2.1.1　GoF 模式模板 ······13
2.1.2　简化模板 ······14
2.2　设计模式分组 ······14
2.2.1　创建型 ······14
2.2.2　结构型 ······15
2.2.3　行为型 ······15
2.3　如何选择和运用设计模式 ······16
2.4　快速模式示例 ······17
2.4.1　根据设计原则进行重构 ······19
2.4.2　根据 Adapter 模式进行重构 ······21
2.4.3　利用企业模式 ······24
2.5　小结 ······25

第Ⅱ部分　剖析 ASP.NET 应用程序：学习并应用模式

第 3 章　应用程序分层与关注点分离 ······29
3.1　应用程序体系结构与设计 ······29
3.1.1　反模式：智能 UI ······29
3.1.2　分离关注点 ······35
3.2　小结 ······51

第 4 章　业务逻辑层：组织 ······53
4.1　理解业务组织模式 ······53
4.1.1　Transaction Script ······53
4.1.2　Active Record ······55
4.1.3　Domain Model ······65
4.1.4　Anemic Domain Model ······86
4.1.5　领域驱动设计 ······88
4.2　小结 ······91

第 5 章　业务逻辑层：模式 ······93
5.1　应用设计模式 ······93
5.1.1　Factory Method 模式 ······93
5.1.2　Decorator 模式 ······97
5.1.3　Template Method 模式 ······103
5.1.4　State 模式 ······107
5.1.5　Strategy 模式 ······113
5.2　应用企业模式 ······117
5.2.1　Specification 模式 ······117
5.2.2　Composite 模式 ······119

5.2.3 Layer Supertype 模式 …… 124
5.3 应用设计原则 …… 127
5.3.1 依赖倒置原则和依赖注入模式 …… 127
5.3.2 接口分离原则 …… 133
5.3.3 里氏替换原则 …… 137
5.4 小结 …… 147

第 6 章 服务层 …… 149
6.1 服务层介绍 …… 149
6.1.1 SOA …… 149
6.1.2 SOA 的 4 项信条 …… 152
6.1.3 Facade 设计模式 …… 152
6.2 应用 Messaging 模式 …… 153
6.2.1 Document Message 和 Request-Response 模式 …… 154
6.2.2 Reservation 模式 …… 155
6.2.3 Idempotent 模式 …… 156
6.3 SOA 示例 …… 156
6.3.1 领域模型和资源库 …… 157
6.3.2 服务层 …… 166
6.3.3 客户端代理 …… 180
6.3.4 客户端 …… 183
6.4 小结 …… 187

第 7 章 数据访问层 …… 189
7.1 DAL 介绍 …… 189
7.2 数据访问策略 …… 189
7.2.1 Repository 模式 …… 190
7.2.2 Data Access Objects 模式 …… 191
7.3 数据访问模式 …… 191
7.3.1 Unit of Work 模式 …… 191
7.3.2 数据并发控制 …… 198
7.3.3 Lazy Loading 和 Proxy 模式 …… 201
7.3.4 Identity Map 模式 …… 206
7.3.5 Query Object 模式 …… 208
7.4 使用对象关系映射器 …… 218
7.4.1 NHibernate …… 219
7.4.2 MS Entity Framework …… 219
7.4.3 ORM 代码示例 …… 219
7.5 小结 …… 280

第 8 章 表示层 …… 283
8.1 反转控制 …… 283
8.1.1 Factory Method 设计模式 …… 283
8.1.2 Service Locator …… 285
8.1.3 IoC 容器 …… 286
8.1.4 StructureMap …… 286
8.2 Model-View-Presenter …… 290
8.3 Front Controller …… 313
8.3.1 Command 模式 …… 314
8.3.2 Chain of Responsibility 模式 …… 336
8.4 Model-View-Controller …… 344
8.4.1 ViewModel 模式 …… 344
8.4.2 ASP.NET MVC 框架 …… 345
8.4.3 利用 AutoMapper 映射 ViewModel …… 357
8.4.4 Castle MonoRail …… 362
8.5 Page Controller 模式 …… 369
8.6 小结 …… 370

第 9 章 用户体验层 …… 371
9.1 什么是 AJAX …… 371
9.2 使用 JavaScript 库 …… 372
9.3 理解 AJAX 模式 …… 372
9.3.1 Periodic Refresh 和 Timeout …… 372
9.3.2 Unique URL …… 390
9.3.3 利用 JavaScript Template 实现数据绑定 …… 390
9.3.4 Predictive Fetch …… 408
9.4 小结 …… 414

第III部分 案例研究：在线电子商务商店

第 10 章 需求和基础设施 …… 417
10.1 Agatha 服装店需求 …… 417
10.1.1 Product Catalog 和 Basket 截屏 …… 418
10.1.2 顾客账号屏幕 …… 420
10.1.3 结账屏幕 …… 422
10.1.4 缓存和日志 …… 423
10.2 架构 …… 423
10.3 小结 …… 443

第 11 章　创建商品目录 445
11.1　创建产品目录 445
11.1.1　Product Catalog 模型 445
11.1.2　Product Catalog 数据表 450
11.1.3　Product Catalog 资源库 451
11.1.4　Product 服务 465
11.1.5　控制器 480
11.1.6　Product Catalog 视图 490
11.1.7　设置 IoC 513
11.2　小结 516
第 12 章　实现购物车 519
12.1　实现购物车 519
12.1.1　Basket 领域模型 519
12.1.2　创建购物车数据表 529
12.1.3　NHibernate 映射 530
12.1.4　购物车服务 533
12.1.5　购物车控制器和购物车视图 543
12.2　小结 565
第 13 章　顾客会员 567
13.1　顾客会员 567
13.1.1　Customer 模型 568
13.1.2　Customer 数据表 573
13.1.3　Customer NHibernate 映射 573
13.1.4　Customer 服务 576
13.1.5　身份验证服务 585
13.1.6　Customer 控制器 593
13.1.7　Account 控制器 597
13.1.8　顾客关系视图 607
13.1.9　身份验证视图 611
13.2　小结 617
第 14 章　订购和支付 619
14.1　结账 619
14.1.1　Order 模型 620
14.1.2　Order 数据表 635
14.1.3　Order NHibernate 映射 636
14.1.4　Order 服务 639
14.1.5　利用 PalPay 进行支付 648
14.1.6　Order、Payment 与 Checkout 控制器 657
14.1.7　Order 和 Checkout 视图 666
14.2　小结 676

第 1 章 成功应用程序的模式

本章内容：

- 介绍“四人组”(Gang of Four，GoF)设计模式
- 概述一些常见的设计原则和 SOLID 设计原则
- 描述 Fowler 企业模式

约翰 • 列侬曾经写道“没有问题，只有出路”。现在，虽然据我所知列侬先生从未从事 ASP.NET 编程，但是在软件开发甚至人性方面(但这超出了本书的研究范畴)，他所说的这句话却极为中肯。作为软件开发者，我们的工作涉及解决问题，而之前已经有其他开发者曾经无数次不得不解决这些问题，虽然这些问题披着各式的外衣。自从面向对象编程方法出现以来，人们已经发现、命名和归类许多模式、原则和最佳实践。了解了这些模式和常见解决方法词汇表，就可以着手将复杂问题分解，将不同部分封装起来，并采用经过检验的可信解决方案，以一种统一的方式来开发应用程序。

本书的目标是介绍可以运用到 ASP.NET 应用程序中的设计模式、原则和最佳实践。本质上，对于设计模式和原则，其语言可以是不可知的，因此可以把从本书学到的知识运用到 WinForm、WPF 和 Silverlight 应用程序以及其他优秀的面向对象语言。

本章将介绍设计模式的定义、起源以及学习它们的重要性。设计模式的基础是坚实的面向对象设计原则，本章将以 Robert Martin 的 S.O.L.I.D.设计原则为例来讲解这一点。本章还将介绍 Martin Fowler 的 *Patterns of Enterprise Application Architecture* 一书中提出的一些更高级的模式。

1.1 设计模式释义

设计模式是高层次的、抽象的解决方案模板。可以将这些模式视为解决方案的蓝本而不是解决方案本身。从中无法找到一种可以简单地运用到应用程序中的框架；相反，通常是通过重构自己的代码并将问题泛化来实现设计模式。

设计模式不仅适用于软件开发领域，从工程到建筑的所有领域都能够找到它的身影。实际上，模式的思想是由著名建筑大师 Christopher Alexander 在 20 世纪 70 年代引入的，目的是为设计讨论构

建一张公共的词汇表。他写道：

本语言的元素是被称为模式的实体。每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心，这样就可以一次又一次地使用该方案而不必做重复劳动。

Alexander 的观点不仅适用于建筑和城市规划，也适用于软件设计。

1.1.1　起源

当今软件体系结构中比较流行的设计模式起源于程序员多年使用面向对象编程语言而积累的经验和知识。*Design Patterns: Elements of Reusable Object-Oriented Software*(又被亲切地称为 *Design Patterns Bible*，即“设计模式圣经”)一书收录了绝大多数最常见的模式。这本书是由 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 四人合著而成，而他们常被称为“四人组”(GoF)。

他们收录了 23 种设计模式并将它们归纳为 3 组。

- 创建型模式：处理对象构造和引用。
- 结构型模式：处理对象之间的关系以及它们之间如何进行交互以形成更大的复杂对象。
- 行为型模式：处理对象之间的通信，特别是在责任和算法方面。

所有模式均采用模板表达，这样读者就可以学习如何解读模式并加以运用。我们将在第 2 章中讲解使用设计模式模板所需的实用知识，并简要介绍将要在本书剩余部分学到的所有模式。

1.1.2　必要性

模式对于软件设计和开发而言至关重要。不管是在设计阶段解决问题，还是在源代码中，模式都支持通过共享的词汇表来表达思想。模式提倡使用优秀的面向对象软件设计，这是因为它们是围绕可靠的面向对象设计原则而构建的。

模式是描述复杂问题的解决方案的有效方式。如果具备设计模式的牢固知识，就可以与团队中的其他成员快速、顺畅地沟通，而不必纠结于底层的实现细节。

模式是语言不可知的，因此，可以将它们转换成其他面向对象语言。通过学习模式而获得的知识将能够运用于具体编程时采用的任何优秀的面向对象语言。

1.1.3　有效性

设计模式的使用价值和终极价值在于，它们是可靠的、经过验证的解决方案，它们的有效性毋庸置疑。如果是一名经验丰富的开发者并且已经采用.NET 或其他面向对象编程语言从事编程工作多年，或许发现自己已经在使用“GoF”书中提及的一些设计模式。但如果能够辨别正在使用的模式，那么与其他同样理解模式的开发者的沟通效率就会高得多，他们会理解您的解决方案的结构。

设计模式的宗旨就是重用解决方案。当然，并非所有问题都是一模一样的，但如果能够将一个问题分解，并找出它与以前解决过的问题之间的相似之处，就可以运用这些解决方案。经过数十年的面向对象编程实践，所遇到的大多数问题在之前已经被解决过无数次，而且会有一种模式可用于帮助您实现解决方案。即使您认为自己遇到的问题是独特的，也应该可以通过将其分解成若干基本要素，将其泛化至一定程度，从而找出一种合适的解决方案。

设计模式的名称非常有用，这是因为它反映出该模式的行为和目的，并为人们在集思广益讨论

解决方案时提供常用的词汇表。使用模式名称讨论，要比详细地讨论其具体实现如何工作更加容易。

1.1.4　局限性

设计模式并非银弹。您必须充分理解首要解决的问题，将其泛化，然后运用某个适合它的模式。但是，并非所有问题都需要设计模式。设计模式确实能够帮助人们将复杂的问题变得简单，但是同样也能够让本来简单的问题变得复杂。

在阅读有关模式的书籍之后，许多开发者都掉进了一个陷阱：试图把设计模式运用到所做的每件事情，但最终取得的效果却与设计模式初衷(也就是让事情变得简单)相反。前面曾经说过，运用模式的较好方法是，通过识别正在试图解决的基本问题，来寻找适合它的解决方案。本书将帮助您识别什么时候以及如何使用设计模式，继而从 ASP.NET 的角度来讨论如何实现。

并非总要使用设计模式。如果您已经为某个问题找到一种简单(但又不是过于简陋的)、清晰而且可维护的解决方案，那么倘若该方案并不属于 GoF(四人组)设计模式中的某一个模式，也不要责备自己。否则，就会让自己的设计过于复杂。

这段有关设计模式的讨论此时给人的感觉可能相当模糊，但是在继续阅读本书的过程中，您将学习每个设计模式打算解决的问题类型，并了解如何在 ASP.NET 中实现这些设计模式，然后就能够将这些模式运用到自己的应用程序中。

1.2　设计原则

设计原则构成了设计模式赖以构建的基础。它们要比设计模式更加基础。通过遵循经过验证的设计原则，自己的代码基会变得更加灵活、更能够适应变化，而且可维护性更佳。下面将简要地介绍一些比较著名的设计原则以及一组被称为 S.O.L.I.D.原则的原则。在本书后面将更为深入地了解这些原则，然后在 ASP.NET 中实现它们及最佳实践。

1.2.1　常见设计原则

如同设计模式一样，有一些常见的设计原则历经多年已经变成了最佳实践，并构成了企业级软件和可维护软件可以赖以构建的基础。下面将预览一些广为人知的原则。

1. 简约原则(KISS)

软件编程领域普遍存在的一个问题是需要把解决方案过度复杂化。KISS 原则的目标就是让代码保持简洁但不要过于简陋，从而避免引入任何不必要的复杂度。

2. 不要重复自己(DRY)

DRY 原则的目的是通过将公用的部分抽离出来放在一个单独的地方从而避免重复系统中的任何部分。这个原则不仅涉及代码而且包括系统中重复的任何逻辑。最终，系统中的任何一部分知识都应该只有一种表示形式。

3. 讲述而不要询问(Tell, Don't Ask)

“讲述而不要询问”原则体现了封装以及将责任指派到正确的类这两个思想。这个原则要求，应该告诉对象您希望它们执行什么动作，而不是询问有关该对象状态的问题然后由您自己决定希望执行什么动作。这个原则有助于匹配责任并避免类之间的紧密耦合。

4. 您不需要它(YAGNI)

YAGNI 原则指的是只需要将应用程序必需的功能包含进来，而不要试图添加任何其他您认为可能需要的功能。测试驱动开发(TDD)就是一种坚持 YAGNI 原则的设计方法学。TDD 的宗旨就是编写测试来验证系统的功能，然后只需要编写可让测试通过的代码即可。本章稍后部分将讨论 TDD。

5. 分离关注点(SoC)

SoC 这一过程将软件分解为多项不同的功能，每项功能封装了可供其他类使用的唯一行为和数据。通常，一个关注点代表类的一项功能或行为。将程序划分成若干独立职责的做法显著提高了代码的重用程度、维护性和可测试性。

在本书剩余部分将追溯这些设计原则，这样您就能够了解它们是如何实现的，以及如何帮助建立干净的、可维护的面向对象系统。下面将要看到的是 S.O.L.I.D.设计原则分类中收集的设计原则。

1.2.2 S.O.L.I.D.设计原则

S.O.L.I.D.设计原则是一组针对面向对象设计的最佳实践。GoF 设计模式均以这样或那样的形式遵守这些原则。术语 S.O.L.I.D.来自于 Robert C. Martin(朋友们亲切地称呼他 Bob 大叔)的著作 *Agile Principles, Patterns, and Practices in C#*中收集的 5 个设计原则的名称的首字母。下面将依次介绍这些设计原则。

1. 单一责任原则(SRP)

SRP 原则与 SoC 原则保持高度一致。它要求每个对象只应该为一个元素而改变而且只有一个职责关注点。遵循这个原则，就可以避免单体类(就像是软件领域的瑞士军刀)设计问题。使每个类均保持简洁，就可以提升系统的可读性和可维护性。

2. 开放封闭原则(OCP)

OCP 原则要求类对于扩展应该是开放的，而对于修改应该是封闭的，这样应该就能够在不改变类的内部行为的情况下添加新功能并扩展类。这个原则努力避免破坏已有的类以及其他依赖它的类，因为这会在应用程序中造成 bug 和错误的涟漪效应。

3. 里氏替换原则(LSP)

LSP 原则指出应该能够使用任何继承类来替代父类并且让其行为方式保持不变。这个原则与 OCP 原则保持一致：它确保继承类不会影响父类的行为，换句话来说，继承类必须可替代它们的基类。

4. 接口分离原则(ISP)

ISP 原则关注的是将契约的方法划分成若干职责分组，并且为这些分组指派不同的接口，这样客户端就不需要实现一个庞大的接口和一堆它们并不使用的方法。这个原则背后的目的是：使用相同接口的类只需要实现特定的一组方法，而不是实现一个庞大的单体方法接口。

5. 依赖倒置原则(DIP)

DIP 原则的宗旨是将自己编写的类与具体的实现隔离开来，让这些类依赖于抽象类或接口。它提倡面向接口(而不是实现)编程，这确保代码不会与某种实现紧密耦合，从而提高了系统的灵活性。

6. 依赖注入(DI)和控制反转(IoC)原则

与 DIP 紧密相关的是 DI 原则和 IoC 原则。DI 通过构造器、方法或属性来提供底层类或从属类。配合使用 DI 原则，这些从属类可以被反转为接口或抽象类，这样就可以形成一个具有较高的可测试性和易于修改的松散耦合系统。

在 IoC 原则中，系统的控制流与过程式编程方法相比是反转的。这个原则的一个示例是 IoC 容器，它的作用是将服务注入到客户端代码，而不必让客户端代码指定具体的实现。在该实例中，控制反转指的是客户端获取服务的行为。

本书中，将更详细地研究每个 S.O.L.I.D.原则。但接下来，将探讨一些专门用来处理特殊情况的企业级模式，它们以常见设计原则和设计模式为基础构建。

1.3　Fowler 的企业设计模式

Martin Fowler 的著作 *Patterns of Enterprise Application Architecture* 是有关如何构建企业级应用程序的最佳实践和模式的参考书。与 GoF 设计模式著作一样，经验丰富的开发者毫无疑问已经遵循该书中归纳的许多设计模式。但 Fowler 著作的价值在于，在对这些模式进行分类时使用一种公用语言来描述模式。该书分为两部分：前半部分讨论 n 层应用程序和数据访问、中间件以及表示层的组织；后半部分是与 GoF 模式著作相似的模式参考，但本书的实现更加具体。

本书将讨论 Fowler 设计模式的 ASP.NET 实现。下面将介绍本书剩余部分将要涉及的模式。

1.3.1　分层

第 3 章讲解了在企业 ASP.NET 应用程序分层时可供自由采用的选项。我们将了解传统的 Web 表单代码隐藏模型带来的问题以及如何使用传统的分层方法将表示、业务逻辑及数据访问等关注点分离开来。

1.3.2　领域逻辑模式

第 4 章介绍了组织业务逻辑的 3 种常见方法：Transaction Script(事务脚本)、Active Record(活动记录)及 Domain Model(领域模型)。

1. Transaction Script

Transaction Script 模式按照线性的、过程式方法来组织业务逻辑。它将细粒度的业务用例映射为细粒度的方法。

2. Active Record

Active Record 模式按照一种能够紧密匹配底层数据结构的方式来组织业务逻辑，即表中表示数据行的对象。

3. Domain Model

Domain Model 模式是对现实领域对象的抽象。同时对数据和行为建模。对象之间可以存在与真实领域相匹配的复杂关系。

我们将展示如何在 ASP.NET 中使用这些模式，以及何时适合选择某一种模式而非其他模式。

1.3.3　对象关系映射

在第 7 章中，将注意力转向如何将业务实体的状态持久化，以及如何从数据存储中检索它们。介绍以下几种支持持久化的基础设施代码所需的企业模式。

1. Unit of Work

Unit of Work(工作单元)模式用来维护一个由已经经过业务事务修改(添加、删除或更新)的业务对象构成的列表。然后，Unit of Work 模式负责将这些发生变化的对象的持久化工作协调成为一个原子动作。如果出现问题，整个事务就会回滚。

2. Repository

Repository(资源库)模式大体上用于对象的逻辑集合，它们更为人知的名字叫做聚合(aggregate)。它充当业务实体的内存集合或仓库，完全将底层数据基础设施抽象出来。

3. Data Mapper

Data Mapper(数据映射器)模式用来从原始数据中提取信息以生成对象，以及将业务对象中的信息转换到数据库。业务对象和数据库彼此互不了解。

4. Identity Map

Identity Map(标识映射)模式监视每一个从数据库中加载的对象，确保所有对象只加载一次。当后面请求该对象时，在从数据库检索之前先检查标志映射。

5. Lazy Loading

Lazy Loading(惰性加载或延迟加载)模式将获取资源的过程推迟到真正需要用到该资源的时候。如果想象一个携带着通讯录的 Customer 对象，那么可以从数据库中提取这个顾客对象，但保留通讯录的生成操作，直到真正需要用到该通讯录时才生成。这可以实现按需加载通讯录，从而避免在从

来不需要用到该地址数据时访问数据库。

6. Query Object

Query Object(查询对象)模式是 GoF 的 Interpreter(解释器)设计模式的一种实现。查询对象充当一种从底层数据库中抽象出来的面向对象查询，它引用的是属性和类，而不是真正的表和列。通常，还需要使用一个翻译器对象来生成用于查询数据库的原生 SQL 语句。

1.3.4　Web 表示模式

在第 8 章中，将注意力转向企业级 ASP.NET 应用程序的表示需求。这一章关注的是专门用来让业务逻辑与表示逻辑分离的模式。首先，将介绍早期 Web 表单开发中普遍使用的代码隐藏模型带来的问题；然后研究那些能够将领域和表示逻辑分离同时让表示层能够有效测试的模式。

这些模式的任务都是将用于表示的逻辑关注点与业务逻辑关注点分离。ASP.NET 表示需要所涵盖的模式有：

- Model-View-Presenter(模型-视图-表示器)。
- Model-View-Controller(模型-视图-控制器)。
- Front Controller(前端控制器)。
- Page Controller(页面控制器)。

1.3.5　基本模式、行为模式和结构模式

在本书中，将介绍如何在企业 ASP.NET 应用程序中利用 Fowler 著作中的其他企业模式。这些模式将包括 Null Object(空对象)、Separated Interface(独立接口)、Registry(注册表)和 Gateway(网关)。

1. Null Object 模式

Null Object(空对象)模式也称为 Special Case(特殊情况)模式，它充当返回值而不是向调用代码返回 null。空对象将与预期结果共享相同的接口或者从相同的基类继承而来，这样减少了在代码基中到处检查 null 情况的需要。

2. Separated Interface 模式

Separated Interface(独立接口)模式要求将接口放在一个独立于具体实现的程序集或命名空间中。这确保客户端完全不知道具体实现，而且能够遵循面向抽象编程(而不是面向实现)以及依赖倒置原则。

3. Gateway 模式

Gateway(网关)模式允许客户端通过一个简化的接口来访问复杂的资源。网关对象基本上将资源 API 包装成一个能够在应用程序中到处使用的单个方法调用。此外，它还隐藏了所有的 API 复杂性。

这里介绍的所有企业模式都将在本书中更详细地进行讨论，并有配套练习来演示如何在 ASP.NET 方案中实现它们。1.4 节是本章最后一部分，简要介绍一些设计方法学，以及运用本章中已经介绍的模式和原则的实践。

1.4　其他有名的设计实践

除了目前已经介绍的设计模式、原则和企业模式之外，我还想介绍几种设计方法学：测试驱动开发、行为驱动开发及领域驱动开发。本节不会深入讨论这些主题，因为已超出了本书的范畴。但是，每一章用来演示模式和原则的关键示例代码均是采用这些方法学设计的，可以从 www.wrox.com 网站和 http://www.tupwk.com.cn 上下载这些代码。

1.4.1　测试驱动设计

TDD(Test-driven Development，测试驱动设计)并不像它的名称所言，它更多的是一种设计方法学而不是测试策略，这个名称只是不够公正。这种设计方法学背后的主要思想是使用测试来塑造系统的设计。在创建软件解决方案时，首先编写一个导致测试失败的测试程序来断言某种业务逻辑。然后编写代码让测试通过。最终，通过重构来清理所有代码。这三步已经被人们称为红-绿-重构(red-green-refactor)。红和绿指的是测试框架分别用来显示测试通过和测试失败的颜色。

通过经历 TDD 流程，最终将得到一个带有一套可以确认所有行为的测试的松耦合系统。TDD 的一个副产品是这些测试提供了一种描述系统能够做什么以及不能做什么的文档。因为测试属于系统的一部分，所以它绝不会过时，这与编写的文档和代码注释不同。

更多有关 TDD 的信息请参考以下著作：

- *Test Driven Development: By Example*，Kent Beck 著
- *The Art of Unit Testing: With Examples in .NET*，Roy Osherove 著
- *Professional Enterprise .NET*，Jon Arking 与 Scott Millett 合著(Wrox 出版)

1.4.2　领域驱动设计

简而言之，DDD(Domain-driven Design，领域驱动设计)是一组有助于构建反映对业务的理解并满足业务需求的应用程序的模式和原则。除此之外，它是一种思考开发方法学的全新方式。DDD 的建模方式如下：首先通过全面理解真实领域来对真实领域建模，然后将所有的术语、规则和逻辑放到代码的某种抽象表示中(通常是以领域模型的形式)。虽然 DDD 并不是一种框架，但是它确实有一组构建块或概念可以整合到解决方案中。

在第 10 章和第 11 章中构建案例研究应用程序时将运用这种方法学。在第 4 章中我们将更深入研究 DDD 的一些方面。

有关 DDD 的更多信息，请参考以下著作：

- *Domain-Driven Design: Tackling Complexity in the Heart of Software*，Eric Evans 著
- *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET*，Jimmy Nilsson 著
- *.NET Domain-Driven Design with C#: Problem - Design - Solution*，Tim McCarthy 著

1.4.3　行为驱动设计

可以将 BDD(Behavior-driven Design，行为驱动设计)被视为 TDD 与 DDD 合并的结果。BDD 关注系统的行为而不仅仅是测试它。使用 BDD 时所创建的规范可以使用在真实领域中随处可见的语言，这能够让技术用户和业务用户同时受益。

采用 BDD 编写规范时产生的文档可以让读者了解系统在各种情况下表现什么样的行为，而不是简单地验证各个方法正在执行它们应该完成的工作。通过将 DDD 的若干方面与核心 TDD 概念有机融合，BDD 将同时满足业务用户和技术用户的需求。可以使用标准的单元测试框架来执行 BDD，但专门的 BDD 框架已经出现了，而且 BDD 即将成为下一个大事件。

如果从网站上下载第 10 章和第 11 章将要构建的案例研究的代码，则能找到编写用来演示系统行为的 BDD 规范。但在本书写作时还没有涉及 BDD 的书籍。因此，建议在 Internet 上搜索有关这项伟大技术的尽可能多的信息。

1.5　小结

本章介绍了一系列可在 ASP.NET 应用程序中采用的设计模式、原则和企业模式。

- GoF 模式是名著 *Design Patterns Bible* 中归类的 23 种模式。这些设计模式是那些反复出现的常见问题的解决方案模板。在团队讨论复杂问题时，可以使用这些模式作为共享的词汇表。
- Robert Martin 的 S.O.L.I.D.设计原则形成了许多设计模式遵循的基础。这些原则旨在提倡松散耦合、可维护性高的、适应变化的面向对象系统。
- Fowler 的企业模式用于企业级应用程序。它们包括用来组织业务逻辑的模式、用来组织表示逻辑的模式、用来组织数据访问的模式以及一系列可在整个系统中使用的基础模式。

对这些模式和原则的介绍层次已经相当高，但在继续阅读本书的过程中，您将会发现我们会更深入地讲解本章涉及的所有概念，同时还有来自真实场景的 ASP.NET 实现，很可能与您要解决问题的系统相关并可以加以利用。

第 2 章将更近距离地了解本书中涉及的 GoF 模式，介绍如何使用设计模式模板以及如何阅读模式所必需的实用知识。

第 4 章

业务逻辑层：组织

本章内容：

- 何时以及如何使用 Transaction Script 模式来组织业务逻辑
- 何时以及如何使用 Active Record 模式和 Castle Windsor 项目来组织业务逻辑
- 何时以及如何使用 Domain Model 模式和 NHibernate 来组织业务逻辑
- 阐释使用 Anemic Model 和 Domain Model 模式来组织业务逻辑的差异
- 理解领域驱动设计(domain-driven design，DDD)以及如何运用它让自己专注于业务逻辑而不是基础设施关注点

业务层在任何企业应用程序中都是最重要的层次，因此，重要的是以最合适的、与应用程序的复杂性相称的方式来组织业务逻辑。本章将介绍 Fowler 的著作 *Patterns of Enterprise Application Architecture* 中首先提出的 4 种模式：Transaction Script(事务脚本)、Active Record(活动记录)、Anemic Model(贫血模型)及 Domain Model(领域模型)。根据构建的应用程序类型不同，每种领域逻辑模式都有其优缺点。

在学习了用于组织领域逻辑的体系结构模式知识之后，将学习 DDD，这种设计方法有助于更有效地理解正在建模的业务领域并确保牢记业务需求。

4.1　理解业务组织模式

并非所有应用程序都是一样的，也并非所有应用程序都需要复杂的体系结构来封装系统的业务逻辑。作为开发者，重要的是要理解所有领域逻辑模式的优缺点，这样才能使用最合适的模式。

4.1.1　Transaction Script

在本章要学习的 4 种领域逻辑模式中，Transaction Script 是最易于理解、掌握和运用的模式。Transaction Script 模式遵循的是过程式开发风格而不是面向对象方法。通常，为每个业务事务创建一个单独的方法，并将它们组合起来放入某种静态管理程序或服务类。每个过程都包含了完成业务事

务所需的所有业务逻辑，包括工作流、业务规则和数据库持久化验证检查。图 4-1 所示为 Transaction Script 模式的图形表示。

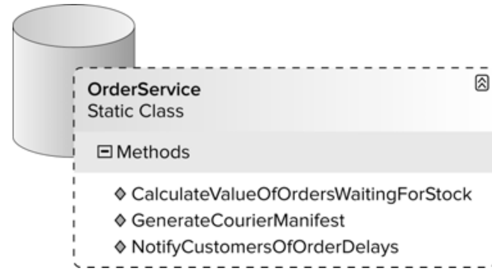

图 4-1

Transaction Script 模式的一个优势是它易于理解，很快就可以让团队新成员上手而不需要具有该模式的预备知识。当出现新需求时，很容易向该类中添加更多方法，而不用担心影响或破坏现有功能。

Transaction Script 模式非常适于那些逻辑中只包含很少或不包含可能增长的功能集合的小型应用程序，以及有不熟悉面向对象编程概念的初级开发者的团队。

当应用程序变大而且业务逻辑变得更复杂时，Transaction Script 模式的问题就会暴露出来。扩展应用程序时，方法的数目也会变多，从而形成一个充斥着功能交叠的细粒度方法的无用 API。尽管可以使用子方法来避免代码重复，如验证和业务规则，但在工作流中的复制不可避免，而且当应用程序规模变大时，代码基很快会变得笨重且不可维护。

因为 Transaction Script 模式非常简单，所以无需一个练习来完整地演示它。相反，考虑下面的代码段，它取自于一个人力资源休假登记应用程序，有助于了解一下该模式的实际用法：

```
public class HolidayService
   {

      public static bool BookHolidayFor(int employeeId, DateTime From, DateTime To)
      {
         bool booked = false;
         TimeSpan numberOfDaysRequestedForHoliday = To - From;

         if (numberOfDaysRequestedForHoliday.Days > 0)
         {
            if (RequestHolidayDoesNotClashWithExistingHoliday(employeeId, From, To))
            {
               int holidayAvailable = GetHolidayRemainingFor(employeeId);

               if (holidayAvailable >=
                          numberOfDaysRequestedForHoliday.Days)
               {
                  SubmitHolidayBookingFor(employeeId, From, To);
                  booked = true;
               }
            }
         }

         return booked;
      }

      private static int GetHolidayRemainingFor(int employeeId)
      {
```

```
            // ...
        }

        public static List<EmployeeDTO> GetAllEmployeesOnLeaveBetween(
                                        DateTime From, DateTime To)
        {
            // ...
        }

        public static List<EmployeeDTO> GetAllEmployeesWithHolidayRemaining()
        {
            // ...
        }

    }
```

代码片段位于 ASPPatterns.Chap4.TransactionScript 文件中

可以看出，整个业务实例被封装到一个单独的方法中。BookHolidayFor 方法处理许多责任，如数据检索和持久化，以及用来确定是否可以休假的业务逻辑。这种过程式编程风格违背了面向对象编程的基本理念。如果逻辑始终非常简单、应用程序较小且易于管理，那么该方式没有问题。

如果应用程序较小，而且业务逻辑简单，不需要采用完全的面向对象方法，Transaction Script 模式可能比较合适。但是，如果应用程序规模会变大，那就可能需要重新考虑业务逻辑结构并寻求更具伸缩性的模式，如 Active Record 模式，这正是 4.1.2 小节的主题。

4.1.2　Active Record

Active Record 模式是一种流行的模式，尤其在底层数据库模型匹配业务模型时它特别有效。通常，数据库中的每张表都对应一个业务对象。业务对象表示表中的一行，并且包含数据、行为以及持久化该对象的工具，此外还有添加新实例和查找对象集合所需的方法。图 4-2 展示一个博客应用程序中的 Post 和 Comment 对象如何与它们对应的数据库表关联起来。该图还说明 Post 中含有一个 Comment 对象集合。

在 Active Record 模式中，每个业务对象均负责自己的持久化和相关的业务逻辑。

Active Record 模式非常适用于在数据模型和业务模型之间具有一对一映射关系的简单应用程序，如博客或论坛引擎。如果已经有数据库模型或者希望采用“数据优先”的方法来构建应用程序，这也是一个可用的好模式。因为业务对象与数据库中的表具有一对一映射关系，而且均具有相同的创建、读取、更新和删除(CRUD)方法，所以可以使用代码生成工具自动生成业务模型。优秀的代码生成工具还会内置所有的数据库验证逻辑，以确保只有有效的数据才会持久化。在第 7 章中讨论如何持久化业务对象时将研究业务对象自动化生成以及使用 Active Record 模式的框架。与 Transaction Script 模式一样，Active Record 模式也非常简单而且易于掌握。

Active Record 模式随着基于数据库的 Web 应用程序而流行，其中一个典型就是结合了 MVC 模式(第 8 章)和 Active Record ORM(第 7 章)的 Ruby on Rails 框架。在.NET 领域，构建在 NHibernate(第 7 章)之上的 Castle ActiveRecord 项目是最流行的开放源代码 Active Record 框架之一，本书将使用该项目以及 ASP.NET MVC 应用程序来构建一个简单的博客网站。因为博客网站只包含少量的业务

逻辑，因此在业务对象和数据模型之间存在较好的相关性，Active Record 模式此时就是一个很好的选择。

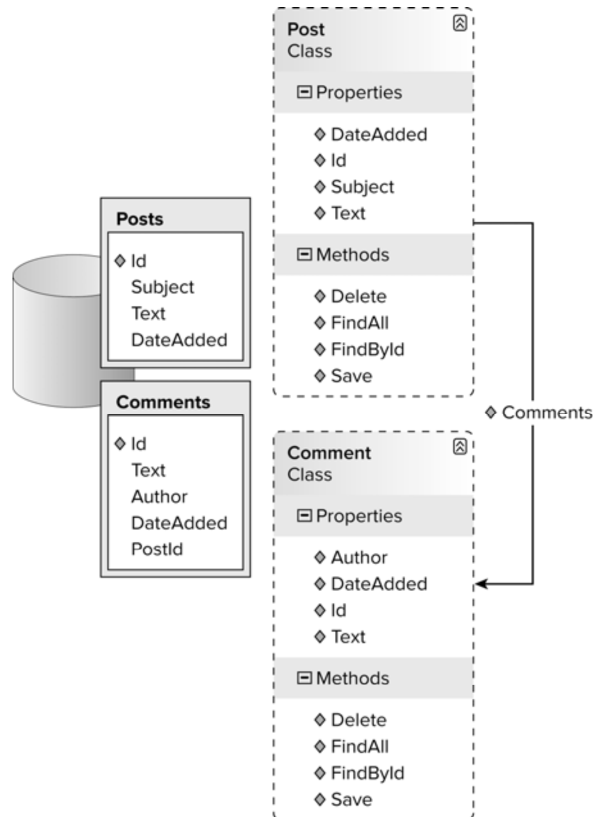

图 4-2

导航到 www.castleproject.org/castle/download.html 并下载 ActiveRecord 项目的最新版本，在本书写作期间，它是 2010 年 1 月 15 日发布的 ActiveRecord 2.1.1。这个下载文件只是一个包含了使用 ActiveRecord 框架所需的所有程序集的 zip 文件。下载这个 zip 文件之后，将所有文件提取到桌面的某个文件夹中。

现在需要为项目创建一个新的解决方案。创建一个名为 ASPPatterns.Chap4.ActiveRecord 的解决方案。向该解决方案中添加一个名为 ASPPatterns.Chap4.ActiveRecord.Model 的新的 C#类库和一个名为 ASPPatterns.Chap4.ActiveRecord.UI.MVC 的新的 MVC Web 应用程序。

ASP.NET MVC Framework 2.0 版已经随 Visual Studio 2010 预安装。但对于 Visual Studio 2008 用户来说需要导航到 www.asp.net/mvc/来安装该框架。

在该解决方案上右击，并在 Windows Explorer 中选择 Open Folder，在这个文件夹中创建一个名为 Lib 的新文件夹，并将 Castle ActiveRecord 下载文件中的所有文件移到该文件夹中。然后回到该解决方案并在 ASPPatterns.Chap4.ActiveRecord.Model 项目上右击，单击 Add Reference，选择 Browse

选项卡，导航到解决方案根目录中的新的 Lib 文件夹并添加所有程序集。在 ASPPatterns.Chap4.ActiveRecord.UI.MVC 项目上右击，并添加下列程序集的引用：

- Castle.ActiveRecord.dll
- NHibernate.dll

最后，还是在 ASPPatterns.Chap4.ActiveRecord.UI.MVC 项目中，添加对 ASPPatterns.Chap4.ActiveRecord.Model 项目的项目引用。现在解决方案已经搭建完毕，可以创建数据库来存放博客帖子。

在 ASPPatterns.Chap4.ActiveRecord.UI.MVC 项目上右击，选择 Add Item 命令。然后选择一个名为 Blog.mdf 的新数据库。一旦创建该数据库，就在它上面双击，从而打开 Server Explorer，并创建两张表，定义如表 4-1 和表 4-2 所示。

表 4-1　Posts 表

列　名	数 据 类 型	是否允许空
Id	Int IDENTITY, Primary Key	False
Subject	nvarchar(200)	False
Text	nvarchar(MAX)	False
DateAdded	Datetime	False

表 4-2　Comments 表

列　名	数 据 类 型	是否允许空
Id	Int IDENTITY, Primary Key	False
Text	nvarchar(MAX)	False
Author	nvarchar(50)	False
DateAdded	Datetime	False
PostId	Int	False

创建一个新的数据库图，将两张表添加进去，并在二者之间建立关系：选择 Posts 表的 Id 列并拖放到 Comments 表的 PostId 列。在修改之后，保存该图，并确认数据库表已更新。

最后，可以开始创建用来表示 Blog Posts 和 Post Comments entities 的模型。向 ASPPatterns.Chap4.ActiveRecord.Model 项目添加一个名为 Comment 的新 C#类，代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Castle.ActiveRecord;

namespace ASPPatterns.Chap4.ActiveRecord.Model
{
    [ActiveRecord("Comments")]
    public class Comment : ActiveRecordBase<Comment>
```

```
    {

        [PrimaryKey]
        public int Id { get; set; }

        [BelongsTo("PostID")]
        public Post Post { get; set; }

        [Property]
        public string Text { get; set; }

        [Property]
        public string Author { get; set; }

        [Property]
        public DateTime DateAdded { get; set; }
    }
}
```

用来修饰 Comment 类属性的特性(attribute)会提示框架这些属性与数据库表的列相匹配。然后，Castle ActiveRecord 框架使用该信息来自动完成业务实体的持久化和检索，而不必编写冗长的 SQL 语句。

向该项目中添加第二个类 Post，其定义如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Castle.ActiveRecord;
using Castle.ActiveRecord.Queries;

namespace ASPPatterns.Chap4.ActiveRecord.Model
{
    [ActiveRecord("Posts")]
    public class Post : ActiveRecordBase<Post>
    {
        [PrimaryKey]
        public int Id { get; set; }

        [Property]
        public string Subject { get; set; }

        [Property]
        public string Text { get; set; }

        public string ShortText
        {
            get {
                if (Text.Length > 20)
```

```
                return Text.Substring(0, 20) + "...";
            else
                return Text;
        }
    }

    [HasMany]
    public IList<Comment> Comments { get; set; }

    [Property]
    public DateTime DateAdded { get; set; }

    public static Post FindLatestPost()
    {
        SimpleQuery<Post> q = new SimpleQuery<Post>
                (@"from Post p order by p.DateAdded desc");

        return (Post)q.Execute()[0];
    }
  }
}
```

这就是模型和数据访问所需的全部代码。是不是非常简单？这正是 Ruby on Rails 开发者们一直在炫耀的技术。

现在可以构造网站来显示帖子和评论，但首先需要将 Visual Studio 在创建项目时添加的所有文件移除。返回到 MVC 项目，将 Visual Studio 自动生成的全部文件从下面的文件夹中移除：

- Content
- Controllers
- Views

向 Controllers 文件夹中添加一个新的控制器 BlogController，其代码定义如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Mvc.Ajax;
using ASPPatterns.Chap4.ActiveRecord.Model;

namespace ASPPatterns.Chap4.ActiveRecord.UI.MVC.Controllers
{
    public class BlogController : Controller
    {
        // GET: /Blog/
        public ActionResult Index()
        {
            Post[] posts = Post.FindAll();
```

```
    if (posts.Count() > 0)
    {
        ViewData["AllPosts"] = posts;
        ViewData["LatestPost"] = Post.FindLatestPost();
        return View();
    }
    else
        return Create();
}

// POST: /Blog/
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult CreateComment(string id, FormCollection collection)
{
    int postId = 0;
    int.TryParse(id, out postId);
    Post post = Post.Find(postId);

    Comment comment = new Comment();
    comment.Post = post;
    comment.Author = Request.Form["Author"];
    comment.DateAdded = DateTime.Now;
    comment.Text = Request.Form["Comment"];

    comment.Save();

    return Detail(post.Id.ToString());
}

// GET: /Blog/Detail/1
public ActionResult Detail(string id)
{
    ViewData["AllPosts"] = Post.FindAll();

    int postId = 0;
    int.TryParse(id, out postId);

    ViewData["LatestPost"] = Post.Find(postId);

    return View("Index");
}

// GET: /Blog/Create
public ActionResult Create()
{
    return View("AddPost");
}

// POST: /Blog/Create
[AcceptVerbs(HttpVerbs.Post)]
```

```
        public ActionResult Create(FormCollection collection)
        {
            Post post = new Post();
            post.DateAdded = DateTime.Now;
            post.Subject = Request.Form["Subject"];
            post.Text = Request.Form["Content"]; ;
            post.Save();

            return Detail(post.Id.ToString());
        }
    }
}
```

向 Views 文件夹中添加两个新的文件夹：Blog 和 Shared。向 Shared 文件夹中添加一个新的 Master 页面 BlogMaster.Master，其标记如下所示：

```
<%@ Master Language="C#" Inherits="System.Web.Mvc.ViewMasterPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
            "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <link href="../../Content/Site.css" rel="stylesheet" type="text/css" />
    <title><asp:ContentPlaceHolder ID="TitleContent" runat="server" /></title>
</head>
<body>
     <div id="document">
        <div id="header"><h1>My Blog</h1></div>
        <div id="nav"><%= Html.ActionLink("Create Post", "Create") %></div>
        <asp:ContentPlaceHolder ID="MainContent" runat="server">
        </asp:ContentPlaceHolder>
    </div>
</body>
</html>
```

向 Blog 视图文件夹中添加一个新的视图 Index，其标记如下所示：

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/BlogMaster.Master"
        Inherits="System.Web.Mvc.ViewPage" %>
<%@ Import Namespace="ASPPatterns.Chap4.ActiveRecord.Model" %>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<div id="content"><h2><%= Html.Encode(((Post)ViewData["LatestPost"]).Subject) %></h2>
        <%= ((Post)ViewData["LatestPost"]).Text.Replace("\n", "<br/>") %>
        <br /><i>posted on
            <%= Html.Encode(((Post)ViewData["LatestPost"])
                                .DateAdded.ToLongDateString()) %></i>
```

```
        <hr />
        Comments<br />
         <% foreach (var item in ((Post)ViewData["LatestPost"]).Comments)
           { %>
               <p><i><%= Html.Encode(item.Author) %>
               said on <%= Html.Encode(item.DateAdded.ToLongDateString()) %>
               at <%= Html.Encode(item.DateAdded.ToShortTimeString()) %>...</i><br />
               <%= Html.Encode(item.Text) %>
               </p>
        <% } %>

        <p>Add a comment</p>
        <% using (Html.BeginForm("CreateComment", "Blog", new {
                   Id = ((Post)ViewData["LatestPost"]).Id }, FormMethod.Post))
        {%>
        <p>
        Your name<br />
        <%= Html.TextBox("Author")%> </p>

        <p>
        Your comment<br />
        <%= Html.TextArea("Comment")%></p>

        <p>
        <input type="submit" value="Add Comment" />

        </p>
        <%} %>
        </div>
        <div id="rightNav"><h2>All Posts</h2>
        <ul>
        <% foreach (var item in (Post[])ViewData["AllPosts"])
           { %>
               <li>
               <%= Html.ActionLink(item.Subject, "Detail",
                                      new { Id=item.Id  })%>
               <br />
               <%= Html.Encode(item.ShortText) %>
               </li>
        <% } %>
        </ul>
        </div>
</asp:Content>
```

向 Blog 视图文件夹中再添加一个新的视图 AddPost，其标记如下所示：

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/BlogMaster.Master"
        Inherits="System.Web.Mvc.ViewPage" %>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
```

```
<% using (Html.BeginForm())
  {%>
  <p>
  Subject<br />
  <%= Html.TextBox("Subject")%> </p>

  <p>
  Content<br />
  <%= Html.TextArea("Content")%></p>

  <p>
  <input type="submit" value="Create" />
  </p>
<%} %>
</asp:Content>
```

打开 Global.asax 文件并进行更新，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using ASPPatterns.Chap4.ActiveRecord.Model;
using Castle.ActiveRecord.Framework;
using System.Configuration;

namespace ASPPatterns.Chap4.ActiveRecord.UI.MVC
{
   public class MvcApplication : System.Web.HttpApplication
   {
      public static void RegisterRoutes(RouteCollection routes)
      {
         routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

         routes.MapRoute(
            "Default",
            "{controller}/{action}/{id}",
            new { controller = "Blog", action = "Index", id = "" }
         );

      }

      protected void Application_Start()
      {
         RegisterRoutes(RouteTable.Routes);

         IConfigurationSource source = ConfigurationManager
                            .GetSection("activeRecord") as IConfigurationSource;
```

```
            Castle.ActiveRecord.ActiveRecordStarter
                                        .Initialize(source, typeof(Post), typeof(Comment));
        }
    }
}
```

Global.asax 文件中的代码只是告诉 Castle ActiveRecord 框架进行初始化，这样就可以开始使用它。

要让 Castle ActiveRecord 运行起来还需要做的最后一件事情就是，修改 web.config 文件，将 Castle ActiveRecord 声明包含进来，配置片段代码如下：

```
<configuration>
    <configSections>
      <section
        name="activeRecord"
         type="Castle.ActiveRecord.Framework.Config.ActiveRecordSectionHandler,
        Castle.ActiveRecord"/>
      …
     </configSections>
     <activeRecord isWeb="true">
       <config>
        <add key="hibernate.connection.driver_class"
            value="NHibernate.Driver.SqlClientDriver"/>
        <add key="dialect" value="NHibernate.Dialect.MsSql2005Dialect"/>
        <add key="hibernate.connection.provider"
            value="NHibernate.Connection.DriverConnectionProvider"/>
        <add key="connection.connection_string"
            value="DataSource=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\Blog.mdf;
                  Integrated Security=True;User Instance=True"/>
       <add key="proxyfactory.factory_class"
           value="NHibernate.ByteCode.Castle.ProxyFactoryFactory,
                 NHibernate.ByteCode.Castle"/>
       </config>
    </activeRecord>
    …
</configuration >
```

为了让博客更加美观，可以在 Content 文件夹中添加一个新的样式表文件 Site.css：

```
#document{
width:750px;
margin:0 auto;
}

#content {
float:left;
width:500px;
}
```

```
#rightNav {
float:right;
width:250px;
}
```

运行该解决方案，将能够使用该博客。图 4-3 给出了该博客应用程序运行时的情况。

图 4-3

这里以极快的速度构建好博客应用程序，这在很大程度上要归功于 Castle ActiveRecord 框架，由于对象模型与数据模型之间紧密相关，它能够将数据的检索和访问操作自动化。

Active Record 模式并非灵丹妙药。它擅长于处理底层数据模型能够很好映射到业务模型的情形，但是当出现不匹配时(有时候称为*阻抗失配*)，该模式将很难应对。这是由于复杂系统的概念业务模型有时与数据模型不同所造成的。如果业务领域非常丰富，有着大量的复杂规则、逻辑和工作流，那么采用 Domain Model 方法将更加有利。这正是 4.1.3 小节中要讨论的模式。

4.1.3　Domain Model

可以将 Domain Model 视为表示正在处理的领域的概念层。事物以及事物之间的关系都存在于这个模型中。这里的“事物”表示什么意思呢？例如，如果正在构建一个电子商务商店，那么这个模型中的“事物”将表示购物车、订单、订单项及类似的事物。这些事物包含数据，更重要的是它们还有行为。一张订单不仅有表示创建日期、状态和订单流水号的属性；还包含应用到凭证的业务逻辑，其中包括围绕它的所有领域规则：凭证是否有效？该凭证能否用于购物车中的商品？是否存在任何其他导致该凭证失效的出价？Domain Model 越能密切地表示真实的领域越好，这是因为更容易理解和复制组织中的复杂的业务逻辑、规则和验证过程。Domain Model 与 Active Record 模式之间的

主要差别在于，Domain Model 中存在的业务实体并不知道如何持久化自己，而且没有必要在数据模型和业务模型之间建立一对一的映射关系。

1. POCO 和 PI

前面曾经提到过，Domain Model 与 Active Record 模式不同，它不知道持久化。术语持久化不知(persistence ignorance，PI)表示普通 CLR 对象(plain old common runtime object，POCO)业务实体的朴实本质。那么如何将 Domain Model 的业务对象持久化呢？通常使用 Repository 模式(第 7 章)。采用 Domain Model 模式时，Repository 对象以及数据映射器(第 7 章)负责将业务实体及相关实体的对象图映射到数据模型。

2. 代码示例

为了演示 Domain Model 模式，将创建一个解决方案来为银行领域建模，这涉及账号的创建以及账号之间的现金转账。

创建一个名为 ASPPatterns.Chap4.DomainModel 的新的解决方案，并向其中添加下面的类库项目：

- ASPPatterns.Chap4.DomainModel.Model
- ASPPatterns.Chap4.DomainModel.AppService
- ASPPatterns.Chap4.DomainModel.Repository

此外还添加一个新的 Web 应用程序 ASPPatterns.Chap4.DomainModel.UI.Web。在 Repository 项目上右击，并添加对 Model 项目的项目引用。在 AppService 项目上右击，添加对 Model 和 Repository 项目的项目引用。最后，在 Web 项目上右击，添加对 AppService 项目的项目引用。

图 4-4 给出了已创建项目的图形表示。在该图之后列出了每个对象的责任。

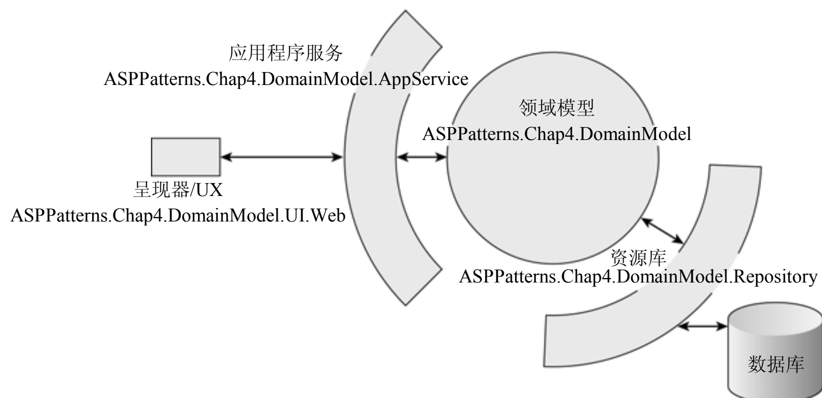

图 4-4

- **ASPPatterns.Chap4.DomainModel.Model** Domain Model 项目将包含应用程序内的所有业务逻辑。领域对象将存放在此处，并与其他对象建立关系，从而表示应用程序正在构建的银行领域。该项目还将以接口的形式为领域对象持久化和检索定义契约，将采用 Repository 模式来实现所有的持久化管理需求。(将在第 7 章中更加详细地讨论 Repository 模式)。Model 项目不会引用其他任何项目，从而确保：让它与任何基础设施关注点保持隔离，并坚定地只关注业务领域。

- ASPPatterns.Chap4.DomainModel.Repository　Repository 项目将包含 Model 项目中定义的资源库接口的具体实现。Repository 引用了 Model 项目，从而从数据库提取并持久化领域对象。Repository 项目只关注领域对象持久化和检索的责任。
- ASPPatterns.Chap4.DomainModel.AppService　AppService 项目将充当应用程序的网关(API，如果愿意的话)。表示层将通过消息(简单的数据传输对象)与 AppService 通信。将在第 7 章中详细讨论消息传送模式。AppService 层还将定义视图模型，这些是领域模型的展开视图，只用于数据显示。第 8 章中将更详细地讨论该主题。
- ASPPatterns.Chap4.DomainModel.UI.Web　UI.Web 项目负责应用程序的表示和用户体验需求。这个项目只与 AppService 交互，并接收专门为用户体验视图创建的强类型视图模型。

在确定解决方案结构之后，就可以搭建数据库来存放领域中银行账号的状态。向 Web 项目中添加一个新项，选择 new database，并将其命名为 BankAccount.mdf。一旦数据库创建完毕，双击它打开 Server Explorer，并创建两张表，定义如表 4-3 和表 4-4 所示。

表 4-3　BankAccounts 表

列　　名	数 据 类 型	是否允许空
BankAccountId	uniqueidentifier, Primary Key	False
Balance	money	False
CustomerRef	nvarchar(50)	False

表 4-4　Transactions 表

列　　名	数 据 类 型	是否允许空
BankAccountId	uniqueidentifier	False
Deposit	money	False
Withdrawal	money	False
Reference	nvarchar(50)	False

创建新的数据库图，将两张表添加进去并在它们之间建立关系：选择 BankAccounts 表的 BankAccountId 列并将其拖放到 Transactions 表的 BankAccountId 列。在修改之后，保存该图并确认已更新这些表。

在搭建好解决方案框架和数据库之后，就可以真正开始领域建模。在这个场景中，BankAccount 为发生的每个动作创建一个 Transaction 对象。图 4-5 给出了这个简单领域模型的类图。

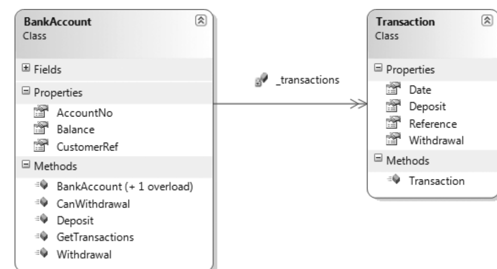

图 4-5

在 Model 项目中创建一个新类 Transaction，其代码定义如下：

```
public class Transaction
{
   public Transaction(decimal deposit, decimal withdrawal,
                string reference, DateTime date)
   {
      this.Deposit = deposit;
      this.Withdrawal = withdrawal;
      this.Reference = reference;
      this.Date = date;
   }

   public decimal Deposit
   { get; internal set; }

   public decimal Withdrawal
   { get; internal set; }

   public string Reference
   { get; internal set; }

   public DateTime Date
   { get; internal set; }
}
```

注意，根据本示例演示的目的，Transaction 对象并没有标识符(identifier)属性，而对应的数据表没有指定主键。Transaction 对象被称为值对象，这是 DDD(领域驱动设计)中使用的一个术语，将在本章末尾讨论。

添加第二个类 BankAccount，并输入如下代码：

```
public class BankAccount
{
   private decimal _balance;
   private Guid _accountNo;
   private string _customerRef;
   private IList<Transaction> _transactions;

   public BankAccount() : this(Guid.NewGuid(), 0,
                       new List<Transaction>(), "")
   {
      _transactions.Add(new Transaction(0m, 0m, "account created", DateTime.Now));
   }

   public BankAccount(Guid Id, decimal balance,
                IList<Transaction> transactions, string customerRef)
   {
      AccountNo = Id;
      _balance = balance;
      _transactions = transactions;
      _customerRef = customerRef;
```

```
        }

        public Guid AccountNo
        {
            get { return _accountNo; }
            internal set { _accountNo = value; }
        }

        public decimal Balance
        {
            get { return _balance; }
            internal set { _balance = value; }
        }

        public string CustomerRef
        {
            get { return _customerRef; }
            set { _customerRef = value; }
        }

        public bool CanWithdraw(decimal amount)
        {
            return (Balance >= amount);
        }

        public void Withdraw(decimal amount, string reference)
        {
            if (CanWithdraw(amount))
            {
                Balance -= amount;
                _transactions.Add(new Transaction(0m, amount,
                                            reference, DateTime.Now ));
            }
        }

        public void Deposit(decimal amount, string reference)
        {
            Balance += amount;
            _transactions.Add(new Transaction(amount, 0m, reference, DateTime.Now));
        }

        public IEnumerable<Transaction> GetTransactions()
        {
            return _transactions;
        }
    }
```

BankAccount 有如下 3 个简单的方法：

- CanWithdraw
- Withdraw

- Deposit

因为有一个 CanWithdraw 方法，所以应该期望调用代码在尝试从某个账号取回现金时使用 Test-Doer(先测试再执行)模式：

```
If (myBankAccount.CanWithdraw(amountToWithdraw))
{
      myBankAccount.Withdraw(amountToWithdraw);
}
```

如果在不进行检查的情况下对一个没有足够现金金额的账号调用 Withdraw 方法，则应该抛出一个异常。为此，需要一个新的自定义异常，因此，向 Model 项目中添加一个新类 InsufficientFundsException，代码如下：

```
public class InsufficientFundsException : ApplicationException
{
}
```

修改 BankAccount 类的 Withdraw 方法，如下所示：

```
public void Withdraw(decimal amount, string reference)
{
   if (CanWithdraw(amount))
   {
      Balance -= amount;
      _transactions.Add(new Transaction(0m, amount, reference, DateTime.Now));
   }
   else
   {
      throw new InsufficientFundsException();
   }
}
```

现在需要某种方法来持久化 BankAccount 和 Transactions，但因为不希望污染 Domain Model 项目，所以将添加 Repository 的接口来定义契约，以满足实体的持久化和检索需求。这印证了本章前面学习术语 PI 和 POCO 时学到的概念。

创建一个新的带有如下契约的新接口 IBankAccountRepository：

```
   public interface IBankAccountRepository
   {
      void Add(BankAccount bankAccount);
      void Save(BankAccount bankAccount);
      IEnumerable<BankAccount> FindAll();
      BankAccount FindBy(Guid AccountId);
}
```

有些动作没有很好地映射到领域实体的方法。对于这类情况，可以使用领域服务。在两个账号之间转账的动作就是一种属于服务类的责任。本章末尾将学习更多有关领域服务的知识。

向 Model 项目中添加一个名为 BankAccountService 的新类，代码定义如下：

```
public class BankAccountService
{
   private IBankAccountRepository _bankAccountRepository;

   public BankAccountService(IBankAccountRepository bankAccountRepository)
   {
      _bankAccountRepository = bankAccountRepository;
   }

   public void Transfer(Guid accountNoTo, Guid accountNoFrom, decimal amount)
   {
      BankAccount bankAccountTo =
                       _bankAccountRepository.FindBy(accountNoTo);
      BankAccount bankAccountFrom = _bankAccountRepository.FindBy(accountNoFrom);

      if (bankAccountFrom.CanWithdraw(amount))
      {
         bankAccountTo.Deposit(amount,
                          "From Acc " + bankAccountFrom.CustomerRef + " ");
         bankAccountFrom.Withdraw(amount,
                     "Transfer To Acc " + bankAccountTo.CustomerRef + " ");

         _bankAccountRepository.Save(bankAccountTo);
         _bankAccountRepository.Save(bankAccountFrom);
      }
      else
      {
         throw new InsufficientFundsException();
      }
   }
}
```

在 BankAccountService 的当前实现中，在保存两个银行账号之间发生的任何错误均会让数据处于非法状态。在第 7 章中，将介绍工作单元模式如何能够确保所需的事务作为一个原子动作进行提交，万一出现异常就回滚。

现在已经构建领域模型，那么可以编写方法来持久化 Bank Account 和 Transaction 业务对象。在 Repository 项目中，添加一个新类 BankAccountRepository。该类将实现接口 IBankAccountRepository。下面的代码有点长。在第 7 章中，将学习一些常见的对象关系映射器，能够节省编写 ADO.NET 基础设施代码的时间。

还需要添加对 System.Configuration 程序集的引用，因为 BankAccountRepository 需要从应用程序的 web.config 文件中获取一个数据库连接字符串。

```
using ASPPatterns.Chap4.DomainModel.Model;
using System.Data.SqlClient;
using System.Data;
```

```
using System.Configuration;

namespace ASPPatterns.Chap4.DomainModel.Repository
{
   public class BankAccountRepository : IBankAccountRepository
   {
      private string _connectionString;

      public BankAccountRepository()
      {
         _connectionString = ConfigurationManager.ConnectionStrings
                        ["BankAccountConnectionString"].ConnectionString;
      }

      public void Add(BankAccount bankAccount)
      {
         string insertSql = "INSERT INTO BankAccounts " +
                        "(BankAccountID, Balance, CustomerRef) VALUES " +
                        "(@BankAccountID, @Balance, @CustomerRef)";

         using (SqlConnection connection =
               new SqlConnection(_connectionString))
         {
            SqlCommand command = connection.CreateCommand();
            command.CommandText = insertSql;

            SetCommandParametersForInsertUpdateTo(bankAccount, command);

            connection.Open();

            command.ExecuteNonQuery();
         }

         UpdateTransactionsFor(bankAccount);
      }

      public void Save(BankAccount bankAccount)
      {
         string bankAccoutnUpdateSql =
                "UPDATE BankAccounts " +
                "SET Balance = @Balance, CustomerRef= @CustomerRef " +
                "WHERE BankAccountID = @BankAccountID;";

         using (SqlConnection connection =
               new SqlConnection(_connectionString))
         {
            SqlCommand command = connection.CreateCommand();
            command.CommandText = bankAccoutnUpdateSql;

            SetCommandParametersForInsertUpdateTo(bankAccount, command);
```

```
        connection.Open();

        command.ExecuteNonQuery();
    }

    UpdateTransactionsFor(bankAccount);
}

private static void SetCommandParametersForInsertUpdateTo(
                    BankAccount bankAccount, SqlCommand command)
{
   command.Parameters.Add(new SqlParameter(
    "@BankAccountID", bankAccount.AccountNo)); command.Parameters.Add(
     new SqlParameter("@Balance", bankAccount.Balance));
   command.Parameters.Add(
        new SqlParameter("@CustomerRef", bankAccount.CustomerRef));
}

private void UpdateTransactionsFor(BankAccount bankAccount)
{
    string deleteTransactionSQl =
       "DELETE Transactions WHERE BankAccountId = @BankAccountId;";

    using (SqlConnection connection =
           new SqlConnection(_connectionString))
    {
        SqlCommand command = connection.CreateCommand();
        command.CommandText = deleteTransactionSQl;
        command.Parameters.Add(
           new SqlParameter("@BankAccountID", bankAccount.AccountNo));
        connection.Open();
        command.ExecuteNonQuery();

    }

    string insertTransactionSql =
        "INSERT INTO Transactions " +
          "(BankAccountID, Deposit, Withdraw, Reference, [Date]) VALUES " +
        "(@BankAccountID, @Deposit,  @Withdraw,  @Reference, @Date)";

    foreach (Transaction tran in bankAccount.GetTransactions())
    {
        using (SqlConnection connection =
               new SqlConnection(_connectionString))
        {
            SqlCommand command = connection.CreateCommand();
            command.CommandText = insertTransactionSql;

            command.Parameters.Add(new SqlParameter(
```

```
                "@BankAccountID",bankAccount.AccountNo));
              command.Parameters.Add(new SqlParameter("@Deposit", tran.Deposit));
            command.Parameters.Add(
                new SqlParameter("@Withdraw", tran.Withdrawal));
            command.Parameters.Add(
                new SqlParameter("@Reference", tran.Reference));
            command.Parameters.Add(new SqlParameter("@Date", tran.Date));

            connection.Open();
            command.ExecuteNonQuery();
        }
    }
}

public IEnumerable<BankAccount> FindAll()
{
    IList<BankAccount> accounts = new List<BankAccount>();

    string queryString =
      "SELECT * FROM dbo.Transactions INNER JOIN " +
      "dbo.BankAccounts ON " +
        "dbo.Transactions.BankAccountId = dbo.BankAccounts.BankAccountId " +
      "ORDER BY dbo.BankAccounts.BankAccountId;";

    using (SqlConnection connection =
          new SqlConnection(_connectionString))
    {
          SqlCommand command = connection.CreateCommand();
          command.CommandText = queryString;

          connection.Open();

          using (SqlDataReader reader = command.ExecuteReader())
          {
              accounts = CreateListOfAccountsFrom(reader);
          }
    }

    return accounts;
}

private IList<BankAccount> CreateListOfAccountsFrom(
                                IDataReader datareader)
{
    IList<BankAccount> accounts = new List<BankAccount>();
    BankAccount bankAccount;
    string id = "";
    IList<Transaction> transactions = new List<Transaction>();

    while (datareader.Read())
```

```
    {
        if (id != datareader["BankAccountId"].ToString())
        {
            id = datareader["BankAccountId"].ToString();
            transactions = new List<Transaction>();
            bankAccount = new BankAccount(
                  new Guid(id), Decimal.Parse(datareader["Balance"].ToString()),
                  transactions, datareader["CustomerRef"].ToString());

            accounts.Add(bankAccount);
        }
        transactions.Add(CreateTransactionFrom(datareader));
    }

    return accounts;
}

private Transaction CreateTransactionFrom(IDataRecord rawData)
{
    return new Transaction(
            Decimal.Parse(rawData["Deposit"].ToString()),
            Decimal.Parse(rawData["Withdraw"].ToString()),
            rawData["Reference"].ToString(),
            DateTime.Parse(rawData["Date"].ToString()));
}


public BankAccount FindBy(Guid accountId)
{
    BankAccount account;

    string queryString = "SELECT * FROM " +
                     "dbo.Transactions INNER JOIN " +
                     "dbo.BankAccounts ON " +
                     "dbo.Transactions.BankAccountId = " +
                     "dbo.BankAccounts.BankAccountId " +
       "WHERE dbo.BankAccounts.BankAccountId = @BankAccountId;";

    using (SqlConnection connection =
          new SqlConnection(_connectionString))
    {
        SqlCommand command = connection.CreateCommand();
        command.CommandText = queryString;

        SqlParameter Idparam = new SqlParameter("@BankAccountId", accountId);
        command.Parameters.Add(Idparam);

        connection.Open();

        using (SqlDataReader reader = command.ExecuteReader())
```

```
                {
                    account = CreateListOfAccountsFrom(reader)[0];
                }
            }
            return account;
        }
    }
}
```

现在已经处理好持久化和检索需求，可以为客户端添加一个服务层，让其与系统以一种简单的方式进行交互。

在 AppServices 项目中添加一个新文件夹 ViewModel，并向该文件夹中添加两个新类 BankAccountView 和 TransactionView，其定义如下：

```
public class TransactionView
{
    public string Deposit { get; set; }
    public string Withdrawal { get; set; }
    public string Reference { get; set; }
    public DateTime Date { get; set; }
}

public class BankAccountView
{
    public Guid AccountNo { get; set; }
    public string Balance { get; set; }
    public string CustomerRef { get; set; }
    public IList<TransactionView> Transactions { get; set; }
}
```

BankAccountView 和 TransactionView 提供了领域模型用于表示的展开视图，第 6 章将更深入研究这个主题。为了将领域实体转换成数据传输视图模型，需要映射器类。第 8 章将更深入研究这个主题并学习一种将这个过程自动化的方法。创建一个带有如下两个静态方法的新类 ViewMapper：

```
using ASPPatterns.Chap4.DomainModel.Model;

namespace ASPPatterns.Chap4.DomainModel.AppService
{
    public static class ViewMapper
    {
        public static TransactionView CreateTransactionViewFrom(Transaction tran)
        {
            return new TransactionView
            {
                Deposit = tran.Deposit.ToString("C"),
                Withdrawal = tran.Withdrawal.ToString("C"),
                Reference = tran.Reference,
                Date = tran.Date
```

```
            };
        }

        public static BankAccountView CreateBankAccountViewFrom(BankAccount acc)
        {
            return new BankAccountView
            {
                AccountNo = acc.AccountNo,
                Balance = acc.Balance.ToString("C"),
                CustomerRef = acc.CustomerRef,
                Transactions = new List<TransactionView>()
            };
        }
    }
}
```

向 AppServices 项目中再添加一个文件夹 Messages，该文件夹将包含所有用来与服务层通信的请求-应答对象。第 6 章将更详细地讲解 Messaging(消息传送)模式。因为所有的应答都共享一组相同的属性，所以可以创建一个基类。向 Messages 文件夹中添加一个新类 ResponseBase，代码如下：

```
namespace ASPPatterns.Chap4.DomainModel.AppService.Messages
{
    public abstract class ResponseBase
    {
        public bool Success { get; set; }
        public string Message { get; set; }
    }
}
```

Success 属性指明被调用的方法是否成功运行，而 Message 属性包含该方法运行结果的详细信息。现在需要实现所有的请求和应答对象。为下面显示的每个类代码清单分别创建一个新类：

```
    public class BankAccountCreateRequest
    {
        public string CustomerName { get; set; }
    }

    public class BankAccountCreateResponse : ResponseBase
    {
        public Guid BankAccountId { get; set; }
    }

    public class DepositRequest
    {
        public Guid AccountId { get; set; }
        public decimal Amount { get; set; }
    }

    public class FindAllBankAccountResponse : ResponseBase
```

```
    {
        public IList<BankAccountView> BankAccountView { get; set; }
    }

    public class FindBankAccountResponse : ResponseBase
    {
        public BankAccountView BankAccount { get; set; }
    }

    public class TransferRequest
    {
        public Guid AccountIdTo { get; set; }
        public Guid AccountIdFrom { get; set; }
        public decimal Amount { get; set; }
    }

    public class TransferResponse : ResponseBase
    {
    }

    public class WithdrawalRequest
    {
        public Guid AccountId { get; set; }
        public decimal Amount { get; set; }
    }
```

在消息传送对象就位之后，就可以添加服务类来协调对领域实体(服务和资源库)的方法调用。在 AppService 项目的根目录下面添加一个新类 ApplicationBankAccountService：

```
using ASPPatterns.Chap4.DomainModel.Model;
using ASPPatterns.Chap4.DomainModel.Repository;
using ASPPatterns.Chap4.DomainModel.AppService.Messages;

namespace ASPPatterns.Chap4.DomainModel.AppService
{
    public class ApplicationBankAccountService
    {
        private BankAccountService _bankAccountService;
        private IBankAccountRepository _bankRepository;

        public ApplicationBankAccountService() :
            this (new BankAccountRepository(),
                  new BankAccountService(new BankAccountRepository()))
        { }

        public ApplicationBankAccountService(
                        IBankAccountRepository bankRepository,
                        BankAccountService bankAccountService)
        {
            _bankRepository = bankRepository;
```

```
        _bankAccountService = bankAccountService;
    }

    public ApplicationBankAccountService(
                  BankAccountService bankAccountService,
                  IBankAccountRepository bankRepository)
    {
        _bankAccountService = bankAccountService;
        _bankRepository = bankRepository;
    }

    public BankAccountCreateResponse CreateBankAccount(
              BankAccountCreateRequest bankAccountCreateRequest)
    {
        BankAccountCreateResponse bankAccountCreateResponse =
                                  new BankAccountCreateResponse();
        BankAccount bankAccount = new BankAccount();

        bankAccount.CustomerRef = bankAccountCreateRequest.CustomerName;
        _bankRepository.Add(bankAccount);

        return bankAccountCreateResponse;
    }

    public void Deposit(DepositRequest depositRequest)
    {
        BankAccount bankAccount = _bankRepository.FindBy(depositRequest.AccountId);

        bankAccount.Deposit(depositRequest.Amount, "");

        _bankRepository.Save(bankAccount);
    }

    public void Withdrawal(WithdrawalRequest withdrawalRequest)
    {
        BankAccount bankAccount =
            _bankRepository.FindBy(withdrawalRequest.AccountId);

        bankAccount.Withdraw(withdrawalRequest.Amount, "");

        _bankRepository.Save(bankAccount);
    }

    public TransferResponse Transfer(TransferRequest request)
    {
        TransferResponse response = new TransferResponse();

        try
        {
            _bankAccountService.Transfer(request.AccountIdTo,
```

```
                                request.AccountIdFrom, request.Amount);
                response.Success = true;
            }
            catch (InsufficientFundsException)
            {
                response.Message = "There is not enough funds in account no: " +
                                        request.AccountIdFrom.ToString();
                response.Success = false;
            }

            return response;
        }

        public FindAllBankAccountResponse GetAllBankAccounts()
        {
            FindAllBankAccountResponse FindAllBankAccountResponse =
                                        new FindAllBankAccountResponse();
            IList<BankAccountView> bankAccountViews =
                                new List<BankAccountView>();
            FindAllBankAccountResponse.BankAccountView = bankAccountViews;

            foreach (BankAccount acc in _bankRepository.FindAll())
            {
                bankAccountViews.Add(
                        ViewMapper.CreateBankAccountViewFrom(acc));
            }

            return FindAllBankAccountResponse;
        }

        public FindBankAccountResponse GetBankAccountBy(Guid Id)
        {
            FindBankAccountResponse bankAccountResponse = new FindBankAccountResponse();
           BankAccount acc = _bankRepository.FindBy(Id);
            BankAccountView bankAccountView = ViewMapper.CreateBankAccountViewFrom(acc);

           foreach (Transaction tran in acc.GetTransactions())
           {
             bankAccountView.Transactions.Add(
                  ViewMapper.CreateTransactionViewFrom(tran));
           }

           bankAccountResponse.BankAccount = bankAccountView;

           return bankAccountResponse;
        }

    }
}
```

BankAccountApplicationService 类协调应用程序活动并将所有的业务任务委托给领域模型。该层并不包含任何业务逻辑，有助于防止任何与业务无关的代码污染领域模型项目。该层还将领域实体转换成数据传输对象，从而保护领域的内部操作，并为一起工作的表示层提供了一个易于使用的 API。

为了简单起见，这里选择了“穷人的依赖注入”方式并对默认的构造器硬编码，以便使用已经编码的资源库领域服务实现。在第 8 章中，将学习 IoC 和 IoC 容器来提供类的依赖关系。

最后的动作是创建用户界面，从而能够创建账号并执行交易。在 Web 项目中以源代码方式打开 Default.aspx 并编辑标记，让其匹配如下的片段：

```
 …
<form id="form1" runat="server">
<div>

 <fieldset>
    <legend>Create New Account</legend>
 <p>
    Customer Ref:
    <asp:TextBox ID="txtCustomerRef" runat="server" />

    <asp:Button ID="btCreateAccount" runat="server" Text="Create Account"
       onclick="btCreateAccount_Click" />
</p>
</fieldset>

<fieldset>
    <legend>Account Detail</legend>
    <p>
    <asp:DropDownList AutoPostBack="true"
         ID="ddlBankAccounts" runat="server"
         onselectedindexchanged="ddlBankAccounts_SelectedIndexChanged"/ >
    </p>
    <p>
       Account No:
       <asp:Label ID="lblAccountNo" runat="server" />
    </p>
    <p>
       Customer Ref:
       <asp:Label ID="lblCustomerRef" runat="server" />
    </p>
    <p>
       Balance:
       <asp:Label ID="lblBalance" runat="server" />
    </p>
    <p>
       Amount ¡ê<asp:TextBox ID="txtAmount" runat="server" Width="60px"/>
        
       <asp:Button ID="btnWithdrawal" runat="server" Text="Withdrawal"
          onclick="btnWithdrawal_Click" />
```

```
             
            <asp:Button ID="btnDeposit" runat="server" Text="Deposit"
               onclick="btnDeposit_Click" />
        </p>
        <p>
            Transfer
            ¡ê<asp:TextBox ID="txtAmountToTransfer" runat="server"
                        Width="60px" />

         to
        <asp:DropDownList AutoPostBack="true"
             ID="ddlBankAccountsToTransferTo" runat="server"/>
         
         <asp:Button ID="btnTransfer" runat="server" Text="Commit"
               onclick="btnTransfer_Click" />
        </p>
        <p>
            Transactions</p>
            <asp:Repeater ID="rptTransactions" runat="server">
                <HeaderTemplate>
                    <table>
                    <tr>
                        <td>deposit</td>
                        <td>withdrawal</td>
                        <td>reference</td>
                    </tr>
                </HeaderTemplate>
                <ItemTemplate>
                    <tr>
                        <td><%# Eval("Deposit") %></td>
                        <td><%# Eval("Withdrawal") %></td>
                        <td><%# Eval("Reference") %></td>
                        <td><%# Eval("Date") %></td>
                    </tr>
                </ItemTemplate>
                <FooterTemplate>
                    </table>
                </FooterTemplate>
            </asp:Repeater>
    </fieldset>
    </div>
    </form>
</body>
</html>
```

切换到 Default.aspx 页面的隐藏代码并进行更新，以匹配如下的代码清单：

```
using System;
using System.Web.UI.WebControls;
using ASPPatterns.Chap4.DomainModel.AppService;
```

```
using ASPPatterns.Chap4.DomainModel.AppService.Messages;

namespace ASPPatterns.Chap4.DomainModel.UI.Web
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            if (!Page.IsPostBack)
                ShowAllAccounts();
        }

        private void ShowAllAccounts()
        {
            ddlBankAccounts.Items.Clear();

            FindAllBankAccountResponse response =
                    new ApplicationBankAccountService().GetAllBankAccounts();
            ddlBankAccounts.Items.Add(new ListItem("Select An Account", ""));

            foreach (BankAccountView accView in response.BankAccountView)
            {
                ddlBankAccounts.Items.Add(
                 new ListItem(accView.CustomerRef, accView.AccountNo.ToString()));
            }
        }

        protected void btCreateAccount_Click(object sender, EventArgs e)
        {
            BankAccountCreateRequest createAccountRequest =
                                       new BankAccountCreateRequest();
            createAccountRequest.CustomerName = this.txtCustomerRef.Text;
            ApplicationBankAccountService service = new ApplicationBankAccountService();

            service.CreateBankAccount(createAccountRequest);

            ShowAllAccounts();
        }

        protected void ddlBankAccounts_SelectedIndexChanged(object sender, EventArgs e)
        {
            DisplaySelectedAccount();
        }

        private void DisplaySelectedAccount()
        {
            if (ddlBankAccounts.SelectedValue.ToString() != "")
            {
                ApplicationBankAccountService service =
                                        new ApplicationBankAccountService();
```

```
            FindBankAccountResponse response =
                    service.GetBankAccountBy(
                       new Guid(ddlBankAccounts.SelectedValue.ToString()));
            BankAccountView accView = response.BankAccount;

            this.lblAccountNo.Text = accView.Balance.ToString();
            this.lblBalance.Text = accView.Balance.ToString();
            this.lblCustomerRef.Text = accView.CustomerRef;

            rptTransactions.DataSource = accView.Transactions;
            rptTransactions.DataBind();

            FindAllBankAccountResponse allAccountsResponse =
                                          service.GetAllBankAccounts();

            ddlBankAccountsToTransferTo.Items.Clear();

            foreach (BankAccountView acc in allAccountsResponse.BankAccountView)
            {
              if (acc.AccountNo.ToString() != ddlBankAccounts.SelectedValue.ToString())
                  ddlBankAccountsToTransferTo.Items.Add(
                   new ListItem(acc.CustomerRef, acc.AccountNo.ToString()));
            }
        }
    }

    protected void btnWithdrawal_Click(object sender, EventArgs e)
    {
        ApplicationBankAccountService service = new ApplicationBankAccountService();
        WithdrawalRequest request = new WithdrawalRequest();
        Guid AccId = new Guid(ddlBankAccounts.SelectedValue.ToString());
        request.AccountId = AccId;
        request.Amount = Decimal.Parse(txtAmount.Text);

        service.Withdrawal(request);
        DisplaySelectedAccount();
    }

    protected void btnDeposit_Click(object sender, EventArgs e)
    {
        ApplicationBankAccountService service = new ApplicationBankAccountService();
        DepositRequest request = new DepositRequest();
        Guid AccId = new Guid(ddlBankAccounts.SelectedValue.ToString());
        request.AccountId = AccId;
        request.Amount = Decimal.Parse(txtAmount.Text);

        service.Deposit(request);
        DisplaySelectedAccount();
    }
```

```
        protected void btnTransfer_Click(object sender, EventArgs e)
        {
            ApplicationBankAccountService service = new ApplicationBankAccountService();
            TransferRequest request = new TransferRequest();
            request.AccountIdFrom = new Guid(ddlBankAccounts.SelectedValue.ToString());
            request.AccountIdTo = new Guid(ddlBankAccountsToTransferTo
                                                    .SelectedValue.ToString());
            request.Amount = Decimal.Parse(txtAmountToTransfer.Text);

            service.Transfer(request);
            DisplaySelectedAccount();
        }
    }
}
```

最后，将数据库连接字符串添加到 Web 应用程序的 web.config 文件中：

```
    <connectionStrings>
        <add name="BankAccountConnectionString"
             connectionString="DataSource=.\SQLEXPRESS;
             AttachDbFilename=|DataDirectory|\BankAccount.mdf;
             Integrated Security=True;User Instance=True"
             providerName="System.Data.SqlClient"/>
    </connectionStrings>
```

这就是全部代码。启动应用程序，将会看到如图 4-6 所示的屏幕。

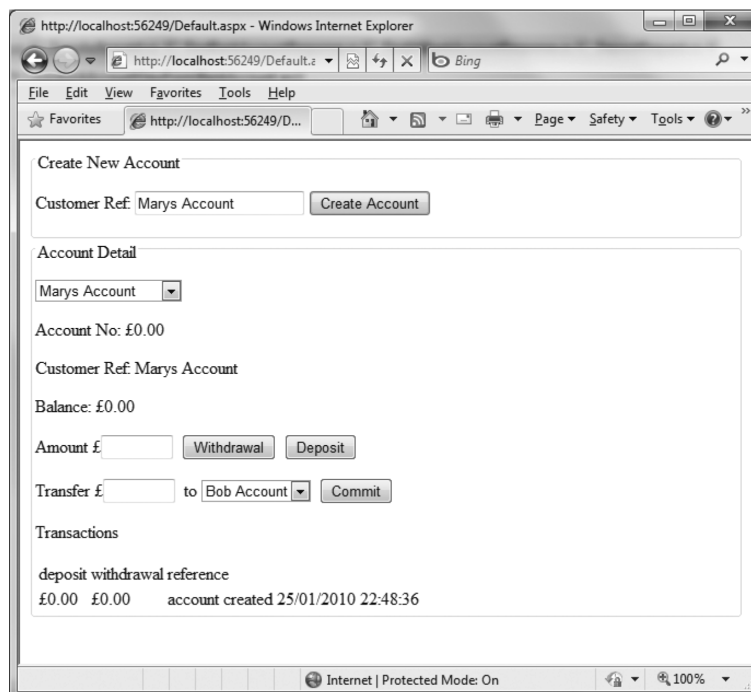

图 4-6

试图在软件中解决复杂业务问题非常困难，但使用 Domain Model 模式时，首先为真实的业务模型创建一个抽象的模型。有了这个模型之后，就可以对复杂逻辑进行建模：追踪真实的领域并在领域模型中重建工作流和处理流程。与 Transaction Script 模式和 Active Record 模式相比，Domain Model 模式的另一个优势是，由于它不包含数据访问代码，因此可以很容易地进行单元测试而不必模拟并隔离数据访问层所依赖的类。另外，Domain Model 模式可能并不总能匹配应用程序需求。它的强大之处在于处理复杂的业务逻辑，但对于只包含非常少量业务逻辑的应用程序而言，采用一个全方位的领域模型有大材小用之嫌。该模式的另一个不足之处在于，与 Active Record 和 Transaction Script 模式相比，为了精通领域模型模式，需要面临陡峭的学习曲线。需要很多时间和经验才能高效地使用该模式，而且最重要的是，需要对正在试图建模的业务领域有全面的了解。

4.1.4　Anemic Domain Model

Anemic Domain Model 有时候被称为一种反模式。初看起来，该模式与 Domain Model 模式非常类似，仍会找到表示业务领域的领域对象。但这些领域对象中不包含任何行为。相反，这些行为位于模型之外，而让领域对象作为简单的数据传输类。

这种模式的主要不足之处在于，领域服务扮演更加过程式的代码，与本章开头看过的 Transaction Script 模式比较类似，这会带来一些与之相关的问题。其中一个问题就是违背了“讲述而不要询问”原则，也就是对象应该告诉客户它们能够做什么或不能够做什么，而不是暴露属性并让客户端来决定某个对象是否处于执行给定动作所需的状态。

如果考虑用于领域模型练习的示例，那么领域对象 Transaction 和 BankAccount 的逻辑现在被剥离出来，它们只是数据容器，代码如下：

```
public class Transaction
{
   public Guid Id { get; set; }
   public decimal Deposit { get; set; }
   public decimal Withdraw { get; set; }
   public string Reference { get; set; }
   public DateTime Date { get; set; }
   public Guid BankAccountId { get; set; }
}

public class BankAccount
{
   public BankAccount()
   {
      Transactions = new List<Transaction>();
   }

   public Guid AccountNo { get; set; }
   public decimal Balance { get; set; }
   public string CustomerRef { get; set; }
   public IList<Transaction> Transactions { get; set; }
}
```

现在，单独的类被包含进来以实现逻辑。可以采用 Specification 模式(将在第 5 章中更详细讨论)来判断某个账号是否有足够的金额来完成提现，代码如下：

```
public class BankAccountHasEnoughFundsToWithdrawSpecification
{
   private decimal _amountToWithdraw;

   public BankAccountHasEnoughFundsToWithdrawSpecification(
                              decimal amountToWithdraw)
   {
      _amountToWithdraw = amountToWithdraw;
   }

   public bool IsSatisfiedBy(BankAccount bankAccount)
   {
      return bankAccount.Balance >= _amountToWithdraw;
   }
}
```

协调提现或银行转账时，在 Domain 模型中创建的领域服务类将利用该规范：

```
public class BankAccountService
{
   ...

   public void Transfer(Guid accountNoTo, Guid accountNoFrom, decimal amount)
   {
      BankAccount bankAccountTo =
              _bankAccountRepository.FindBy(accountNoTo);
      BankAccount bankAccountFrom =
              _bankAccountRepository.FindBy(accountNoFrom);

      BankAccountHasEnoughFundsToWithdrawSpecification HasEnoughFunds =
           new BankAccountHasEnoughFundsToWithdrawSpecification(amount);

      if (HasEnoughFunds.IsSatisfiedBy(bankAccountFrom))
      {
         // … make the bank transfer..
      }
      else
      {
         throw new InsufficientFundsException();
      }
   }

   public void Withdraw(Guid accountNo, decimal amount, string reference)
   {
      BankAccount bankAccount =
              _bankAccountRepository.FindBy(accountNo);
```

```
            BankAccountHasEnoughFundsToWithdrawSpecification HasEnoughFunds =
                      new BankAccountHasEnoughFundsToWithdrawSpecification(amount);

            if (HasEnoughFunds.IsSatisfiedBy(bankAccount))
            {
              // … make the withdraw ;
            }
          }

          ...
        }
```

在 4.1.5 小节中将讨论领域驱动设计，这是一种流行的设计方法学，它专注于业务逻辑而不是基础设施关注点，适合用于具有领域模型模式和复杂业务逻辑的组织。

4.1.5　领域驱动设计

在处理复杂业务逻辑时，Domain Model 模式非常有用。而 DDD(Domain-Driven Design，领域驱动设计)就是一种流行的利用 Domain Model 模式的设计方法学。

简而言之，DDD 就是一组帮助人们构建能够反映业务理解并满足业务需求的应用程序的模式和原则。除此之外，它还是一种思考开发方法学的新方法。DDD 探讨对真实领域建模，首先要全面理解该领域，并将所有的术语、规则和逻辑放入到代码的抽象表示(通常是以领域模型的形式)中。

下面将介绍 DDD 的主要方面，这是本书剩余部分中的大多数练习均要使用的方法学。

1. 通用语言

通用语言(ubiquitous language)的概念是，它应该充当一个公共词汇表，开发者、领域专家及任何其他参与项目的人都使用它来描述该领域。领域专家具有特定领域知识和技能，并且在开发领域模型的过程中与您密切协作，以确保在尝试使用代码表示业务模型之前完全理解该模型。在贷款应用程序中，担保人就可能是领域专家。通过听取此人的讲解，可以构建一个囊括了在申请贷款过程中使用的所有术语的词汇表。所编写的类、方法和属性名称都应该基于同样的通用语言。这可以让您使用领域专家能够理解的语言来谈论代码。此外，接触该代码的新开发者也应该能够了解该领域。它还让他们能够以相对容易的方式与业务专家谈论复杂业务逻辑的哪怕是最细微的细节。当参与应用程序开发的各方都使用相同的语言时，人们就可以容易地表达问题和解决方法，从而让应用程序更快、更容易地构建。

DDD 并不是一个框架，但它确实有一组构建块或概念可供整合到解决方案中。在下面将逐个介绍这些概念。

2. 实体

实体就是 4.1.3 小节中曾经讨论过的事物，如电子商务网站中的订单、客户、商品，博客应用程序中的博客和帖子对象。它们以一种抽象的方式包含了真实实体中的数据和行为。任何与实体相关的逻辑都应该包含在它内部。实体属于需要标识符的事物，在其整个生命周期中，该标识符都将保持不变。考虑贷款应用程序中的借款人。借款人有姓名，但姓名可能变化也可能重复，因此需要添

加一个单独的标识，借款人在该借款应用程序的整个生命周期中该标识将保持不变，无论其姓名、职业或地址改变与否。通常，系统使用某种唯一标识符或自动编号值来为所有无法采用自然方式标识的实体提供标识符。有时候，实体确实具有自然键，如社会保险号或员工号码。并不是领域模型中的所有对象都是唯一的而且需要标识。对于某些对象，数据是最重要的，而不是标识。这些对象就被称为值对象。

3. 值对象

值对象没有标识，它们之所以重要只是因为它们的特性。值对象通常并不会单独存在，它们通常是(但并非总是)实体的属性。回顾 4.1.3 小节中编写的简单银行账号应用程序，应该记得 Transaction 对象没有标识，这是因为它只存在于与之相关的银行账号实体中，它是一个值对象，在其上下文中它本身并不单独存在。

4. 聚合和聚合根

大型系统或复杂领域可能有成百上千的实体和值对象，它们有着错综复杂的关系。领域模型需要一种方法来管理这些关联，更重要的是，在逻辑上属于同一分组的实体和值对象需要定义一个接口，让其他实体能够通过该接口与它们交互。如果没有这类构造，那么以后不同分组对象之间的交互将会相互干扰并产生问题。

聚合概念将逻辑实体和值对象分组。根据 DDD 的定义，聚合只是“一族出于数据变化目的而被视作一个单元的相关联对象”。聚合根是一个实体，它是这个聚合中唯一能够允许聚合外的对象持有引用的成员。DDD 中的聚合概念是为了确保领域模型中的数据完整性。聚合根是一个充当进入聚合的逻辑途径的特殊实体。例如，如果在电子商务商店上下文中获取一张订单，那么可以将其视为聚合根，因为我们只希望通过访问聚合的根来编辑订单项或应用一张凭证。这使得复杂对象图能够保持一致，而且能够遵守业务规则。因此，与其让一个订单对象通过简单的 List 属性来暴露它发出的凭证集合，不如让它拥有一些带有复杂规则的方法，能够允许将凭证应用到它并且把凭证列表表现为一个用于显示的只读集合。

5. 领域服务

在 Domain Model 模式银行账号练习中曾经见到，BankAccountService 类包含在两个银行账号之间转账的逻辑。那些没有真正位于单个实体中或者需要访问资源库的方法都被放到领域服务中。领域服务层还可以包含自己的领域逻辑，而且可以作为领域模型的重要组成部分，像实体和值对象一样。

6. 应用程序服务

应用程序服务是位于领域模型之上的一个瘦层，负责协调应用程序活动。它并不包含业务逻辑，也没有保存任何实体的状态。但它可以存放业务工作流事务的状态。在领域模型银行账号练习中，可以采用 Request-Reply 消息传送模式，使用应用程序服务来提供访问领域模型的 API。

7. 资源库

Repository 模式(将在第 7 章中更详细地研究)充当业务实体的内存集合或仓库，它完全将底层

的数据基础设施抽象出来。该模式可用来将领域模型与任何基础设施关注点分离，使其成为 POCO 和 PI。

8. 分层

在 DDD 中，分层是一种重要的概念，因为它有助于加强关注点的分离。图 4-7 所示为构成 DDD 的各个层次和概念的图形化表示。但应该强调的是，在开发复杂业务应用程序时，DDD 更多地关乎心态，而不是如何建立解决方案。

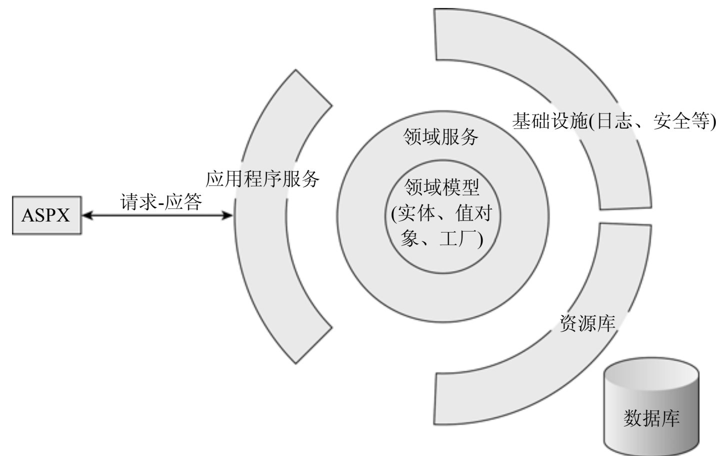

图 4-7

在领域模型练习中编写的银行账号应用程序就是围绕 DDD 概念构建的。图 4-8 给出了银行账号应用程序中的各个层次，以及它们如何与 DDD 的概念相关联。

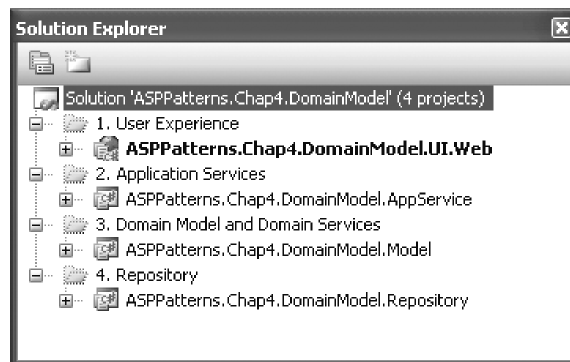

图 4-8

本章只是简单地介绍了 DDD，在案例研究中还将复习，届时还将介绍用户的故事，以方便构建需求并理解正在处理的领域。为了更深入地研究该方法学，推荐阅读下面这两本著作：

- *Domain-Driven Design: Tackling Complexity in the Heart of Software*(Addison-Wesley，2003)，Eric Evans 著
- *Applying Domain-Driven Design and Patterns: Using .Net With Examples in C# and .NET* (Addison-Wesley，2006)，Jimmy Nilsson 著

4.2　小结

本章介绍了一些流行的、经过验证的业务逻辑组织模式。下面是主要的 4 种方法。

- Transaction Script：如果应用程序较小，业务逻辑很少甚至没有业务逻辑，则 Transaction Script 模式就是建立简单解决方案的很好选择，接手代码的开发人员容易理解它。
- Active Record：如果业务层只是位于数据库之上的一个瘦层，则 Active Record 是很好的模式选择。有许多代码生成工具能够自动根据数据库模式创建业务对象，而且手工创建这些对象也并非难事。
- Domain Model：Domain Model 非常适于为棘手的、丰富的、复杂的业务领域建模。它是一种涉及创建真实业务领域抽象模型的纯粹的面向对象方法，在处理复杂逻辑和工作流时非常有用。领域模型是持久化透明的，它依靠映射器类和 Repository 模式来持久化和检索业务实体。
- Anemic Model：Anemic Model 模式是 Domain Model 的反模式。粗略一看它们一样，但进一步研究之后发现，表示正在建模领域的领域对象只是没有行为的数据传输对象。领域的逻辑被放在过程式方法中来验证或检查对象的状态，这违背了第 1 章中讨论的“讲述而不要询问”原则。

在学习了组织业务逻辑层的 4 种主要方法之后，介绍了 DDD(领域驱动设计)设计方法学，它利用领域模型以服务、实体、值对象和聚合的形式来表示复杂逻辑。DDD 还鼓励人们关注业务逻辑和正在建模的领域，它运用 POCO 或 PI 原则来确保不让基础设施关注点污染纯粹的业务领域模型。

本章介绍了如何将 DDD 的概念和构造块应用到银行账号应用程序中，同时也展示了它们如何能够让我们为正在工作的领域建立清晰的模型，无需关注任何基础设施关注点，而在项目、类和方法名称上，应用程序使用的语言与领域所用语言相同。在第 10 章和第 11 章的案例研究中，将介绍如何使用更大型的、更复杂的领域，通过遵循 DDD 原则，将它们很容易地映射到复杂的工作流和业务事务。

第 5 章将考察能够用于企业应用程序的业务层中的模式和原则。