中南大学

硕士学位论文

设计模式研究及应用

姓名：柳小文

申请学位级别：硕士

专业：计算机应用技术

指导教师：杨邦荣

20070517

摘 要

设计模式可以用来解决软件设计过程中反复出现的问题，并且使用设计模式，可以有效地提高软件的可复用性，可靠性和可维护性。

本文介绍了设计模式的定义、描述方法、意义、分类以及程序设计语言与设计模式的关系；分析了 Observer 模式和 Visitor 模式存在的不足，并利用高级程序设计语言的新特性进行改进；分析了 Strategy 模式存在的不足，并结合 Abstract Factory 模式，对其进行优化。最后，提出了一个设计模式选取策略模型，并以实例说明如何应用此模型在设计中选取合适的设计模式。

关键词： 设计模式,模式改进,模式选取策略

ABSTRACT

Design patterns are effective solutions to common problems in Software Development. They are used to improve software reusability, reliability and maintainability.

The article introduces the definition, description approach, significance and classification of the design patterns as well as the relationship between Advanced Programming Language and design patterns. The shortcomings of Observer Pattern and Visitor Pattern are identified in this article. Both can be improved by adopting new features of Advanced Programming Language. Shortcomings of Strategy Pattern are also pointed out in this article and they can be improved by combining it with Abstract Factory Pattern. In the end this article proposes a design pattern selection model to guide pattern selection in software design, and exemplifies how to use it to select a suitable design pattern.

KEY WORDS: design pattern, design pattern improvement, design pattern selection

原 创 性 声 明

本人声明，所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了论文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得中南大学或其他单位的学位或证书而使用过的材料。与我共同工作的同志对本研究所作的贡献均已在论文中作了明确的说明。

作者签名：柳小文　日期：2007年5月17日

关于学位论文使用授权说明

本人了解中南大学有关保留、使用学位论文的规定，即：学校有权保留学位论文，允许学位论文被查阅和借阅；学校可以公布学位论文的全部或部分内容，可以采用复印、缩印或其它手段保存学位论文；学校可根据国家或湖南省有关部门规定送交学位论文。

作者签名：柳小文　导师签名：[illegible]　日期：2007年5月17日

第1章 绪论

1.1 研究背景

随着计算机技术和软件开发技术的不断发展，软件开发中的问题也不断出现，存在着许多低效率、高成本的因素。为了解决这些长期困扰软件业从业人士的问题，提高软件生产的效率和质量，人们不断地探索新的解决途径。从20世纪70年代提出用软件工程的思想解决软件危机以来，学术界以及工程界付诸了许多的努力试图找到解决问题的“银弹”[1]，从极限编程方法到Rational的RUP（Rational Unified Process）方法，但从实际的效果来说，都没有达到这一目的。Fred Brooks在1987年发表的《没有银弹》这篇论文中强调真正的银弹并不存在，这篇经典论文的核心论述通常被解释为复杂的软件工程问题无法靠简单的答案来解决。虽然没有“银弹”，却不意味着我们无法提高软件开发的效率和质量。基于构件的软件复用是当前软件工程的热点之一，这种方法可以有效的复用已有的构件(如控件、类库、函数库、需求分析、设计文档等)。复用可以避免重复设计，提高开发效率；复用可以增强开发产品的可靠性、提高软件质量；复用可以培养客户的使用习惯，减少培训客户的代价。那么怎样实现有效的复用，设计模式是一种有效的手段。

1. 设计模式的提出

程序设计的规模和复杂度不断增加，尤其是对于软件的可复用性和可维护性的要求越来越高，使得程序设计开发面临巨大挑战，而设计模式作为解决程序开发过程中问题的一种有效手段被提了出来。

模式是指在一个特定的背景下反复出现问题的解决方案，它是经验的文档化。在各种设计(建筑、机械、软件等)中，经验是非常重要的，好的经验可以指导我们的工作，节约我们的时间；而坏的经验则可以给我们以借鉴，从而减少失败的风险。但是经验仅仅是工作的积累，很难被记录和传授。为了解决这个问题，建筑学家Alexander首先提出了“模式”的概念[2]。Alexander指出，模式是一个出现在世界上的实物，同时也是一条规则，告诉你应该如何创建一个实物、应该在何时创建。它既是过程，也是实物；既是对当前实物的描述，也是对创建实物的过程的描述。后来模式这个概念逐渐被计算机科学所采用。ErichGamma,RichardHelm, Ralph Johnson和John Vlissides出版了重要著作《设计模式—可复用面向对象软件的基础》。该著作给出了模式的最初分类，他们四人后来就被称作“GoF”四人组。

在面向对象的抽象层次上，设计模式是用来描述在特定的场景中解决一般

设计问题的类和对象。设计模式确定了所包含的类和实例、协作方式和职责分配。

随着软件工程的发展，面向对象的编程思想已得到了广泛的认可。而伴随着面向对象编程思想的发展，设计模式已成为软件工程中一个不可缺少的工具。目前，设计模式在计算机领域的影响已经超过了在建筑界的影响。

软件体系结构的设计模式描述了在特定环境中重复出现的设计问题，并为这种问题提供了一个被证明良好的一般计划。设计模式可以用来构建具有特定属性的软件体系结构。恰当的利用模式来指导软件结构设计，能够起到事半功倍的作用，不仅缩短软件设计和实现的周期，还可以提高软件的重用性、移植性。

2. 设计模式的国内外研究现状

设计模式的研究最早要追溯到八十年代初，那时，Smalltalk 是最流行的面向对象语言，而 C++还处在刚起步阶段，同样在这一时期，结构化编程思想是主流，而面向对象的编程思想还并不被看好。编程框架在当时很流行，随着框架的发展，设计模式开始出现。被引用频率最高的框架是 Model-View-Controller，它将用户接口问题分解成三部分:数据模型、视图和控制者，这正体现了设计模式的优点。

设计模式正式被提出的标志是 Gamma, Helm, Johnson 和 Vlissides 的经典著作《设计模式一可复用面向对象软件的基础》。随着这本书在 1995 年的出版，设计模式的研究状况开始出现了一个高潮。

设计模式的正式研究历史也就十多年，是一个新兴的研究领域，同时也是一个热点研究领域。设计模式起源于国外，目前的主要研究工作是在国外展开。现在，国外的软件工程界正在把设计模式应用于软件体系结构、设计、编码和开发的过程和组织中去，其理论成果主要体现在从 1994 年开始的“程序设计模式”年会论文集中[4]。国内对于设计模式的研究和应用时间都很短，尚属起步阶段。

当前，国内外针对设计模式的研究工作主要集中在以下几个方面[5]：

(1) 新设计模式的发现与研究

每年国际上都举办包括 PloP, ECOOP 等在内的学术会议对设计模式进行讨论，人们正在各个领域总结设计模式，如通信领域、Web 开发领域。设计模式与其他面向对象技术的关系也是当前研究的热点。

(2) 设计模式的应用及具体化

虽然设计模式有很多，并且设计模式的提出也是针对所有领域的，但在使用时，并不是所有的模式都可以拿来直接应用，需要对它进行改进，才能更加适合于该应用，所以设计模式的应用和改进也是当前研究的热点。

(3) 模式的组织和索引

面对模式数量的增长，需要一种合适的覆盖所有模式的组织方法，并对其

中的模式提供一个索引，建造一个可以有效的支持高质量软件开发的真正成熟的模式系统。

(4) 形式化模式

人们提出了两类设计模式结构的描述方法，即图示化方法和形式化数学表示法，目前的研究趋势是将这两类方法逐步靠拢。

(5) 设计模式的使用自动化和设计模式支持工具。

(6) 设计模式和其他面向对象技术的关系。

在我国，关于设计模式的研究相对滞后，而且多为已有设计模式的应用讨论，同样，国内有关设计模式的书籍也大都为译作。从这点看来，要赶上世界上软件发达的国家，我们还有很长的路需要走。但是随着设计模式在中国的应用越来越广泛，国内的开发人员必将更多的参与到应用设计模式的项目开发中。随着大师们的设计模式思想逐渐深入人心，中国设计模式的研究也必将走上新的台阶。

1.2 研究内容

1. 设计模式的定义、意义和常用设计模式

将以往的设计经验总结出来用于解决新的设计问题，这些设计经验就是设计模式。但不是所有的经验总结都可以成为设计模式，设计模式一般应具有4个基本要素：

- 模式名称（pattern name）：它用一两个词来描述模式的问题、解决方案和效果。
- 问题（problem）：描述应该在何时使用模式。
- 解决方案（Solution）：描述设计的组成部分，它们之间的相互关系及各自的职责和写作方式。
- 效果（consequences）：描述模式应用的效果及使用模式应权衡的问题。

人们如此热衷于研究设计模式，是因为它能给我们的设计带来帮助，也就是说，设计模式为设计者交流讨论、书写文档以及探索各种不同设计提供了一套通用的设计词汇。学习这些设计模式将有助于我们理解已有的面向对象系统，提高我们的设计水平，最重要的是设计模式能够有效的支持变化，增强设计的可复用性，提高软件设计的效率和可靠性。

一般，我们将设计模式划分为创建型、构造型和行为型三类模式。本文选取了一个模式按照描述模式的方法从模式的意图、动机、适用性、参与者、效果、实现、代码示例这几个方面进行介绍。

2. 设计模式与程序设计语言的关系

模式本身是不依赖于编程语言（与之相反，语法是编程语言特有的）。但是不同的语言有各自的特点，本文以 Observer 模式和 Singleton 模式为例，比较同一模式采用不同编程语言的实现方式和实现效果，印证了在不同应用场合选择适合的编程语言的必要性。同时，也从侧面说明模式的重要性，以至于不断发展的高级程序设计语言把经过验证的设计模式集成到了语言本身中，或者设计新的特性来更好的完善和实现模式。

3. 设计模式的改进

设计模式是对以往设计经验的总结，那么随着实践的深入，我们会不断地发展已有的设计，解决其中的缺陷和不足，总结经验，形成新的设计模式。我们利用 C#等高级语言的反射机制解决了 Visitor 模式违反面向对象程序设计的开闭原则的不足；利用 C#的 delegate 机制使 Observer 模式获得更大的灵活性；通过引入 Abstract Factory 模式对 Strategy 模式进行改进，解决了其违反封装性原则的不足。

4. 如何应用设计模式

提出一个如何在设计中选用设计模式的模型，以帮助提高软件项目应对变化的能力。主要内容包括：（1）选取模式的规则和设计模式的选取策略；（2）以一个软件项目为例，阐述了在该项目中使用了哪些设计模式，选取的策略以及实现手段。

1.3 论文组织结构

本论文共分为六章，各章的主要内容如下

第一章、绪论：本章首先介绍了研究的背景，指出研究设计模式的重要性，其次介绍了本课题的主要研究内容，最后介绍了论文的主要内容与组织结构。

第二章、设计模式：本章主要介绍了设计模式的定义、意义、描述方法及分类，并选取了 Factory Method 加以介绍。

第三章、程序设计语言与模式：本章对 Observer 模式和 Singleton 模式在不同的程序语言下的实现方式加以比较，说明了模式和程序设计语言相辅相成、互相促进的关系，以及在不同应用场合选择适合的编程语言的必要性。

第四章、设计模式的优化：本章分析了 Visitor 模式、Observer 模式和 Strategy 模式的不足之处，并对它们进行了改进。

第五章、设计模式在软件设计中的应用：本章提出了一个设计模式的选取策略，并通过一个实际的软件项目应用，进一步解析了设计模式的选取问题。

第六章、总结与展望：本章对本文所作的工作进行了总结，指出尚存在的问题与不足，对以后工作进行展望。

第2章 设计模式

2.1 什么是设计模式

模式是一种方案，利用这种方案，我们可以完成某项工作；模式也是一种途径，通过这种途径，我们可以达到某个目的；同时，模式也是一种技术，我们必须获取并利用有效的技术。设计模式也是一种模式，是一种完成某个目的或构思的方案。它要求使用某种面向对象提供的类及相关机制[29]。

一般而言，一个模式有四个基本要素。

1. 模式名称（pattern name）：它一个助记名，用一两个词来描述模式的问题、解决方案和效果。模式名可以帮助我们思考，便于我们与其他人交流设计思想及设计结果。

2. 问题（problem）：它描述了应该在何时使用模式，解释设计问题和问题存在的前因后果；它可能描述了特定的设计问题，如怎样使用对象表示算法等；也可能描述了导致不灵活设计的类或对象结构。有时候，问题部分会包括使用模式必须满足的一系列先决条件。

3. 解决方案（solution）：描述了设计的组成部分，它们之间的相互关系及各自的职责和协作方式。因为模式就像一个模板，可应用于多种不同场合，所以解决方案并不描述特定而具体的设计或实现，而是提供设计问题的抽象描述和怎样用一个具有一般意义的元素组合（类或对象组合）来解决这个问题。

4. 效果（consequences）：描述了模式应用的效果及使用模式应权衡的问题。尽管我们描述设计决策时，并不总提到模式效果，但它们对于评价设计选择和理解使用模式的代价及好处具有重要意义，软件效果大多数关注对时间和空间的衡量，它们也表述了语言和实现问题。因为复用是面向对象设计的要素之一，所以模式效果包括它对系统的灵活性，扩充性或可移植性的影响。

2.2 使用设计模式的意义

根据我们日常使用设计模式的经验，我们认为它将在以下几个方面对面向对象软件的设计方式产生影响。

（1）设计模式使我们得到一套通用的设计词汇

设计模式为设计者交流讨论、书写文档以及探索各种不同设计提供了一套通用的设计词汇。设计模式使我们可以在比设计表示或编程语言更高的抽象级别上谈论一个系统，从而降低了其复杂度。设计模式也提高了我们的设计及讨论这些

设计的层次。

（2）设计模式是书写文档和学习的辅助手段

了解各种设计模式可使我们更容易理解已有的系统。大多数规模较大的面向对象系统都使用了这些设计模式。

设计模式也能提高我们的设计水平。它们为我们提供一些常见问题解决方案，而且，按照一个系统的使用模式来描述该系统可以使其他人理解起来容易得多，否则就必须对该系统进行逆向工程来弄清其使用的设计模式。

设计模式是现有方法的补充。面向对象设计方法可以用来促进良好的设计，以及对设计活动进行标准化。一个设计方法通常定义了一组（常常是图形化的）用来为问题各方面进行建模的记号（notation），以及决定在什么情况下以什么样的方式使用这些记号的一组规则。设计方法通常描述一个设计中出现的问题，如何解决这些问题，以及如何评估一个设计。

一个成熟的设计不仅要有设计模式，还可有其他类型模式，如分析模式、用户界面设计模式，或者性能调节模式。但是设计模式是最主要的部分。

(3)设计模式可以支持变化，增强设计的复用性

获得最大限度复用的关键在于对新需求和已有需求发生变化时的预见性，要求我们的系统设计要能够相应地改进。

为了设计适应这种变化、且具有健壮性的系统，我们必须考虑系统在它的生命周期内会发生怎样的变化。一个不考虑系统变化的设计在将来就有可能需要重新设计。这些变化可能是类的重新定义和实现，修改客户和重新测试。重新设计会影响软件系统的许多方面，并且未曾料到的变化总是代价巨大的。

设计模式可以确保系统能以特定的方式变化，从而帮助我们避免重新设计系统。每一个设计模式允许系统结构的某个方面的变化独立于其他方面，这样产生的系统对于某一种特殊变化将更健壮。

2.3 设计模式的描述

设计模式的描述方法有自然语言描述法、统一标记语言(UML) 描述法、形式化语言描述法等。 自然语言描述法比较简单、方便,但在现实与设计之间的过渡描述不够流畅。 对象建模技术(OMT) 描述法是利用类图和对象图对设计模式中的类、实例以及整体模式结构进行图形描述的方法,而 UML 是在 OMT 基础上进一步发展起来的,其描述更加清晰和统一，符合大部分软件设计人员的习惯，也便于设计人员的理解和应用。 形式化语言主要包括 DisCo 、LePUS、LayOM、ADV/ADO、CDL 、PDL 、PDSP 等,其中 DisCo 侧重于描述设计模式中参与者的交互行为。

按《设计模式——可复用面向对象软件的基础》一书中描述设计模式的方法，每一个模式根据以下的模板被分成若干部分。模板具有统一的信息描述结构。

（1）模式名:模式名简洁的描述了模式的本质。

（2）意图:是回答下列问题的简单陈述：设计模式是做什么的？它的基本原理和意图是什么？它解决的是什么样的特定设计问题？

（3）别名:模式的其他名称。

（4）动机:用以说明一个设计问题以及如何用模式中的类、对象来解决该问题的特定情景。

（5）适用性:什么情况下可以使用该设计模式？该模式可用来改进哪些不良设计？怎样识别这些情况？

（6）结构:采用基于对象建模技术（OMT）的表示法对模式中的类进行图形描述。我们也使用了交互图来说明对象之间的请求序列和协作关系。

（7）参与者:指设计模式中的类或对象以及它们各自的职责。

（8）协作:模式的参与者怎样协作以实现它们的职责。

（9）效果:模式怎样支持它的目标？使用模式的效果和所需要做的权衡取舍？系统结构的哪些方面可以独立改变？

（10）实现:实现模式时需要知道的一些提示、技术要点及应避免的缺陷，以及是否存在某些特定于实现语言的问题。

（11）代码示例:该模式的示范性的代码片断。

（12）已知应用:实际系统中已经发现的模式的例子。

（13）相关模式:与这个模式紧密相关的模式有哪些？其间重要的不同之处是什么？这个模式应与哪些模式一起使用？

2.4 设计模式的分类

根据设计模式在粒度和抽象层次上各不相同，存在着众多的设计模式，我们按照模式的目的（即模式是用来完成什么工作的）对模式进行分类，从而更好地分析和使用模式。模式依据其目的可以分成创建型（Creational）、结构型（Structural）、行为型（Behavioral）三种。

创建型模式：它与对象的创建有关，如 Factory Method 模式，Abstract Factory 模式，Prototype 模式，Singleton 模式等。

结构型模式：用于处理类和对象的组合，如 Adapter 模式，Bridge 模式，Composite 模式，Decorator 模式，Façade 模式，Flyweight 模式，Proxy 模式。

行为型模式：它对类或对象怎样交互和怎样分配职责进行描述，包括 Interpreter 模式，Template Method 模式，Command 模式，Iterator 模式，

Mediator 模式，Observer 模式，State 模式，Memento 模式，Strategy 模式，Visitor 模式。

本文选取创建型的 Factory Method 模式，按照意图、结构、动机、适用性、效果、应用和代码示例这几个方面加以介绍。

许多时候，并不希望软件知道它所需要具体化的实例化是哪个类，那么对于这样的要求我们可以采用以下的工厂方法来实现。

1. 意图

定义一个用于创建对象的接口，让子类决定实例化属哪一个类，使一个类的实例化延迟到其子类。

2. 结构

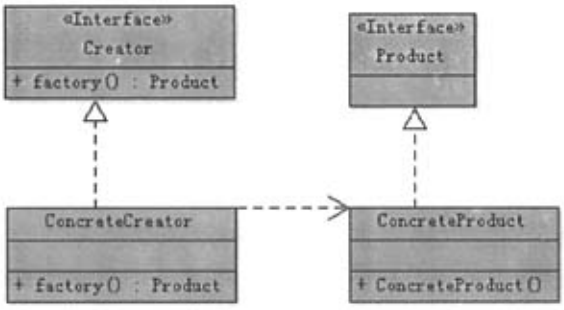


图 2-1 Factory Method 模式结构图

3. 动机及应用实例

有一个采用面向对象构建的信息管理系统。在原始设计中，数据库的连接信息包括数据库名称、数据库服务器、用户名、密码都保存在注册表里。在主模块里有一个类负责从注册表中读取这些信息，以实现对数据库的访问。但实际情况是数据库连接信息的保存方式随操作系统和用户需求的不同而会有所不同，有可能会保存在 xml 文件或者 ini 文件中。如果对于不同的需求，都重新设计相应的类，显然这样的设计是不可取的。对于这种情况，我们可以采用 Factory Method 的模式加以解决。

如图 2-2 所示，DatabaseInfo 定义工厂方法所创建的对象的接口，它提供一个方法返回由数据库信息生成的连接字符串。PickUp 接口就是工厂方法模式中的 Creator，声明了一个工厂方法 CreatInfo()，该方法返回一个 DatabaseInfo 类型的对象。XmlPickup 实现 Pickup 接口，它的 CreatInfo 方法，返回 XmlDatabaseInfo 的实例。而 XmlDatabaseInfo 类实现从 xml 文件存储的信息中提取数据，并生成返回数据库所需的连接信息。RegDatabaseInfo 类实现从注册

表里存储的信息中提取数据库连接字符串。如果要实现更多的存储方式，只需要定义一个类实现 DatabaseInfo 接口。这样就将具体的数据库的提取方式延迟到子类实现。并且由于接口依赖，需求的变化对系统的影响很小，我们在不改变系统设计的情况下最大程度实现了系统复用。

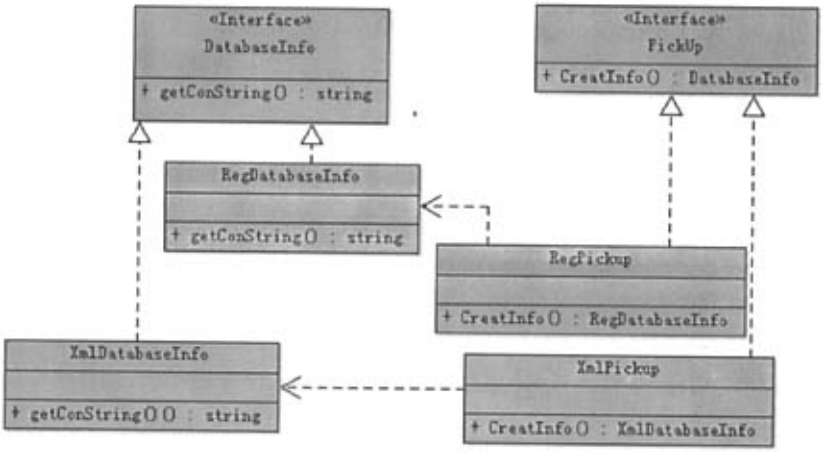


图 2-2 基于 Factory Method 模式的数据库连接信息结构图

部分实现的代码：

```
using System;
using System.Xml; //操作xml文件用到的库
using Microsoft.Win32 ;//操作注册表用到的库
namespace Robin
{
    public interface DatabaseInfo
    {
        string getConString();
    }
    public interface Pickup
    {
        DatabaseInfo CreatInfo();
    }
    public class XmlPickup :Pickup
    {
        public XmlPickup()
        {
        }
        public DatabaseInfo CreatInfo()
        {
```

```
            return new XmlDatabaseInfo ();
        }
    }
    public class RegPickup :Pickup
    {
        public RegPickup()      {       }
        public DatabaseInfo CreatInfo()
        {
            return new RegDatabaseInfo ();
        }   }
    public class XmlDatabaseInfo :DatabaseInfo
    {
        public XmlDatabaseInfo()        {   }
        public string getConString()
        {
            return getInfofromXml();
        }
        private string getInfofromXml()
        {
            string strInfo;
            //从xml文件中读取连接信息
            ....//省略部分代码
            return strInfo;
        }
    }
    public class RegDatabaseInfo :DatabaseInfo
    {
        public RegDatabaseInfo()        {       }
        public string getConString()
        {
            return getInfofromReg();
        }
        private string getInfofromReg()
        {
            string strInfo;
            //从系统注册表中读取连接信息
            ....//省略部分代码
            return strInfo;
        }
    }
}
```

启动部分代码:

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

```
    private DatabaseInfo info;
    private Pickup pick;
    public Form1()
    {
        string strcon;
        pick=new XmlPickup ();//如果想更加灵活的选择提取器,可以做一个抽象工厂方法
        info=pick.CreatInfo ();
        strcon=info.getConString ();          }
    static void Main()          {
        Application.Run(new Form1());
    }
}
}
```

4. 实现及效果

(a)可以采用两种不同的方法来设计 Creator。方法一：仅仅声明一个 Creator 抽象类或接口并不提供声明的工厂方法的实现。方法二：Creator 是一个具体的类，且为工厂方法提供一个缺省的实现。

(b)使用参数化工厂方法——通过传递一个标识(参数)来指定要创建的产品的种类。重新定义一个参数化的工厂方法使得可以简单而有选择性的扩展或者改变一个 Creator 生产的产品。例如可以为新的产品引入新的标识或将已有的标识与不同的产品相关联。

(c)在 C#和 Java 等支持反射机制的语言中，可以利用反射机制实现参数化工厂方法，避免了在构造函数中通过条件判断语句创建指定的产品，降低代码的复杂性。

(d)使用工厂方法比直接生成对象更灵活。

5. 适应性

在下列情况下可以使用 Factory Method 模式。

(a)当一个类不知道它必须创建的对象的类的时候。

(b)当一个类希望由它的子类来指定它所创建的对象的时候。

(c)当类将创建对象的职责委托给多个帮助子类中的一个，并且系统将哪一个帮助子类是代理这一信息局部化的时候。

Factory Method 模式经常出现在客户代码中，特别是如果我们不希望客户知道应该对哪个类进行实例化，那么我们可以应用这个模式。在 C#语言里，大量用到了多种设计模式（这个在后面的章节也会谈到），C#的迭代器就运用了工厂模式，使得客户在使用迭代器时并不需要知道应该要对哪一个实例化。另外，在获取数据库连接信息的例子，为了适应系统可能发生的变化，我们把创建数据库连接对象的提取器的任务交给了 DatabaseInfo 的子类，从而获取更大的灵活

性。

2.5 本章总结

设计模式是对解决问题的方法的高层次的抽象。“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，我们就能一次又一次得使用该方案而不必做重复劳动”[4]。本章详细阐述了使用设计模式的目的和意义，设计模式的描述的方法，设计模式的分类。并抽取出一种有代表性的模式加以介绍。

第3章 程序设计语言与模式

3.1 概述

模式是不依赖于编程语言的[3]。从某种程度上讲，模式构成了一种语言，它比编程语言更进了一步，使开发人员可以彼此交流设计思想。然而，认识到特定的编程语言在软件系统的发展过程中扮演着重要角色是非常重要的。现在，我们必须考虑工具、库以及他们提供的通用环境，特别是当我们考虑使用象 Java 和 C#这样的新的面向对象程序设计语言时。这些语言不但提供了传统的编程语言语法，而且还构造在虚拟机之上，并且带有完整而丰富的库或包，使得开发和重用更加容易。尽管可以用任何语言实现任何设计模式（根据定义），但是对于特定的模式，用某些语言比用其他语言更容易实现。比如说，如果某种模式建立在多线程控制的基础上，在 C++或 Visual Basic 中实现跨越线程边界的消息传递就比在 Java 中困难一些，也就是说，如果 C++使用线程库（例如 Pthreads）或者 Visual Basic 使用底层 windows 调用也可以完成跨越线程边界的消息传递，但是实现起来却要麻烦的多。

3.2 Observer 模式在不同语言中的实现

我们经常会碰到这样的问题,当需要在软件中处理多个对象时，这些对象依赖于另一个对象的状态。例如，在一个记账系统中，我们可能会为同一笔数据提供多种视图，包括多个电子表格视图和扇形或者柱状图之类的图形视图。当我们在某个数据表中修改数据时,用户完全有理由要求系统中的其它视图也相应发生改变，而且这个过程应该是自动的，不需要用户选择“刷新”选项；在一个嵌入式系统中，也可能遇到类似的问题：多个对象（例如监视器）通过软件封装或设备驱动程序监控同一个硬件设备，当发生中断（例如电力不足）时，这些对象必须快速的做出反应。有一种模式能够描述这种情况，并以一种容易移植的方式来解决这个问题，即使用 Observer 模式。Observer 模式属行为模式，一般用于维护相关对象的一致性，即当一个对象的状态发生改变的时候，所有依赖于它的对象都会得到通知而自动被更新。

下面我们以 Observer 模式在一个简化的记账系统中的应用来说明不同的程序设计语言在实现具体设计模式的区别。

3.2.1 Observer 模式的 Java 语言实现

在 Java 语言的 java.util 库里面，提供了一个 Observable 类以及一个 Observer 接口，构成 Java 语言对 Observer 模式的支持。

1. Observer 接口

这个接口定义了一个 update()方法，当被观察对象的状态发生变化时，这个方法就会被调用。并且当调用这个方法时，将调用每一个被观察者对象的 notifyObservers()方法，从而通知所有的观察对象。

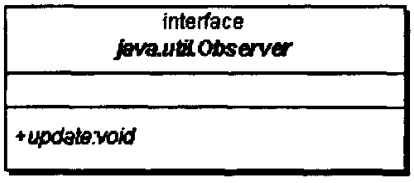


图 3-1 java.util 提供的 Observer 接口的类图。

2. Observable 类

被观察者类都是 java.util.Observable 类的子类。java.util.Observable 提供公开的方法支持观察者对象，这些方法中有两个对 Observable 的子类非常重要：一个是 setChanged()，另一个是 notifyObservers()。setChanged()被调用之后会设置一个内部标记变量，代表被观察者对象的状态发生了变化。notifyObservers()被调用时，会调用所有登记过的观察者对象的 update()方法，使这些观察者对象可以更新自己。

java.util.Observable 类还有其它的一些重要的方法。比如，观察者对象可以调用 java.util.Observable 类的 addObserver()方法，将对象一个一个加入到一个列表上。当有变化时，这个列表可以告诉 notifyObservers()方法那些观察者对象需要通知。由于这个列表是私有的，因此 java.util.Observable 的子对象并不知道观察者对象一直在观察着它们。

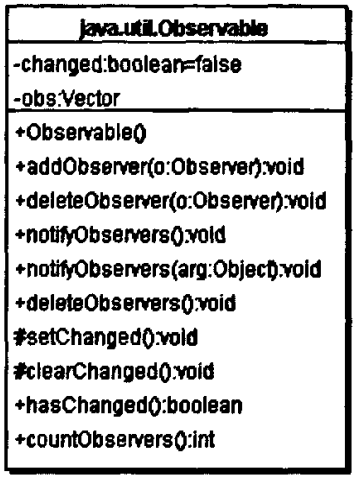


图 3-2 Java 语言提供的被观察者的类图

这个 Observable 类代表一个被观察者对象。一个被观察者对象可以有数个观察者对象，一个观察者可以是一个实现 Observer 接口的对象。在被观察者对象发生变化时，它会调用 Observable 的 notifyObservers()方法，此方法调用所有的具体观察者的 update()方法，从而使所有的观察者都被通知更新自己，如图 3-3 所示。

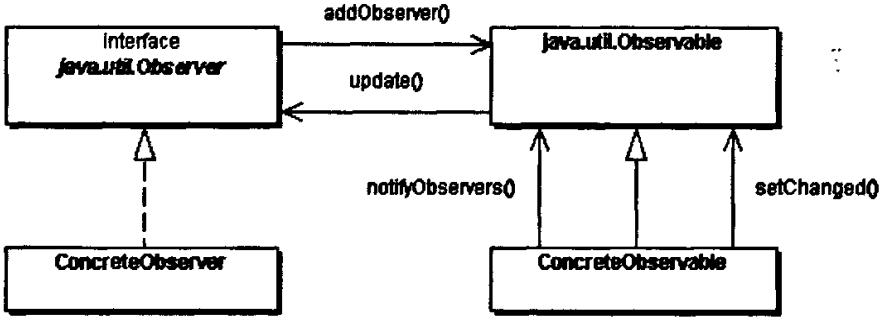


图 3-3 使用 Java 语言提供的对观察者模式的支持

发通知的秩序在这里没有指明。Observerable 类所提供的缺省实现会按照 Observers 对象被登记的秩序通知它们，但是 Observerable 类的子类可以改变这一秩序。子类可以在单独的线程里通知观察者对象；或者在一个公用的线程里按照秩序执行。

3. Observer 模式的 Java 实现代码

Account

```
Package JavaObserver;
import java.util.*;
public class Account extends Observable
{
    public void addEntry(Date entryDate,Duble amout)
    {
        AccountEntry ae=new AccountEntry(entryDate,amount);
        m_entries.addElement(ae);
        setchanged();
    }
    public Enumeration getEntries()
    {
        return m_entries.elements();
    }
    private Vector m_entries;
}
AccountEntry
Package JavaObserver;
import java.util.Date;
public class AccountEntry
{
    AccountEntry(Date entryDate,double amount)
    {
        m_entryDate=entryDate;
        m_amount=amount;
    }
    public Date getEntryDate()
    {
        return m_entryDate;
    }
    public double getAmount()
    {
        return m_amount;
    }
    private  Date m_entryDate ;
    private  double m_amount;
}
AccountObserver
Package JavaObserver;
import java.util.*;
public class AccountObserver implements Observer
{
    public AccountObserver (Account account)
    {
```

```
        m_account=account;
        m_account.addObserver(this);
    }
    public void sync()
    {
        m_accoun.notifyObservers();
    }

    public void update(observable observer,object unused)
    {
        Account acct=(Account)observer;
        For(Enumeration e=acct.getEntries();e.hasMoreElements();)
        {
            AccountEntry entry=(AccountEntry)e.nextElement();
            System.out.println(entry.getEntryDate().toString() + ":"
                + (new Double(entry.getAmount().toString());

        }
    }
    private Account m_account;
}
```

3.2.2 Observer 模式的 Visual Basic 实现

如果用 Visual Basic 来实现上面的例子，实现方式和效果上就有很大的不同了。我们知道 Visual Basic 没有类继承机制，这使得 Visual Basic 在实现需要利用继承特性实现的设计模式时存在先天的不足，必须为每一个对象定一个类来继承接口，增加了实现模式的难度。接下来我们分析在 Visual Basic 中如何用 Observer 模式来解决这个问题。

1. AccountObserver 类模块

```
'定义观察者接口
Public Sub Update(ByVal maccountt As Account)

End Sub
'AccountEntry 类模块
'AccountEntry 类
Option Explicit
Private mvarentryDate As Date
Private mvarmount As Currency
Public Property Get amount() As Currency
    amount = mvarmount
End Property
Public Property Let amount(ByVal vData As Currency)
```

```
    mvarmount = vData
End Property
Public Property Get entryDate() As Date
    entryDate = mvarentryDate
End Property
Public Property Let entryDate(ByVal vData As Date)
    mvarentryDate = vData
End Property
```

2. Account 类模块

```
'Account 为目标,这里省略了接口.
Private mvarAccountEntryCollection As Collection
Private mvarObserverCollection As Collection
Private mvarccountName As String
Public Property Let accountName(ByVal vData As String)
    mvarccountName = vData
End Property
Public Property Get accountName() As String
            accountName = mvarccountName
End Property
Public Property Get getEntries() As Collection
    Set getEntries = mvarAccountEntryCollection
End Property
Public Sub addEntry(entryDate As Date, amount As Currency)
     '添加一个 AccountEntry 对象
End Sub
Public Sub notify()  '通知观察者内容发生变化了
    Dim observer As AccountObserver
    For Each observer In mvarObserverCollection
        Call observer.Update(Me)
    Next
End Sub
Public Sub detachObserver(key As String)
    Call mvarObserverCollection.Remove(key)
End Sub
Public Sub attachObserver(observer As AccountObserver, key As String)
    Call mvarObserverCollection.Add(observer, key)
End Sub
```

3. 窗体模块 FrmObserver1

```
'FrmObserver1 继承并实现了 Observer 接口
Implements AccountObserver  '继承 Observer 接口
Dim Accountfrm As Form
Public Sub setForm(ByRef mFrm As FrmAccount)
    Set Accountfrm = FrmAccount
```

```
End Sub
Private Sub AccountObserver_Update(ByVal maccount As Account)
'实现 Observer 接口，显示更新的内容
End Sub
```

4. 窗体模块 FrmAccount

```
'FrmAccount 为 client,拥有主体对象 testAccount ,并在窗体加载时添加了两个'观察者
frmOberver1 和 frmObserver2
Private testAccount As New Account
Private Sub CmdAdd_Click()
...'被观察者的内容发生变化
    Call testAccount.notify '通知更新
End Sub
Private Sub Form_Load()
    '添加观察者视图 1，添加观察者视图 2 代码略
End Sub
```

3.2.3 Observer 模式不同语言实现的效果比较

Java 语言为实现 observer 模式专门提供了一个包，包中有实现此模式的类及接口的声明。设计者只需要引入包，并作简单扩展，即可实现 Observer 模式在不同环境下的应用，降低模式应用的门槛，充分发挥此模式的长处。而 Visual Basic 语言在实现 Observer 模式时存在一些先天不足，如 Visual Basic 没有类继承机制，所有通过继承可以实现的扩展，需要改为接口或者“组合”来实现；Visual Basic 中容器的实现也比较复杂。目标（被观察对象）需要保存观察者的引用，在 Visual Basic 中是使用集合类添加、删除对象，通过硬编码在模块中实现，不易于修改。

3.3 Singleton 模式在不同语言中的实现

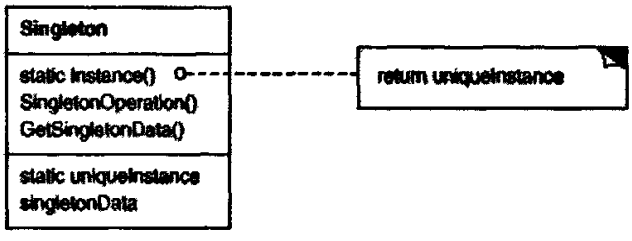


图 3-4 Singleton 模式结构图

3.3.1 Singleton 模式的 C++实现

Singleton 模式使得类的唯一实例是类的一般实例，但该类被写成只有一个

实例能被创建。做到这一点的常用方法是将创建这个实例的操作隐藏在一个类操作中（即一个静态成员函数或一个类方法后面）。由它保证只有一个实例被创建。这个操作可以访问保存唯一实例的变量，而且它包含了保证这个变量在返回值之前用这个唯一实例初始化。这种方法保证了单件在它首次使用前被创建和使用。在C++中可以用Singleton类的静态成员函数instance来定义这个操作。

```
class Singleton
{
public:
        static Singleton* Instance()
        {
                if(_instance==0)
                {
                        _instance=new Singleton();
                }
                return _instance;
}
protected:
      Singleton();
private:
        static Singleton* _instance;
};
```

3.3.2 Singleton模式的Visual Basic实现

Singleton模式如果采用Visual Basic来实现，存在一定问题。Visual Basic的类没有protect类型，也没有构造函数。因此类似C++的实现方法是行不通的。Visual Basic实现Singleton模式可以采用进程外组件，利用组件的唯一实例；或者利用全局变量实现。下面介绍如何用全局变量实现Singleton模式。

```
Option Explicit
Private m_Singleton As SingletonClass
Private m_SingletonCreationOk As Boolean
'返回对SingtonClass对象的引用
Public Property Get Singleton() As SingletonClass
    If m_Singleton Is Nothing Then
        ' Flag that this instantiation is ok.
        m_SingletonCreationOk = True
        Set m_Singleton = New SingletonClass
        m_SingletonCreationOk = False
    End If
    Set Singleton = m_Singleton
End Property
```

```
'表示是否正在创建
Public Function SingletonCreationOk() As Boolean
    SingletonCreationOk = m_SingletonCreationOk
End Function

'Singletonclass类模块
Option Explicit
' 如果已经存在则触发一个错误
Private Sub Class_Initialize()
    If Not SingletonCreationOk Then
        Err.Raise vbObjectError + 1001, _
            "SingletonClass", _
            "Illegal SingletonClass instantiation"
    End If
End Sub
'测试:
Option Explicit
Private Sub cmdCheat_Click()
Dim obj As SingletonClass
    Set obj = New SingletonClass
    MsgBox TypeName(obj)
End Sub
Private Sub cmdCreateLegally_Click()
Dim obj As SingletonClass
    Set obj = Singleton
    MsgBox TypeName(obj)
End Sub
```

3.3.3 Singleton模式的C#实现

接下来我们看在C#中怎样实现Singleton模式。C#的独特语言特性使得C#拥有实现Singleton模式的独特方法。下面给出利用C#语言新特性实现Singleton模式的代码:

```
sealed class Singleton
{
   private Singleton();
   public static readonly Singleton Instance=new Singleton();
}
```

与前面的两种语言相比较，代码减少了，变得更加简洁明了。那么它又是怎样工作的呢?

注意到，Singleton类被声明为sealed，以此保证它自己不会被继承，其次

和C++的实现相比较没有了 Instance 的方法，将原来_instance 成员变量类型声明为 public readonly，并在声明时被初始化。通过这些改变，我们也能实现 Singleton 模式。原因是在 JIT（Just-In-Time，运行编译技术）的处理过程中，如果类中的 static 属性被任何方法使用时，.NET Framework 将对这个属性进行初始化，于是在初始化 Instance 属性的同时 Singleton 类实例得以创建和装载。而私有的构造函数和 readonly(只读)属性保证了 Singleton 不会被再次实例化。

3.3.4 Singleton 模式不同语言实现效果的比较

采用 Singleton 模式的意图是为了保证一个类仅有一个实例，并提供一个访问它的全局访问方法。C++与 Java 之类的面向对象语言实现的方式是通过将实例化过程的操作隐藏在类的静态成员函数后面。而 Visual Basic 由于没有 Protected 类型修饰符，且没有构造函数。在实现 Singleton 模式时是通过将单件定义为全局的对象来确保类的唯一实例.这样做存在以下两个问题：1.不能保证单件类只有一个实例被声明。2.无法直接子类化此单件。而 C#除了能像 C++那样实现 Singleton 模式，还可以利用新的 readonly 属性修饰符给出 Singleton 模式更简洁的方式。通过 Singleton 模式的三种不同语言的实现方法的比较，可以看到，随着程序设计语言的发展，在具备新特性的新的程序设计语言中，Singleton 模式更加易于实现。

3.4 本章总结

从以上两个设计模式采用不同的设计语言来实现的效果与难易程度，我们不难得出结论，新的高级面向对象程序设计正在逐步地将模式直接纳入语言体系之中，或者为实现模式提供了更多的便利。许多程序设计语言提供的框架性平台中也直接使用设计模式，这使得使用此程序语言实现系统时具有更大的灵活性和可靠性。由此，我们可以得出一个结论，模式的发展在程序设计语言的发展过程中起着重要的作用，模式与程序设计语言相辅相成，共同促进。

第4章 设计模式优化

设计模式是记录、提炼存在于软件开发人员头脑中或文档中的一些反复出现的共性问题及其经过多次验证的成功解，它表达了在特定上下文情形下产生的、反复出现的典型性问题以及相应的一整套解决方案之间的关系[41]，是对以往成功的设计经验的总结，所以并不是僵硬的不变的东西，它随着设计经验的丰富、新的设计实践的展开、新的开发手段的诞生而被优化改进——优点得以发挥、缺点被削弱直至消除。下面就经典设计模式中几个模式的分析与改进，给出一些模式优化、改进的方法及手段。

4.1 Visitor 模式研究

4.1.1 Visitor 模式介绍

Visitor 模式属于行为型模式的一种，与大多数设计模式不同的是，Visitor 模式包含了两个类层次关系（Element 和 Visitor），也是较为复杂的一种模式。Visitor 设计模式基于这样一个应用环境：在不改变对象集合中元素的前提下，定义对这些元素的新操作。Visitor 模式中的 Visitor 接口类（抽象类）代表了一个施加于对象集合中元素的操作，Visitor 模式也由此而得名。它的结构如图 4-1 所示 。

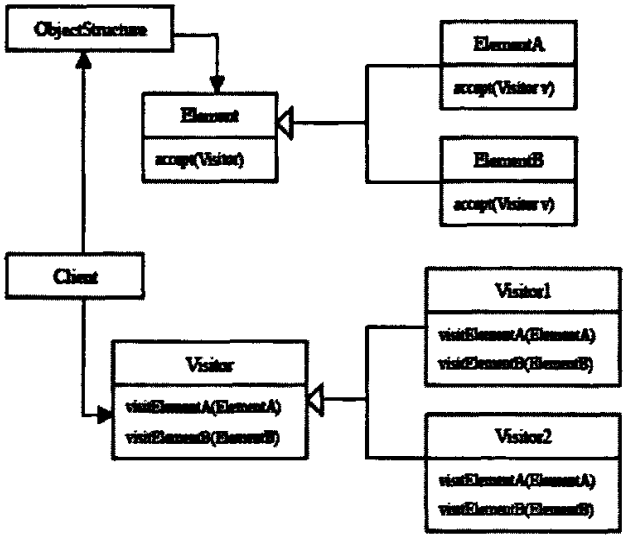


图 4-1 Visitor 模式结构图

模式中定义了两个接口类（或抽象类）Element 和 Visitor。Element 是集

合中所有元素的抽象，Visitor 则是施加在集合元素上操作的抽象。ElementA 和 ElementB 是集合中具体的元素类型，Visitor1 和 Visitor2 代表了具体的两组不同的操作。Visitor 设计模式的关键点在于：借助于面向对象中的多态技术，在 Element 类的 accept 方法中通过运用两次动态绑定实现函数调用的双向分配（Double-dispath）。从而把传统的显式的对象类型判断转化为隐式的动态绑定来实现，大大简化了实现的复杂度。

以下是 Visitor 模式实现的关键代码片断：

```
class ElementA implements Element{ public void accept(Visitor visitor)
{
   visitor.visitElementA(this); //double dispath, 发生两次动态绑定
}
class Visitor1 implements Visitor{
public void visitElementA(ElementA elementA){
// Visitor1 对 ElementA 操作内容 }
public void visitElementB(ElementB elementB){
// Visitor1 对 ElementB 操作内容 }
} }
```

如前所述，Visitor 模式的突出优点是可以在不改变集合中对象元素的前提下，增加对集合中对象元素施加的操作。例如，增加一个 Visitor3，并在其中实现 Visitor 中的两个方法，就可对集合中元素 ElementA 和 ElementB 添加一组操作。

4.1.2 Visitor 模式缺陷分析

为了从本质上分析 Visitor 模式中缺陷产生的原因，在此引入面向对象设计理论中的 3 条基本原则：

（1）OCP（Open Closed Principle）：不应通过修改现有的类来扩展模块功能，而是通过添加新的类来实现，这是面向对象设计中最重要的原则，也是面向对象系统所追求的目标。

（2）DIP（Dependency Inversion Principle）：依赖于抽象，而不依赖于具体。换句话说，要依赖于接口或抽象类，而不是依赖于具体类。DIP 是实现 OCP 的主要机制，当前流行的组件模型，如 COM、EJB、CORBA 等，其设计思想都体现了这条原则。

（3）ISP（Interface Segregation Principle）：分离不同用途的接口。将针对不同用途的接口函数放入不同的接口类或抽象类中，有利于降低各个类之间的耦合度，增强设计的灵活性。

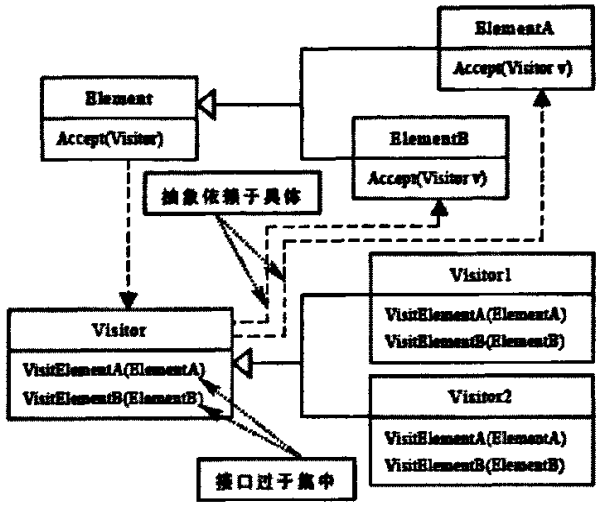


图 4-2 经典 Visitor 设计模式缺陷分析

从图 4-2 中可以看出：

（1）Visitor 接口类直接依赖于 ElementA 和 ElementB 两个具体类，违反了 DIP。

（2）将 visitElementA 和 visitElementB 两个不同用途的接口函数放入一个接口类 Visitor 当中，违反了 ISP。

这使得 Visitor 模式存在两个明显缺陷。

（1）集合中的对象元素的类型必须保持稳定，如果新增一种对象元素，势必引起大量的修改和重新编译。例如，如果在集合中增加一个元素 ElementC，那么必须首先修改 Visitor 接口类，添加一个方法 VisitElementC（），并相应修改 Visitor1 和 Visitor2。

（2）具体的 Visitor 子类必须实现访问所有元素的接口函数。例如，即使 Visitor1 并不需要对 ElementB 进行操作，它也必须实现 visitElementB 接口函数。

Visitor 模式的这两个缺陷使设计方案的可扩展性、可维护性大大降低。当需求改变时，不得不大量修改，增加工作量以及出错几率。

4.1.3 Visitor 模式的改进

现在许多高级程序设计语言（比如 Java 和 C#）都支持反射。反射机制基于这样一种假设：所有的对象都应该是自说明的，一个类实例应该包含了关于它自身的所有信息，包括方法、属性和类型等。在程序运行期间，可以通过反射机制得到对象的类型、方法和属性信息。而反射（Reflection）是.NET 中的重要机制，通过反射，可以在运行时获得.NET 中每一个类型（包括类、结构、委托、接口和枚举等）的成员，包括方法、属性、事件，以及构造函数等。还可以获得

每个成员的名称、限定符和参数等。有了反射，即可对每一个类型了如指掌。如果获得了构造函数的信息，即可直接创建对象，即使这个对象的类型在编译时还不知道。

基于这一点，本文利用反射机制对 Visitor 模式结构进行了改进，使得 Visitor 模式更加简单灵活，并将这种模式命名为 Reflect Visitor 模式。Reflect Visitor 模式的类关系如图 4-3 所示。

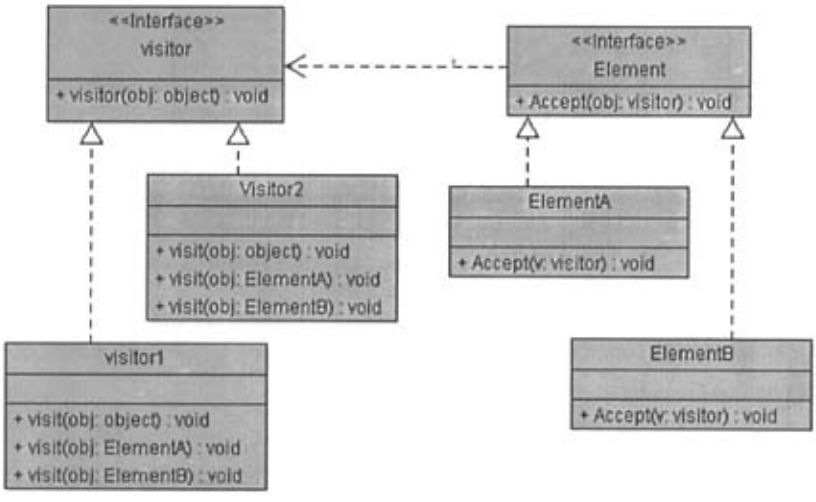


图 4-3 Reflect Visitor 设计模式结构图

按照 DIP 的思想，本文将 Object 类作为 Visitor 类的 Visitor 方法所接受的参数对象的类型，避免了 Visitor 接口类直接依赖于 ElementA 和 ElementB 两个具体类。值得注意的是：Object 类是所有类的基类，从表面上看，这个类型的参数对象已经丧失了其类型信息。不过，通过在程序中恰当地运用反射机制，仍然可以得到参数对象的类型，具体方法可参见下面的程序代码。

按照 ISP 的思想，本文分离 VisitElementA 和 VisitElementB 两个不同用途的接口函数，将其分别放入相应的 Visitor 子类中。如图 4-3 所示，在 Visitor 的子类中重载了 3 种不同参数类型的 visit 方法。程序运行中调用 visit(Object) 方法时，在方法内部通过反射机制决定调用 visit(ElementA) 或是 visit(ElementB) 方法。

以下是 Reflect Visitor 模式实现的关键代码片断：

```
public class ElementA : element
{
    public void accept(visitor v)
    {
```

```
            v.visit(this);
        }
    }
public class Visitor1 : visitor
    {
        public void visit(object obj)
        {

    try{
            MethodInfo mi=this.GetType().GetMethod ("visit",new Type[]
{obj.GetType ()}); //通过反射机制运行时取得对象的类型和方法的信息，并调用对应的
方法。
            mi.Invoke (this,new object []{obj });
        }
        catch(Exception e)
        {
            //处理异常
        }
        }
       public void visit(ElementA obj)
        {
            //对ElementA对象的处理
        }
        public void visit(ElementB obj)
        {
            //对ElementB对象的处理
        }
```

4.1.4 效果分析

Reflect Visitor 模式可以有效消除 Visitor 模式的两个缺陷。(1) 集合中的对象元素的类型可以是变化的，当新增一种对象元素，不会引起大量的修改和重新编译,Visitor 接口不发生改变。例如，如果在集合中增加一个元素 ElementC,只是对 ElementC 感兴趣的 Visitor 子类添加一个 visit(ElementC obj) 方法即可,其它类都不需要变动。(2) 具体的 Visitor 子类可以有选择的访问具体的元素,而不是像 Visitor 模式那样必须实现访问所有具体元素的接口。例如，当 Visitor1 仅需要 ElementA 操作时，它只要添加一个操作 ElementA 的方法 visit(ElementA obj)，而对于不需要的操作可以不作处理。

Reflect Visitor 模式也有其缺点：它的使用范围是有限制的，它只能在支持反射机制的面向对象语言中应用。但是，反射机制其实是面向对象编程语言

的必要组成部分。尽管现在有一些语言还不支持这种机制，但是内建反射机制是面向对象编程语言发展的趋势。有理由相信，这不会成为 Reflect Visitor 模式应用的障碍。

4.2 Observer 模式研究

4.2.1 Observer 模式介绍

第三章已经对 Observer 模式作了介绍，为了便于分析、比较，这里给出了经典 Observer 模式的结构图以及用 C#描述的经典 Observer 模式的一个简单实例。

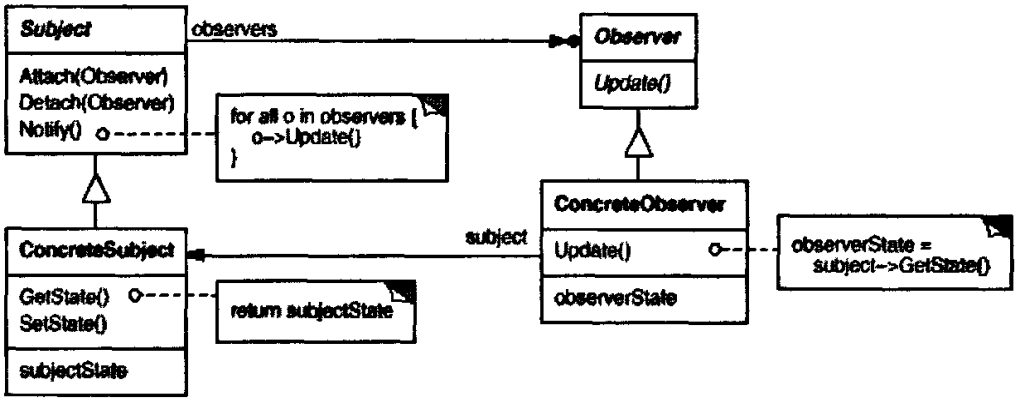


图 4-4 observer 模式结构图

根据《设计模式——可复用面向对象软件的基础》的描述，Observer 模式包括 Subject(目标)——注册和删除观察者对象的接口，Observer（观察者）——在目标发生变化时，需要通知的对象的接口，ConcreteSubject（具体目标），ConcreteObserver(具体观察者)。在具体应用 Observer 模式时，如果仅有一个被观察的目标，那么可以省略 Subject 接口。在一个温度监控系统中，被观察的目标 Subject 负责监视室内温度，当目标状态（即温度）发生改变时，通知观察者（observer1 和 observer2）温度已经变化，观察者得到通知后更新显示。

实现的部分代码：

```
class Subject
{
    //当温度发生改变时，目标对象要通知所有的观察者，所以需要一个容器保存对所有观
    察者的引用
    private ArrayList list = new ArrayList();
    private double temp;
    public double GetState()
    {
```

```
        return temp;
    }
    public double SetState(double tmp)
    {
        this.temp=tmp;
    }
    public void Attach(Observer o)
    {
        list.Add(o);
        o.ObservedSubject = this;
    }
    public void Detach(Observer o)
    {
    }
    public void Notify()
    {
        //在数据发生改变后，遍历列表通知观察者
        foreach (Observer o in list)
        {
            o.Update();
        }
    }
}

Observer(抽象观察者)：
//为那些在目标发生改变时，需要给获得通知的对象定义一个更新接口。
abstract class Observer
{
//内置一个需要观察的对象
  protected Subject s;
}
  abstract public void Update();
}
//ConcreteObserver(具体观察者，在这里就相当于表示层)：
//维护一个指向 ConcreteSubject 的引用。
//储存有关状态，这些状态应与目标的状态保持一致。
//实现 Observer 的更新接口以使自身状态与目标状态保持一致。
class Observer1 : Observer
{
    private string observerName;
    private double curTemp;
    public ConcreteObserver(string name)
    {
        observerName = name;
```

```
    }
    override public void Update()
    {

        curTemp=(s.GetState()*9/5)+32;//转为华氏温度
        //将数据显示出来
        Console.WriteLine("In Observer {0}: data from subject = {1}",
        observerName, s.GetState());
    }
}
```

4.2.2 Observer 模式缺陷分析

Observer 模式的优点是抽象了更新接口，使得目标和观察者之间仅存在抽象耦合（目标对象只所知道它有一系列的观察者，而每个观察者都符合抽象Observer类的简单接口），实现了表示层和数据逻辑层的分离；支持更新的广播通知（通知被自动广播给所有已经向目标对象注册的观察者对象）。但是Observer模式也存在如下缺点。

（1）Observer模式的每个观察者对象必须继承这个抽象出来的接口类，具体的Observer类受到了接口实现的限制。这样就使得系统的灵活性降低，比如有一个构件类，它负责在室内温度发生改变时决定是否打开制冷设备。这个构件独立于上面提到的温度监控系统，它并没有继承自 Observer 抽象类或者Observer 接口。它仅提供了一个用于在温度发生改变之后控制制冷设备的public 接口。我们希望不修改该类直接使用它，甚至可能由于是购买的构件我们无法修改这个类。这给我们提出了一个难题，虽然可以再应用Adapter模式，在一定程度上解决这个问题，但是会造成更加复杂烦琐的设计，增加出错几率。

（2）在Observer模式中，目标对象通过在一个容器对象内显式的保存，对所有观察者的引用来跟踪它应该通知的观察者。然而，当目标很多而观察者较少时，这样存储的代价太高。

（3）在Observer模式中，必须小心翼翼地处理一个问题——避免对已删除对象的悬挂引用，在删除一个目标时要注意不要在其观察者中遗留对该目标对象的悬挂引用。为了避免悬挂引用，通常的做法是当一个目标被删除时，让它通知它的观察者对象将对该目标的引用复位。这样做需要为Observer增加一个接口，加大了系统的开销，增加模式了实现的难度。使用者需要更多地关注细节，而不是设计本身，这违背使用模式的本意。

为了减少Observer模式应用的复杂性，降低Observer对象与Subject对象之间的耦合性，使系统更加灵活，我们采用C#的委托特性(Delegate)对Observer模式进行了优化。

4.2.3 Observer 模式的改进

委托技术是.NET 引入的一种重要技术，使用委托可以实现对象行为的动态绑定，从而提高设计的灵活性。

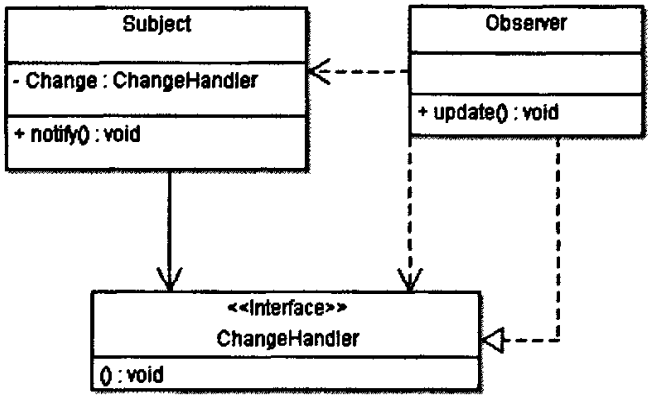
.NET 运行库支持称为“委托”的引用类型，其作用类似于 C++中的函数指针。与函数指针不同，委托实例独立于其封装方法的类，主要是那些方法与委托类型兼容。另外，函数指针只能引用静态函数，而委托可以引用静态和实例方法。委托主要用于.NET Framework 中的事件处理程序和回调函数。所有委托都从 System.Delegate 继承而来并且有一个调用列表，这是在调用委托时所执行方法的一个链接列表，可以使用 Cimbine 及 Remove 方法在其调用列表中添加和移除方法。若要调用委托，可使用 Invoke 方法，或者使用 BeginInvoke 和 EndInvoke 方法异步调用委托。委托类的实现由运行库提供，而不由用户代码提供。

委托适用于那种在某些语言中需要用函数指针来解决的情况，但是与函数指针不同，它是面向对象和类型安全的。

委托声明定义一个类，它是从 System.Delegate 类派生的类。委托实例封装了一个调用列表，其中列出了一个或多个方法，每个方法称为一个可调用实体。对于实例方法，可调用实体由一个实例和该实例的方法组成；对于静态方法，可调用实体仅由一个方法组成。如果用一组合适的参数来调用一个委托实例，则该委托实例所封装的每个可调用实体都会被调用，并且使用上述同一组参数。

委托实例的一个有用的属性是它既不知道，也不关心其封装方法所属类的详细信息，对它来说最重要的是这些方法与该委托的类型兼容。即只要方法的返回类型和参数表是相同的，则方法与委托类型兼容，方法的名称不一定要与委托类相同。

本文利用委托对 Observer 模式进行了优化，改进后的结构如图 4-5 所示



4-5 基于委托的 Observer 模式结构图

下面是改进后的Observer模式实现

```
//先定义一个Delegate:
delegate void UpdateDelegate(double tmp);
/
//Subject(目标):
class Subject
{
    private double temp;
    //定义一个事件，代替前面的观察者对象列表
    public event UpdateDelegate UpdateHandle;
    //没有变化的地方不列出了
    public void Notify()
    {
    //通知更新
    if(UpdateHandle != null) UpdateHandle(tmp);
    }
}

//Observer(抽象观察者):
//无，因为不需要抽象接口类了，所以可以省去抽象观察者类。
//ConcreteObserver(实体观察者):
//为了能更加清楚的说明问题，这里定义了两个独立的观察者，
 class Observer1
{
    private double tmep;
    public Observer1(string name)
    {
        observerName = name;
    }
//只需要与委托声明的参数以及返回值类型一致就可以了
    public void Update1(double tmp)
    {
        temp=32+tmp*9/5;
        //显示
    }
}
class Observer2
{
    ...
    private double tmep;
    public void Update2(double tmp)
    {
        this.temp=tmp;
        //显示
```

```
    }
}
```

4.2.4 效果分析

利用.net的新特性delegate类型，我们提出一种与经典Observer模式不同的实现方法，同样可以满足Observer模式的意图，并具有前面提到的Observer模式的优点。和经典Observer模式相比较，基于委托的Observer模式具有以下优点。

（1）去掉了Observer类抽象接口，并且通过将观察者对象的更新方法委托给Subject类来实现目标对象更新事件的自动通知——当目标对象改变时，即可通过委托通知到各个观察者。这样一来系统在结构上更加紧凑，并降低了系统的复杂性，提高了系统的灵活性。

（2）利用委托机制实现更新事件的自动通知，目标对象不需要保存对所有观察者对象的引用，同样观察者也无需保存对目标对象的引用，甚至观察者可以完全不知道目标对象的存在。这样，维持目标对象和观察者对象之间联系的开销降到了最小。

（3）在Observer模式中，目标对象使用一个容器类对象保存对所有观察者对象的引用，Subject类提供了Attach和Detach两个方法负责添加和删除观察者对象的引用。使用者需要自己负责处理对观察者引用的管理，增加了系统实现的复杂度，容易出错。而且，在删除目标对象或观察者对象时如果处理不当，会产生悬挂引用，引发系统错误。而采用委托机制，观察者对象只是简单的将目标对象发生改变后的处理方法委托给目标对象就可以。而目标对象对于委托的管理是由.NET的委托链机制实现，使用者不需要过多地关注实现的细节。在上面的例子中，用UpdateHandle+=new UpdateDelegate (Observer1.Update1)，就可以实现在UpdateHandle中添加委托，而用“-=”可以方便的实现删除委托。另外，由于仅仅是将更新处理方法委托给目标对象，所以删除目标对象或者观察者对象不会引发删除后产生悬挂引用的问题。

4.3 Strategy模式研究

4.3.1 Strategy模式概述

Strategy模式也是对象行为模式的一种，意图是定义一系列的算法，把它们封装起来，并使他们可以相互替换，使得算法可以独立于使用他们的客户而存在。Strategy模式的结构如图4-6所示。

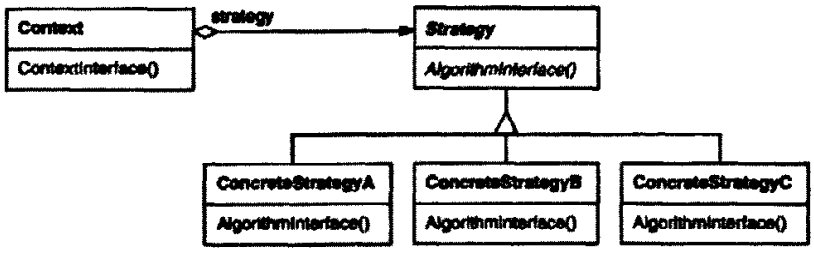


图 4-6 Strategy 模式结构图

其中 Strategy(策略)定义所支持算法的公共接口，ConcreteStrategyA, ConcreteStrategyB, ConcreteStrategyC,（具体策略）以 Strategy 接口实现具体的算，Context（上下文）用一个 ConcreteStrategy 对象来配置。Strategy 模式一般用于以下情况：

（1）许多相关的类仅仅是行为有异，Strategy 模式提供了一种用多个行为中的一个行为来配置一个类的方法。

（2）需要使用一个算法的不同变体。

（3）算法使用了客户不应该知道的数据，可以使用 Strategy 模式以免暴露复杂的，与算法相关的数据结构。

（4）一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现，可以将相关的条件分支移入它们各自的 Strategy 类中以替代这些条件语句。

下面以税务系统的一个业务行为举例，介绍 Strategy 模式的使用。假设我们要开发一个税务系统，那么有关税务的计算就会依照纳税人的不同而分为个人所得税和企业所得税，而这两种税收类型依法应缴纳的税金在计算方式上是迥然不同的。此时，我们就可以应用 Strategy 模式，将税收策略抽象为接口。

ITaxStrategy：

```
public interface ITaxStrategy
{
    double Calculate(double income);
}
```

在对税收计算策略进行抽象以后，就从设计上去除了模块间存在的耦合，消除了因变化而可能造成未来系统的大规模修改，所谓“面向接口编程”[29]正是基于这样的道理。

定义接口之后，各种税收策略均实现该接口：

```
public class PeronalTaxStrategy:ITaxStrategy
{
```

```
public double Calculate(double income)
    {
     //实现略;
    }
}
public class EnterpriseTaxStrategy:ITaxStrategy
{
public double Calculate(double income)
    {
     //实现略;
    }
}
```

如果此时有一个公共的类，提供税收的相关操作，其中就包括计算所得税的方法：

```
public class TaxOp
{
private ITaxStrategy m_strategy;
    public TaxOp(ITaxStrategy strategy)
    {
     this.m_strategy = strategy;
    }
    public double GetTax(double income)
{
     return strategy.Calculate(income);
    }
}
```

在这个类中，接收了一个 ITaxStrategy 类型的对象，由于该对象是一个接口类型，因此类 TaxOp 是与具体税收策略无关的，它们之间因为接口的引入而成为了一个弱依赖的关系，如图 4-7 所示。

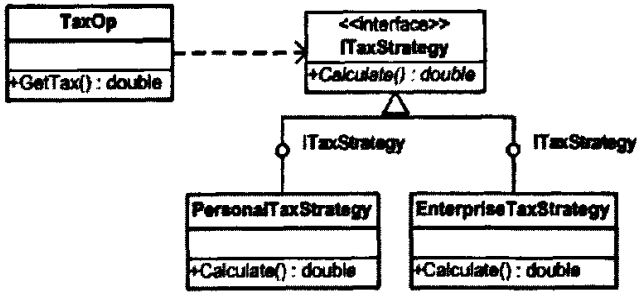


图 4-7 税收 Strategy 模式结构图

如果客户端要调用有关税收的操作时，就可以根据纳税人的类型具体实例化税收策略对象：

```
public class App
{
    public static void Main(string[] args)
    {
        TaxOp op = new TaxOp(new PersonalTaxStrategy());
        Console.WriteLine("The Personal Tax is :{0}", op.GetTax(1000));
    }
}
```

4.3.2 Strategy 模式缺陷分析

Strategy 模式存在一个缺点,即客户要从多个算法选择一个合适的算法,就要求客户了解所有的算法之间存在的差异。此时,就不得不向客户暴露不同算法实现的具体细节,这违背了面向对象封装性的设计原则。对图 4-7 税收 Strategy 模式结构图分析可知,TaxOp 类有一个 ItaxStrategy 接口成员指向一个具体的 Strategy,它在类生成的同时被创建,可以在运行被动态替换。为了解决这个问题,我们可以把具体的算法看成一个产品,把选取具体的算法当作从一类产品中选择要生产的产品。这正是 Abstract Factory 模式要做的。通过一个抽象工厂封装创建 Strategy 对象的职责和过程,从而将客户与 Strategy 实现分离。通过引入 Abstract Factory 模式对 Strategy 模式进行改进,对客户与具体的 Strategy 解藕,可以有效地解决 Strategy 模式的这个缺陷。本文将这种改进后的 Strategy 模式称为基于抽象工厂的策略模式。

4.3.3 Strategy 模式改进

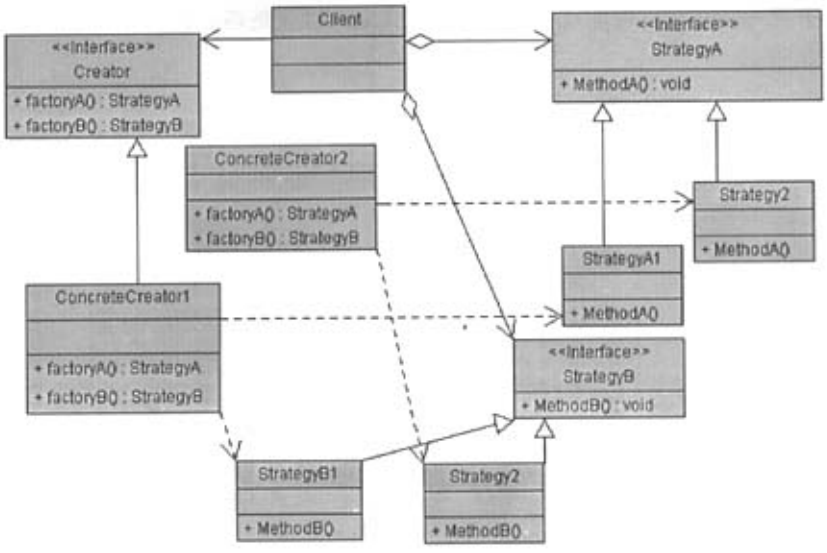


图 4-8 基于抽象工厂的策略模式

图 4-8 所示是基于抽象工厂的策略模式的结构图，通过在 Strategy 模式中引入 Abstract Factory 模式，可以解决 Strategy 模式需要向客户暴露算法实现细节的不足。除此之外，基于抽象工厂的策略模式还可以解决如下问题：从多组系列算法中选择一组算法，这组算法之间存在相互联系，并且希望在运行时能够被方便的替换。

根据结构图，我们可以看到，通过用一个具体的工厂 ConcreteCreator 来实例化 Creator，由 ConcreteCreator 配置多个系列算法中的一组。基于抽象工厂的策略模式，使得算法系列易于交换。一个具体的工厂在一个应用中仅出现一次，即在它初始化的时候，这使得改变一个应用工厂变得很容易——只要改变具体的工厂，就可以使用不同的算法配置；另外，当一组算法具有一定的相关性时，抽象工厂保证一个应用只能使用同一组相关的算法，从而保证了系统的一致性。下面以错误管理机制为例介绍基于抽象工厂的策略模式的应用。

当一个项目被分给多个开发人员时，经常会出现不同的错误存取和表示机制。比如说在存取错误信息时，可以采用数据库、文本文件、xml 文件等；在显示错误时，可以使用基于 Windows 消息框的机制，或使用基于控制台文本输出的显示机制，或基于文本控件的机制等，使不同的开发人员使用不同的消息存取和显示机制的组合。为了能够将错误显示机制集成在一起，我们可以使用基于抽象工厂的策略模式。对于错误的存取和显示各自有不同的策略算法，分别提取出一个接口，ErrorStore 和 ErrDiplay。对于每个需要捕捉错误的 Client，都保存

着对这两个接口的引用，而具体的错误存取和显示由 Client 实例化抽象工厂生成，并且可以在运行时通过传入新的工厂来动态改变策略。它的类结构如图 4-9 所示。

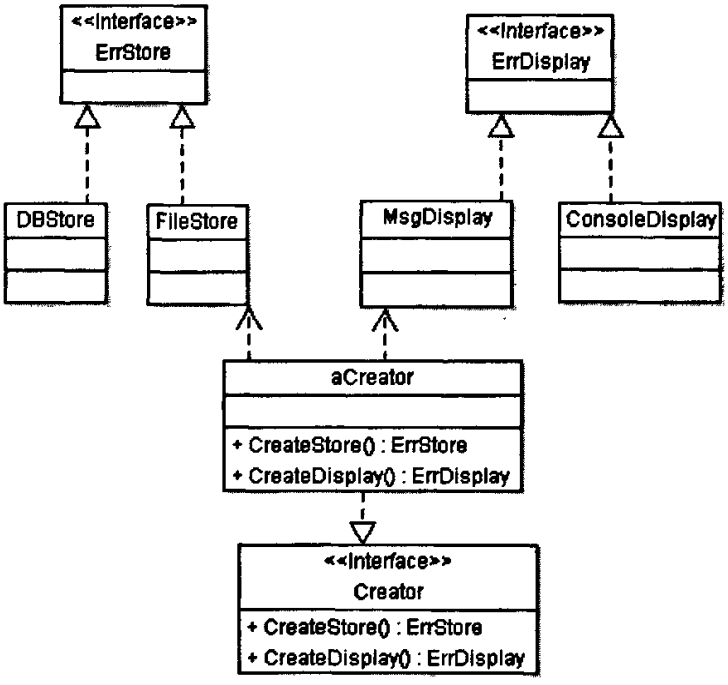


图 4-9 错误日志的抽象工厂多策略模式结构图

从图 4-9 中可以看出，选择具体的策略交由策略工厂负责。首先，我们定义了一个抽象工厂接口（Creator），它有两个方法，CreateStore 方法负责生成 ErrStore 策略产品，CreateDisplay 方法负责生成 ErrDisplay 策略产品。这里只画出了一个具体工厂(aCreator)，它生成的 ErrStore 策略产品是 FileStore 类的一个实例，ErrorDisplay 策略产品是 MsgDisplay 的实例。部分实现的代码如下：

```
interface ErrorStore
{
    void storeError(string strError);
    string getError();
}
interface ErrorDisplay
{
   void Display(string strError);
}
interface Creator
{
```

```
    ErrorStore  CreateStore();
    ErrorDisplay CreateDisplay();
}
class FileStore : ErrorStore
{
    public void storeError(string strError)
    {
    }
    public string getError()
    {
        return "";
    }
}
class DBStore : ErrorStore
{
    public void storeError(string strError)
    {
    }
    public string getError()
    {
        return "";
    }
}
class MsgDisplay : ErrorDisplay
{
    public void Display(string strError)
    {

    }
}
class ConsoleDisplay : ErrorDisplay
{
    public void Display(string strError)
    {

    }
}
class aCreator : Creator
{
    public ErrorDisplay CreateDisplay()
    {
        return new MsgDisplay();
    }
    public ErrorStore CreateStore()
```

```
        {
            return new FileStore();
        }
    }
```

Client 代码：

```
    public partial class Form1 : Form
    {
        ErrorDisplay ed;
        ErrorStore es;
        Creator ec;
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            ec = new aCreator();
            ed = ec.CreateDisplay();
            es = ec.CreateStore();

        }
        private void storeErr(string strError)
        {
            es.storeError(strError);
        }
        private string getError()
        {
            return es.getError();
        }
        private void DisplayError(string strError)
        {
            ed.Display(strError);
        }
```

4.3.4 效果分析

与 Strategy 模式相比，基于抽象工厂的策略模式具有以下优点：

（1）避免向客户暴露算法的细节，确保对象的封装性原则。

（2）易于实现系列算法的运行时替换。负责生成具体算法的工厂仅在使用算法的 Context 被创建时或替换算法时才出现。这使得替换算法系列变得极为容

易——只需要改变一个具体的工厂就可以使用不同的系列算法了。

（3）能保证相关算法的一致性。通过继承自抽象工厂接口的具体工厂子类统一提供的创建Strategy对象的方法来确保在Context中使用的算法的一致性。

但由于 Abstract Factory 接口限制了可以被创建的“算法产品”的集合，我们要想支持新的“算法产品”，就需要修改工厂接口，这将改变 Abstract Factory 类及其所有子类。所以基本的 Abstract Factory 模式仅适合那些“算法产品”比较固定的应用。如果想要方便的实现增加新的“算法产品”，可以考虑采用基于类自说明的反射机制来改进 Abstract Factory 模式。通过反射机制，我们可以根据类名来取得该类的构造函数，并调用其构造函数来生成该类的实例。这样我们就可以实现一个可扩展的抽象工厂，以满足添加新的“算法产品”的需求。

4.4 本章总结

描述一个设计模式时需要提到该模式的效果，即模式的优点和缺点。缺点就说明模式存在的不足，这个不足有可能是由于模式自身结构的不足，也可能是因为实现手段局限造成的不足。本章通过描述 Visitor 模式的改进，以及 Observer 模式新的、更灵活的实现手段，指出了改进设计模式的方向和方法。如采用反射机制，可以弥补 Abstract Factory 和 Factory Method 等模式的不足；用 delegate 机制灵活实现 Mediator 模式；利用 C#、Java 等新型的面向对象程序语言，对现存设计模式的设计或结构上的不足进行改进，为其提供更灵活的实现手段。C# 已经推出 3.0 使用版，相比 1.0 和 2.0 版，提供了许多新的特性，也更丰富和强大了类库。下一步，将关注 3.0 版的新特性和支持库对传统设计模式的改进，以及在该平台下提出新的设计模式。另外，在总结以往设计经验基础上，整合经典设计模式，对 Strategy 模式进行进一步的改进，用于解决需要从多组系列算法中选择一组算法，这组算法之间存在的相互联系和可能被方便替换的问题。

第5章 设计模式在软件设计中的应用

设计模式关注的是特定设计问题及其解决方案，它描述了如何利用面向对象的基本概念和机制来解决软件设计中经常出现的问题，并针对设计问题给出可复用的解决方案。这些解决方案是通过对反复出现的设计结构进行识别和抽象而得到的。同时，每个模式都伴有定义的语境和强度，语境解释了模式的适用情况，强度是语境中的元素。如果问题的环境与模式的语境和强度相匹配，该模式便可应用。因此，设计模式是软件人员在面向对象程序设计过程中对成功解的记录与提炼。

5.1 设计模式的选取

5.1.1 设计模式的应用规律

各种模式均有其自身特点和适用范围，所以在选取模式前应充分了解模式的内涵及侧重点，发掘出不同模式的适用规律。常用模式的应用规律有如下几种。

(1) Flyweight 模式：该模式可用于实现共享细粒度符号对象，主要解决由于系统存在大量类似的、具有共性的对象而严重影响系统性能的问题。 此时，可将对象的共同信息提取出来并作为一个新的 Flyweight 对象，而原有对象需要的且重复的信息描述只需要在一个共享的 Flyweight 对象中描述，从而大大削减了应用程序创建的对象，降低了程序内存的占用率，增强了程序的性能。如果一个应用程序需要显示的对象同属于一种类型，就可以考虑用 Flyweight 模式来共享一定数量的对象。例如，在 Java 程序中，字符串、Swing 树节点、组件边界等都利用了 Flyweight 。如果从一个数据库中读取一系列字符串，这些字符串中有许多是重复的，那么可将这些字符串储存在 Flyweight 池(pool)中。又如字处理软件，若以单字作为一个对象的话，要是有数千个字则需数千个对象，这在处理字的同时无疑要耗费很大的内存资源，所以需要找出这些对象群的共同点，设计一个元类，并将共享的类封装起来。

(2) Proxy 模式：该模式主要是通过代理来控制对象的访问，所覆盖的应用场合从小结构到涉及整个系统的大结构，并具有以下功能。(a) 防止越权访问功能，主要是对不同级别的用户进行权限划分和管理控制，如论坛、银行信息、图书档案管理等系统的管理控制。(b) 存取优化功能，如 word 文档中有很大的图片，要打开该文档通常需要花费很长时间，这时需要做图片 Proxy 来代替真正的图片，以便提高存取效率。(c) 客户端存取远程服务器信息的功能，如果直接操作 Internet 远端服务器上的对象，当因网络运行缓慢而影响访问速度，则可应用

Proxy。

(3) Observer 模式：该模式的特点是在对象间定义一对多的依赖关系，当一个对象的状态发生改变时，所有依赖它的对象都将被告知并自动更新，主要应用有：(a) 在界面设计中的应用。因为应用程序的开发往往要求用户界面与业务逻辑分离，而 Observer 模式就是解决此类问题的最常用的设计模式。(b) 解决对象间的状态同步问题。当一个用户视图中的数据被其他用户改动后，后端的数据库能够自动更新，而当数据库以其他方式更新后，用户视图中的数据显示也会随之改变。(c) Observe 模式还适用于实时更新的系统，如股票系统、税务系统、网上商店等。

(4) Composite 模式：该模式是将对象组织成“整体--部分”的层次结构，即将对象组织成树状结构，从而保证客户机在使用单个对象和复合对象上的一致性。这样做，客户端不但能一致地使用组合结构或单个对象，简化了客户端的调用，也可以在组合体内部加入新的对象，而用户则无需更改代码。Composite 模式主要应用于树形结构系统，另外还常用于一些涉及产品结构和配置管理的系统，此类系统可将产品的结构和配置看作是一种产品结构树，如产品由零件、部件组成，部件又由零件、部件组成。这种嵌套递归关系通常使用 Composite 模式来表示。Composite 模式也适用于界面设计，例如图形由若干基本对象组成，而这些对象的操作基本相同，亦即移动、缩放、绘制、拷贝、粘贴等。使用 Composite 模式可以规范设计，从而提高代码的可重用性。

(5)其他模式也有不同的应用，不再一一列举了。

5.1.2 设计模式的选取步骤

设计模式一般应用于以下 2 个方面[23]：①在软件系统设计的开始阶段就应用设计模式对软件体系结构进行设计；②在系统的体系结构设计初步完成后，通过对系统另有要求的组件或模块应用设计模式进行重构使其更加优化、灵活。

由于设计模式具有一定的复杂性，所以很难将其应用到具体的软件设计中，主要原因有两点：①软件设计人员没有正确把握和理解软件设计模式;；②没有一种有效的方法来指导使用这些设计模式。为此，本文在总结了软件设计模式的应用经验的基础上，提出了设计模式应用于软件设计的策略，其操作步骤如下：

步骤 1：对所要解决的问题进行抽象，并划分适当的类型。

步骤 2：根据问题类型选择适合的设计模式。

步骤 3：规划问题和匹配模式，即将所要解决的问题与所选择的设计模式进行比较，找出共性。在所要解决的问题域内考虑元素对应于模式中的类和模式中的各种角色，如果发现选择的设计模式并不合适，返回步骤 3，重新进行设计。

步骤 4：对选取的模式进行变体，即对模式的原始结构进行修改或扩展，以

解决具体问题。

5.2 图片资料管理系统的设计模式选取

随着扫描仪、数码相机，数码摄像机等品类繁多而操作简易的数码产品的涌现，以及多媒体的计算机的快速发展，人们把相片或文字、图片资料以数字图像的形式存储在电脑里。随着时间的推移，数量巨大而且分散的图像资料使得在浏览和整理相片时十分困难，经常会出现为了找一张图片，而去察看完整个电脑硬盘中图片的情况。因此，应用设计模式开发出一个图像资料管理系统，能够方便地对数字图像资料进行有效的管理。

经需求的调研和分析，一个图像资料管理系统应具备以下基本功能。

（1）图片信息的分类管理。

（2）图片信息的录入与相关信息的维护，如将一幅图片录入到系统时，需要同时保存图片资料的拍摄时间、内容和一些相关信息。

（3）图片信息的输出和索引。

（4）图片信息的安全。如有些图片资料涉及机密或个人隐私等，需要提供安全保密的功能。

经过分析，本系统使用C#作为开发语言。C#语言兼有C++和Java两家之长，并且微软的.net studio工具也为其提供了一个良好的集成开发环境，使它具有类似Visual Basic快速开发能力。C#与Java一样基于虚拟机，具有平台无关性和可移植性。C#的反射机制和delegate等新特性使得设计者可以设计出更灵活的软件构架。第三章和第四章的内容也说明了C#语言在应用设计模式，改善设计质量方面的能力。

通过对本系统需要应对的变化分析，以及为了在设计中获得更大的灵活性，提高代码的可靠性、可复用性。在本系统中应用了Abstract Factory模式、Proxy模式、基于抽象工厂的策略模式、Composite模式和Iterator模式等进行模式的设计。

5.2.1 Abstract Factory模式

为了解决系统的持久化存储因应用的环境不同而可能发生变化的问题，本系统选用抽象工厂模式，使得当更换一种存储介质时只要添加几个子类，而不需要大量修改已有代码，就可以实现，降低了更新的难度和出错的可能性，使系统具有更大的灵活性。按照设计模式的选取策略，将详细分析此设计模式的选取过程。

1．分析问题与划分问题类型

图片的存储路径、文件名、拍摄时间、详细描述等信息需要持久化存储。我们一般采用数据库或者XML文件存储，对于较小的应用也可以采用随机文件，由自定义系统控制数据的插入、删除等物理操作。如果使用硬编码，即采用什么存储方式，就在客户端代码中直接使用处理数据操作的对象类。那么一旦系统的规模发生变化，采用的存储手段也就会发生相应变化，这时需要修改涉及数据存取操作的所有模块；即使都是采用数据库存储，不同的数据库在实现的细节上也有区别，数据库的替换也会导致同样的问题。选择一种具体的数据持久化存储方式,实际上就是创建一个具体的持久化对象，显然应该是属于创建型模式。

2．选择适合的创建型模式

创建型模式包括Factory Method模式，Abstract Factory模式，Prototype模式，Singleton模式和Builder 模式。Singleton 模式确保一个类仅有一个实例，因此可排除该模式。Builder 模式的特点是将复杂对象的构建与它的表示分离，这与本系统持久化存储的需求无关，所以也可以排除。 Prototype 模式用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象，一般适用于当一个类只有几个不同状态的组合，每次建立相应数目的原型并且克隆它们来生成新的实例，而不是在创建时用合适的状态去实例化的情况；或者实例化的类需要在运行时指定的情况，这两种情况对本设计而言都不太符合。Factory Method 模式可以提供一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method使得一个类的实例化延迟到其子类，一般用于不希望客户知道应该对哪个类进行实例化的情况。Factory Method 在实例化的过程中使用一个方法，这个方法需要利用外部因素（outside factor）来确定对哪个类进行实例化。当这些外部因素增加或者子类的数目较多时，会使代码臃肿和复杂，难以维护。因此对于本设计也不是特别适合。Abstractor Factory 模式提供一系列相关或相互依赖对象的接口，而无需指定它们具体的类，适用于一个系统要由多个产品中的一个来配置。而本处的需求是要求系统能够在多种持久化存储方式之中选择一种具体的持久化存储方式，而不需要大量的修改代码。如果把具体的持久化存储方式看作一种“产品”，选择一种具体的持久化存储方式相当于由多个产品中的一个来配置，这点与Abstractor Factory模式的意图是相吻合的。因此，Abstractor Factory模式满足我们设计的要求。

3．规划问题和匹配模式

对数字图片需要进行分类归档，因而需要对目录信息进行管理，即目录的添加、删除、更新和检索；同样，对数字图片的管理也有添加、删除和检索的操作。这些操作都是涉及到持久化存储的管理操作。为了防止概念上的混淆，引用UML关于“操作”和“方法”的定义将这两者的区别开来，方便讨论。一个操作是一

个服务的规范，并且可以由某个类的一个实例发出对此服务的请求。一个方法是某个操作的一个实现。一个操作规范了一个类可以完成的某项工作，并且规范了调用这个服务的接口。不同的持久化存储实现这些操作的方法都不相同，为了可以方便的实现在不同的持久化方式之间切换而不会造成程序适应性的困难。我们声明一个用于创建持久化存储类型的接口 Creator，它仅提供一个操作，返回一个具体的持久化“产品”。Creator 接口如图 5-1 所示。

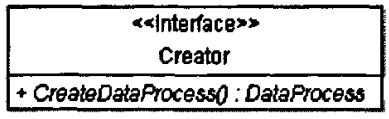


图 5-1 Creator 接口

每一个具体的持久化产品都应该是一个抽象“产品”类的一个实例，都实现了上述操作。这个抽象的“产品”类——定位为 DataProcess，为上述所有的操作提供了一个接口，DataProcess 接口如图 5-2 所示。

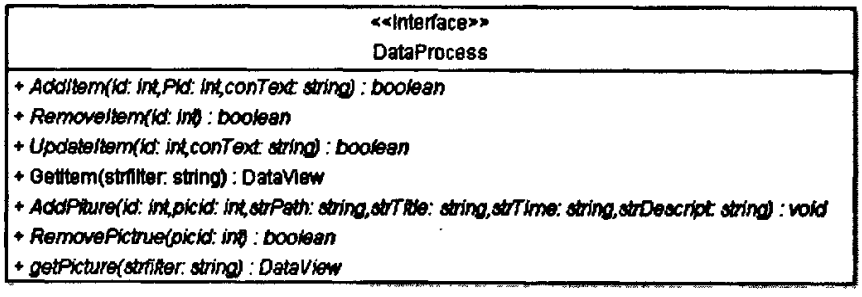


图 5-2 DataProcess 接口

本系统列举了两种具体的持久化方式，一种使用 xml 文件存储，一种使用 sql server 实现数字图片信息的存储管理，那么分别为这两种存储方式定义两个类，继承自 DataProcess 接口，分别命名为 XmlDataProcess 和 SqlDataProcess。通过为每一种持久化方式提供具体的工厂，XmlDataCreator 和 SqlDataCreator。在具体工厂中，返回所选存储方式的实例。其结构图见图 5-3。

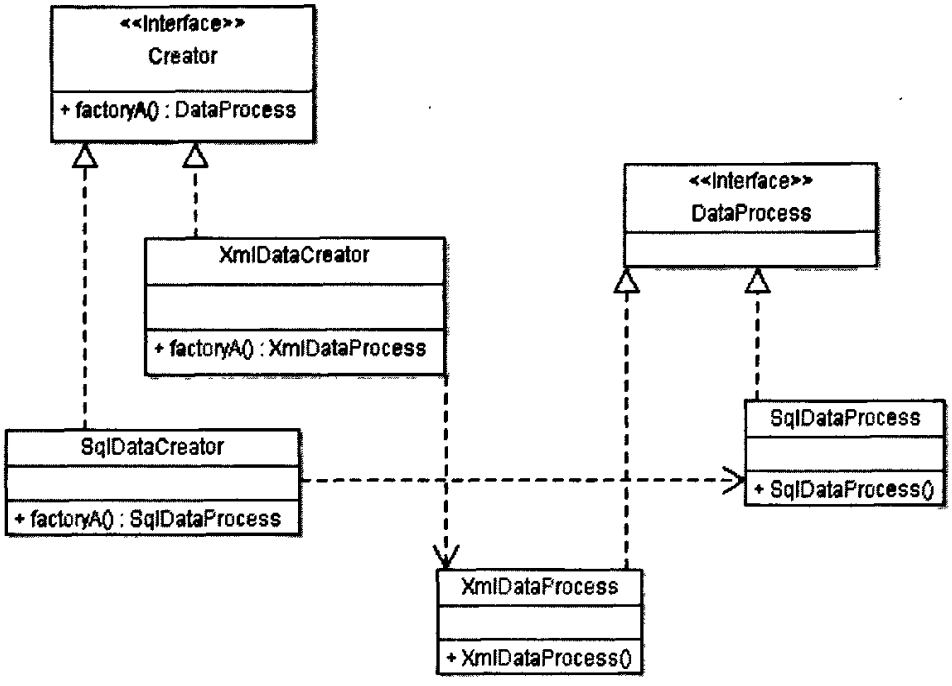


图 5-3 抽象工厂结构图

4. 完善设计

采用抽象工厂模式实现，有两个方面的问题需要考虑：

（1）是否去掉抽象工厂接口，简化设计

当产品总类比较少，并且实例化一个产品的参数类型是一致的，且代码与客户端结合的比较紧密，我们可以去掉抽象工厂接口，仅提供一个具体工厂。在具体工厂中，通过传入的参数（如标记）具体实例化产品。实现的方式如下：

```
class concreteFactory
{
    IProduct createProduct(int flag) //IProduct是产品类接口，flag为标记
    {
        switch (flag)
        { //根据标记返回具体的产品
            case 1:
                return new Product1();
                break;
            case 1:
                return new Product2();
                break;
        //其他的产品
         .... }     } }
```

这样做虽然，减少了具体工厂的数量，但是却将创建实现的细节暴露给了客户，并且要求在此“工厂”为所有可能出现的产品编码，尽管在这个系统中并不会用到该产品类；添加一个新的产品类，要求具有相同的构造函数，并且需要同时修改具体工厂，造成许多的硬伤。对于本系统，我们采用的做法是保留抽象工厂，并且为每个产品定义一个工厂方法，一个具体的工厂方法为每个产品重定义该工厂方法以指定产品。

此模式在本系统的实现的部分代码：

```
interface  Creator
 {
     DataProcess DataProcessFactory();//工厂方法
}
 class XmlDataCrator : Creator //一个具体工厂
 {
    string strXmlFile;
    string strXmlPic;
    public XmlDataCrator(string strFile,string strPic)
    {
        strXmlFile = strFile;
        strXmlPic = strPic;
    }
    public DataProcess DataProcessFactory()
    //重载该工厂方法，返回一个Xml存储方式实例
    {
        return new XmlDataProcess(strXmlFile,strXmlPic );
    }
}
```

（2）在实现抽象工厂模式时，Abstract Factory 通常为每一种它可以生产的产品定义一个操作，产品的类别被硬编码在操作结构中。增加一种新的产品要求改变 Abstract Factory 的接口以及所有与它相关的类。如果要产品类中增加新的产品如何解决？

本系统中原本应该存在两个产品，即目录和数字图片信息。但为了简化设计将两个产品合二为一了。如果项目的需求发生变化，需要存储的信息增加，即需要加入新的“产品”，怎样避免因新增加新的产品而改变抽象工厂接口以及所有的具体工厂子类。解决方案之一是给创建对象的操作增加一个参数，该参数指定被创建的对象的种类。如果是在 C++这类语言中使用此方式要求所有的产品具有相同的抽象基类，而且存在一个本质的问题：那就是所有的产品将返回类型所给定的相同的抽象接口返回给客户。客户将不能区分或对一个产品的类别进行安全的设计[47]。通过第四章的讨论，利用 C#的反射机制我们可以通过一种反射的抽象工厂的设计模式加以解决。解决方案的示范性代码如下：

```
//简单示例
public class AbstractFactory
{
    public Iproduct createProduct (string Name)
    {
     Iproduct MyProduct = null;
      try
      {
          //通过反射由类型名称，获得要创建的类型信息
        Type type = Type.GetType(Name,true);
          //实例化一个产品
       Iproduct = (Iproduct)Activator.CreateInstance(type);
      }
      catch (TypeLoadException e)
       {
              //错误处理
       }
               return MyProduct;
    }
}
```

5.2.2 Proxy 模式

由于一个目录下的图片数目非常多或者像片的容量非常大，一次性加载的时间会比较长。用户在目录之间切换时将耗费大量的时间。当我们需要使用的对象很复杂或者需要很长时间去构造，这时就可以使用 Proxy 模式。例如：如果构建一个对象很耗费时间和计算机资源，Proxy 模式允许我们控制这种情况，直到我们需要使用实际的对象。一个代理(Proxy)通常包含和将要使用的对象同样的方法，一旦开始使用这个对象，这些方法将通过代理(Proxy)传递给实际的对象。一个简单的图像代理的实现方式如图 5-4 所示。

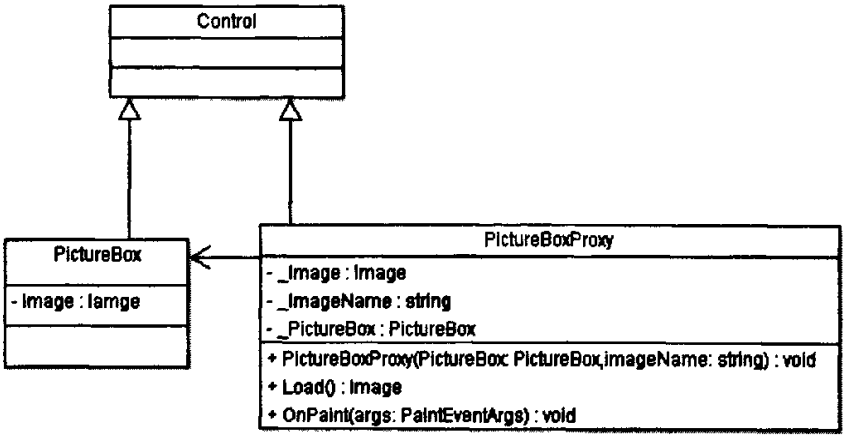


图 5-4 简单图像代理类图

pictureBoxProxy 类的构造函数接收了一个 PictureBox 对象。pictureBoxProxy 类是这个 PictureBox 对象的代理，并接收了一个大型图像的文件名，这个图像将根据请求需要加载到内存中。实现的部分代码如下：

```
class PictureBoxProxy:Control
  {
      private PictureBox _picturebox;
      private Image _image=null;
      private string _imageName;
//构造函数接收被代理的PictureBox对象，imageName是要加载的图像的文件名
      public  PictureBoxProxy(PictureBox picturebox, string imageName)
      {
          _imageName = imageName;
          _picturebox = picturebox;
      }
//当图像未被加载时，显示替代用的小图片，否则显示加载过的实际图片
      public Image Image
      {
          get
          {
              if (_image == null)
              {
                  return _picturebox.Image;
              }
              else
                  return _image;
          }
      }
      public void Load()
```

```
        {
            _image = Image.FromFile(_imageName);
        }
        //OnPaint方法始终显示代理对象的Image属性，如果代理的Load方法已经被调用，
        //将会显示实际图片
        protected override void OnPaint(PaintEventArgs e)
        {
            e.Graphics.DrawImage(Image, 0, 0);
        }
    }
```

但是 PictureBoxProxy 并不是一个良好的设计。首先,对于当前一般处理能力的计算机，都可以在不到 0.1 秒的时间类加载一幅全屏幕的大型图像，那么为每张要显示的图片都这样处理是否有必要？其次，将调用集合的一个子集转发给底层的一个 PictureBox 对象是很危险的。PictureBox 类从 Control 类继承了 50 多个属性和 30 多个方法。作为一个实用的代理，PictureBoxProxy 对象应该将这些调用中的绝大多数进行转发。完全转发需要使用过多的方法，而且容易出错。另外如果 Control 类及其超类发生变化，那么我们还需要维护这些代码。

由于，传统的图像代理模式存在上述的问题。本系统根据需要，对图像代理模式进行变化，最后的实现的结构图如下：

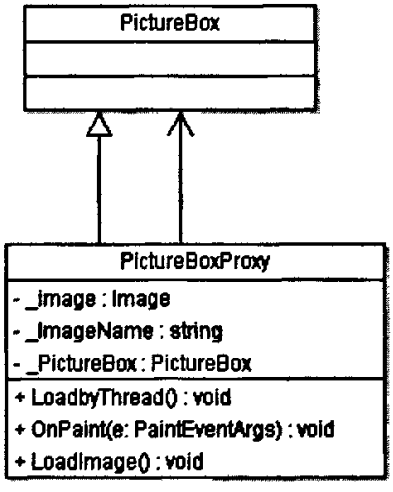


图 5-5 变化后的图像代理模式结构图

PictureBoxProxy 继承自 PictureBox 而不是 Control，这样避免因转发大量的方法和属性，需要使用过多的方法，容易出错的缺陷。取消了原代理中的 Load 方法，而改为将主动加载图片的时机放在 PictureBox 的 Paint 事件，即当该图片需要显示时才被加载，加载时如果图片较大，将开辟一个新的线程来完成加载，

主线程先暂时显示一个小的图片，表示正在加载。部分实现的代码如下：

```
protected override void OnPaint(PaintEventArgs e)
{
    //e.Graphics.DrawImage(Image, 0, 0);
    if (this._image == null)
    {//如果 _image==null，则开辟一个线程来加载图片
        this.LoadbyThead();
        e.Graphics.DrawString("Loading!....", new Font("Arial", 16), new
SolidBrush(Color.Black), new Point(0, 0));
    //在实际实现时被替换成在图片框里写一串文本，这里取决于习惯
    // e.Graphics.DrawImage(_PictureBox.Image, 0, 0);
    }
    else  //已经加载则显示加载后的图片
        e.Graphics.DrawImage(this._image , 0, 0);
}
private void LoadbyThead()
{
    Thread t=new Thread (new ThreadStart (LoadImage ));
    t.Start ();
    this.Invalidate();
}
private void LoadImage()
{
    this._image  = Image.FromFile(this.FileName);
}
```

5.2.3 基于抽象工厂的策略模式

有些图片涉及到隐私或者机密，因此在进入图像资料管理系统时需要输入口令，确保图像信息的安全和保护个人的隐私。不同的加密算法在效率和强度上是不同的，我们希望系统采用的加密算法是可以在多种加密算法中进行选择，而不必大量的修改代码，使用 Strategy 模式可以很好的满足我们的要求。另外使用 Strategy 模式时可以在运行时替换加密算法，经常更换加密算法也是提高系统的安全性能一种方法。对于口令的加密算法的处理，我们可以采用 Strategy 模式。加密策略接口 PassStrategy 包括两个操作，加密操作（Encode）将输入的明文字符串转换为密文字符串输出，解密操作（Decode）是加密操作的逆操作，将加密后的密文字符串转换为明文后输出。PassStrategy 的类图如图 5-6 所示。

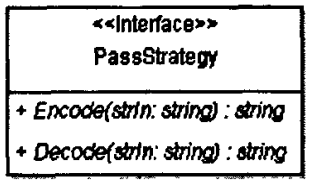


图 5-6 PassStrategy 接口

具体的加密算法继承并提供这个操作的具体实现。系统中使用的策略模式的结构图如下：

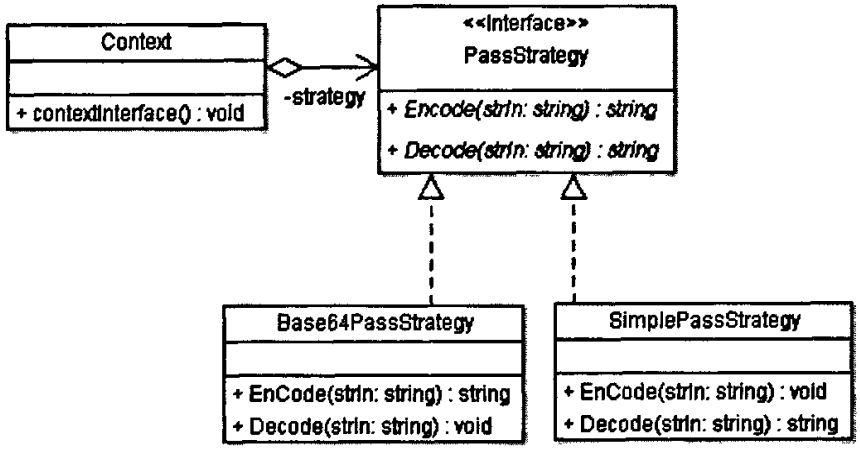


图 5-7 加密算法的策略模式结构图

从图 5—7 可以看出，加密策略的具体使用者(Context)聚合一个 PassStrategy 接口的实例，加密口令的操作一般发生在登录窗体和密码修改窗体。

直接使用 Strategy 模式，客户必须了解不同加密策略之间的差异。这将影响系统的封装性，降低系统的可扩展性和可靠性。我们可以采用第四章提出的基于抽象工厂的策略模式，将策略的选取封装在一个具体工厂中来实现以解决上述的问题。图 5-8 为加密算法基于抽象工厂的策略模式结构图。

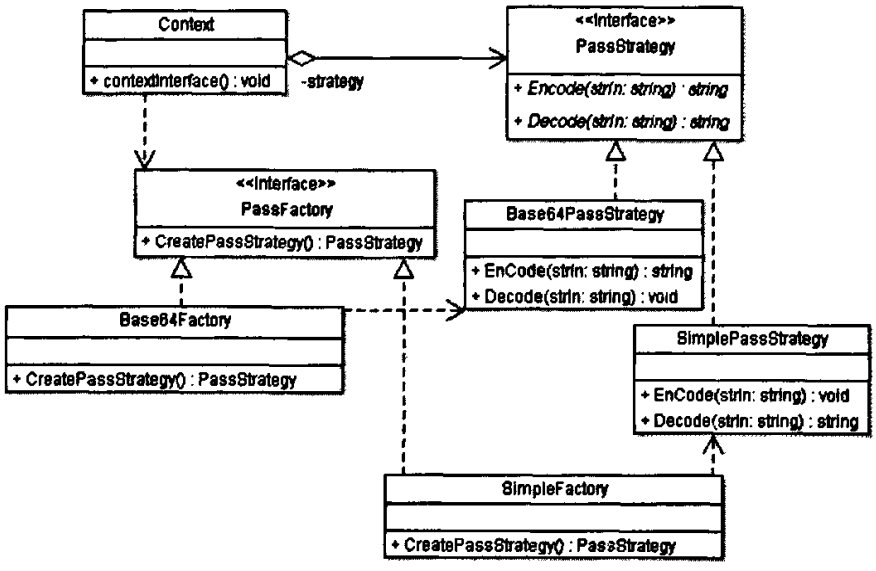


图 5-8 加密算法基于抽象工厂的策略模式结构图

因为有两处使用了加密策略，为了防止不同的地方加载了不同的加密策略，加密策略的配置只允许做一次，通过使用 Singleton 模式我们可以确保一个对象在系统中只有一个实例。我们将 Context 实现为一个单件类，其类图如图 5-9 所示，程序实现如下：

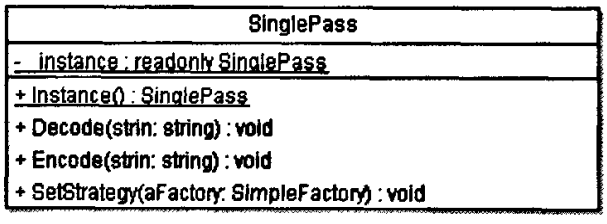


图 5-9 SinglePass 类图

```
//Abstract Factory 接口
public interface PassFactory
{
  PassStrategy  CreatePassStrategy();
}
//SinglePass 类的部分代码：
public class singlePass
{
    //static readonly 确保singlePass只初始化一次
    static readonly singlePass _instance = new singlePass();
    PassStrategy pass;
```

```
    public static singlePass Instance()
    {
        return _instance;
    }
    public void setPassStrategy(PassFactory aFactory)
    {//可以在运行时替换策略
        this.pass = aFactory.CreatePassStrategy();
    }       //对封装加密操作，将实际操作转发给pass
    public string Encode(string strin)
    {
        return pass.Encode(strin);
    }
    public string Decode(string strin)
    {
        return pass.Decode(strin);
    }
}
// Base64PassStrategy 实现的部分代码
public class Base64PassStrategy : PassStrategy
{
    public string Encode(string strin)
    {
        string str;
        byte[] byt = System.Text.Encoding.ASCII.GetBytes(strin);
        str = System.Convert.ToBase64String(byt);
        return str;
    }
    public string Decode(string strin)
    {
        byte[] byt = System.Convert.FromBase64String(strin);
        string str = System.Text.Encoding.ASCII.GetString(byt);
        return str;
    }
   }
public class Base64Factory : PassFactory
{//负责生成Base64加密算法的具体工厂
    public PassStrategy CreatePassStrategy()
    {
        return new Base64PassStrategy();
    }
}
//配置策略的实现的部分代码
public partial class FrmLogin : Form
{ //登陆窗体类
```

```
        private singlePass aPass;
         public FrmLogin(PassFactory aFactory)
           {
            aPass = singlePass.Instance();//singleton
            aPass.setPassStrategy(aFactory.CreatePassStrategy()); //配置策略
           }
    }
```

5.2.4 Adapter 模式

Adapter 模式能够将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。在此系统中，我们想使用 TreeView 控件来显示分类目录。分类目录包含了目录名称、目录 id 和父节点 id 信息，而 TreeView 的 TreeNode 本身是不包含这些信息的，不符合实际的需要。因此，我们可以定义一个 ItemInfo 接口，定义 TreeNode 的一个子类，同时这个子类实现了 ItemInfo 接口。结构如图 5-10 所示。

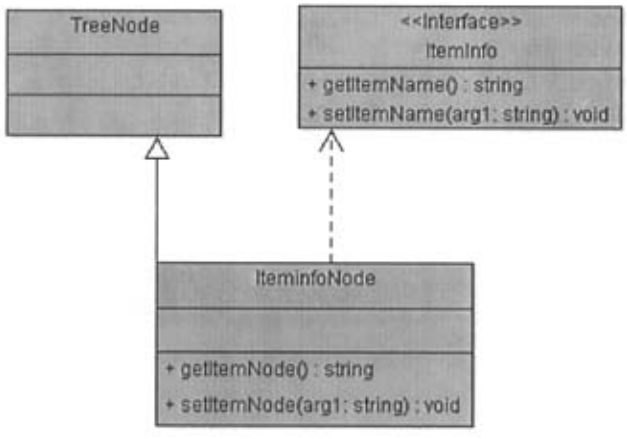


图 5-10 基于适配器模式的分类目录结构图

除了 Factory Method 模式、Proxy 模式、Stategy 模式、Adapter 模式这些模式以外。还间接的使用了诸如 Command 模式、Iterator 模式、Composite 模式、MVC 模式等。在.NET FCL 中，许多框架的实现运用了大量的设计模式，比如有一些 Windows 控件是容器控件，可以在它们内部包容其它的容器控件和非容器控件，像 Windows TreeView 这样的控件反映了一种树型结构，这样的控件在.NET FCL 就是用 Composite 模式实现；在 C#中，专门有个 IEnumerator 接口，它支持在集

合上进行简单迭代。C#中的 ArrayList 类，Stack 类，Queue 类都继承自这个 IEnumerator 接口。对于这些类对象的遍历使用了 Iterator 模式。在第三章，我们也通过分析说明了在新的高级程序设计语言中引入了大量的设计模式，在这里再次得到印证。

5.3 本章总结

通过已被实践证明的设计方案表示成设计模式，软件设计人员可以在类似的上下文中复用以往的设计经验。在面向对象的分析设计环境中，一种模式往往还代表了一种设计的思想和理念，指导软件设计人员从更高的层次上理解和认识面向对象技术的内涵。因此，作为软件复用技术的一个有力的支撑，设计模式在现代软件工程中越来越受到重视。然而由于软件设计人员没有正确把握和理解软件设计模式，又没有一种有效的方法来指导使用这些设计模式，因此在系统如何灵活运用设计模式成为一个难题。

本章总结了部分常用设计模式的应用规律，在此基础上，总结以往的软件设计经验，结合软件工程模型，提出了应用设计模式于软件项目设计的选取策略模型。最后以图像资料管理系统为实例，说明如何在一个软件项目中选取有效的设计模式，达到改进设计的目的。

第6章 总结与展望

设计模式的分类整理开始于Erich的博士论文，经过Erich等人的收集和整理，从众多的设计模式中选取了23个设计模式完成《设计模式——可复用面向对象软件的基础》一书。经过十多年的发展，已经有100多个模式，范围涉及软件设计的通用模式，特定领域设计模式如数据库模式等。同时设计模式的分类，描述等也出现多种方式。这一切都说明设计模式在软件项目开发过程发挥着重要的作用，人们对设计模式的研究日益重视。

本人所作的工作主要是以下三个方面。

（1）通过比较不同设计模式在不同程序设计语言的实现，得出模式与语言相辅相成共同促进的结论，指出语言和模式的发展应该将对方考虑进去。

设计模式是对一类设计问题的解决方案的总结，本质上应该独立于程序设计语言。但是，不同语言的语法各不相同，有独特的优势，也有自己的缺点和不足。那么，离开具体的语言谈模式的应用是不切实际的。许多书籍或参考文献也都提到了设计模式与语言的关系，如《设计模式——可复用面向对象软件的基础》一书在描述一个模式的示例时，有针对性的讨论C++、Java、Smalltalk这几种语言实现模式的区别。除了语法与语义的区别，每种语言提供的库、框架、通用环境和工具等，也会使得实现模式的效果呈现很大的差异。本文比较了Java和C#这些新型的高级面向对象程序语言，与C++和Visual Basic这些语言在某个模式实现上的效果的不同。通过比较Observer模式的Java实现和Visual Basic的实现方式和效果的不同，可以看到Java提供专门的类库使得该模式可以更方便地为开发者所使用，降低了实现难度，而Visual Basic语言由于自身语言先天性的不足，在实现此模式时存在一些硬伤。例如由于没有类继承机制，如果想对被观察者进行扩充，就需要直接修改或者新建一个模块，重新实现被观察者的接口，丧失了部分灵活性；通过比较C++、Visual Basic、C#这三种语言的实现差异，我们看到C#语言提出全新的类修饰符readonly，使得Singleton模式的实现更加的简单和安全。新型面向对象程序语言的发展使得设计模型实现变得简单、可靠、高效，另外设计模式的引入也丰富了语言的实现手段，使得语言更加强大，在具体的设计时可以更多解放设计人员，让他们更多的去关注业务而不是代码，由此可以断言程序设计语言和模式的发展将是相辅相成、互相促进的。语言的发展将使得模式的实现变得更简单，并且可以有效的避免原有缺陷，而模式的发展将使语言更加丰富和灵活。

（2）在第章分析了Visitor模式、Observer模式和Strategy模式的不足

之处并对它们进行改进。

传统的设计模式 Visitor 模式存在两个缺陷：1.集合中的对象元素的类型必须保持稳定，如果新增一种对象元素，势必引起大量的修改和重新编译。2.具体的 Visitor 子类必须实现访问所有元素的接口函数。解决上述缺陷可以利用 C#和 JAVA 等语言的反射机制。通过反射机制，在运行时获得对象的类型，从而避免出现抽象依赖于具体的情况，同时使得可以将不同用途的接口函数放入不同的接口类中。这样降低了各个类之间的耦合度，增强了设计的灵活性。我们还可以利用反射机制对抽象工厂、工厂方法、策略模式等模式进行改进。改进后一个直接的好处是，可以在运行时替换一个算法或一个产品而无需修改任何代码。利用.NET 的委托机制可以改进 Observer 模式的实现，通过将更新的方法委托给目标对象来实现更新通知，从而可以去掉 observer 类的抽象接口。这样做，增加了系统的灵活性。利用委托机制可以轻松实现 Template Method 模式、Mediator 模式等。分析策略模式，我们知道它可以实现单一算法的灵活的扩展或选择，但如果存在一组相关的算法，需要在运行时选择，实现时仍然要修改大量代码。本文结合 Abstract Factory 模式提出一种改进的 Strategy 模式——基于抽象工厂的策略模式，较好的解决了此类问题。

（3）提出一个设计模式的选取策略，并一个实际的软件项目中应用此模型解决设计模式的选取问题。

在软件项目中使用设计模式可以增强系统的可重用性，有效的应对变化。但是灵活正确的运用设计模式仍然是一个摆在开发人员面前的一个难题。本文在总结他人和自己设计模式应用经验的基础上，得出每种模式适合的应用场景，并提出了一个模式选取的模型，并且利用此模型较好的解决了“图像资料管理系统”中模式选取的问题。但是，在模式选取策略时，在做模型匹配时的决策过多依赖于设计者的经验，而不能一种可以度量的公式化操作。使得策略在执行时仍然会因人而异无法保证获得最佳选择。对设计模式的形式化表示的研究也许可以解决上述问题。设计模式的形式化表示为软件设计人员的模式自动获取提供了依据，特别是在结合了设计模式的形式化描述和规范匹配基础上研究设计模式的自动化获取，对推动模式的应用具有重大意义。模式的自动获取建立在需求和设计均需形式化描述的基础上，这对于大多数非形式化描述的软件并不适用。将设计模式的自动化获取与经验模式选取相结合，才能真正推动模式的广泛应用。这将是我的下一步研究方向。

参考文献

[1] Fred Brooks. No Silver Bullet. the Proceedings of the IFIP Tenth World Computing Conference, 1987, pp. 1069-76.

[2]Beck K . Patterns Generate Architecture. Europe an Conference on Objected-Oriented Programming. Springer-Verlag, 1994

[3] Brandon Glodfedder.模式的乐趣[M].北京：清华大学出版社，2003

[4] PublicationsCo,1995.25 3-267.Christopher Alexander. A pattern Language[M]. New York: Oxford University Press, 1977

[5]许幼鸣，徐锦，赵文耘等基于设计模式的软件重用[J].计算机工程，1999,25(3):1 3-1

[6]Alur D.Core J2EE pattern[M].北京:机械工业出版社,2002

[7]Eckel B.Java 编程思想.第二版[M].北京:机械工业出版社,2002

[8]张世博,周树杰,闵艳.JAVA 程序开发中的设计模式[J].微型电脑应用,2002,18(9):45-47

[9]Venners B. inside the Java Virtual Machine[M].2nd ed. New York : McGraw-Hill, 1999

[10]Kalin M Object-Oriendted Programming in JAVA[M].Englewood Clifft. J rentice -Hall , 2001

[11]Stark R F,Schimd J,Borger. Jva and the Java Virtual Machine Definition, Verification, Validation[M].Srinnger,2001

[12]Deepak A. Core J2EE Patterns [M]. 北京：机械工业出版社,2002

[13]Justin Couch.Java 2 Enterprise Editin Bible[M].Wiley Publishing,Inc,2001

[14]Bruce Eckel.Thinking in Java[M].3rd Edition,Published by Prenticehall,2002\12

[15]王小刚.基于设计模式的简化业务组件方法的研究与应用 [J] 计算机工程与应用,2005,14:90-93

[16]易可可,陈志刚. 基于MVC模式的WEB OA 系统设计与研究[J].计算机工程与应用,2005,14:112-115

[17]Chuck Cavaness.Programming Jakarta Struts[Z].2002-11

[18]Anna Maria Jankowka,Karl,Karl Kurbel. Model-View-Controller Design Pattern for Mobile and Desktop-based Application[M].MoMuc,2003

[19]杨睿,姚淑珍. 设计模式复用支持系统的设计实现[J]. 计算机工程, 2004, 30(11):80-81

[20]刘征驰,杨贯中.Visitor设计模式研究.计算机工程[J].2005,31(8):84-86
[21]邹娟,田玉敏. 软件设计模式的选择与实现[J]. 计算机工程, 2004, 30(10):79-82
[22]杨年华,张礼平.JAVA类库中的策略模式[J]. 计算机应用与软件,2004, 21(1):20-21
[23]刘海岩,锁志海. 设计模式及其在软件设计中的应用研究[J]. 西安交通大学学报,2005,39(10):1043-1046
[24]Jacobson I. The unified modeling language development process[M]. Botston:Addison Wesley,1998
[25]钟茂生,王明文.软件设计模式及其使用[J].计算机应用,2002,22(8):32-35
[26]lowayA TrottJ.设计模式精解:面向对象设计的新视角 [M].北京:清华大学出版社,2002
[27]童立,马远良.设计模式在基于组建的框架设计中的应用[J].计算机工程与应用,2002,(17):123-128
[28]崔立剑,吴平.java多线程设计模式研究[J].计算机与现代化,2006,11:92-95
[29]Steven John Metsker. C#设计模式[M]. 北京: 中国电力出版社. 2005
[30]陈贺明, 王彩玲. MFC中设计模式(Design Pattern)简析. 河南广播电视大学学报,2006,19(3):49-52
[31]吴雁军.MVC模式在J2EE程序中的应用[J].太原科技,2006,5:52-53
[32]赖英旭等.MVC模式在B/S系统开发中的应用研究[J].微计算机信息2006, 10(3):62-65
[33]陈颖峰,王玉红.MVC设计模式在WEB应用系统中的实现[J].承德石油高等专科学校学报, 2006, 8(3):32-36
[34]杨小姝, 高娇, 杨志东.MVC与COMMAND设计模式的应用.2006,22(4):59-63
[35]陈思楔, 郑声恩, 叶少珍.NET架构下结合MVC设计模式的招标系统的设计与实现[J]. 福州大学学报,2006,34(4):502-506
[36]刘玉华,王丽琴.Observer模式的探索及其在MFC中的应用[J].中国科技信息, 2006, 19:112-114
[37]齐索, 那正宏, 赵政灰.基于C#和XML的自动化测试框架系统的设计[J]. 计算机测量与控制.2006. 14(10):1304-1306
[38]丁振兰等.基于J2EE设计模式的小城镇电子政务系统研究应用[J]. 2006, 9: 169-171
[39]李志英,黄强,楼新远.基于反射机制的MVC业务模块简化方案[J].成都信息工程学院学报,2006,21(5):675-678

[40]张晓鑫,雍俊海,郑国勤,陈玉健.面向对象系统的应用编程接口设计模式[J].计算机工程与应用,2006,30:108-111

[41]计春雷.软件设计模式及其应用研究[J].上海电机学院学报,2006,5:46-49

[42]陈立岩，吕兴凤.设计模式在可复用MIS软件开发中的应用[J].佳木斯大学学报,2006,4:505-506

[43]仪维等.设计模式在软件PLC平台中的应用.机电一体化[J]，2006,5:47-39

[44]吕攀,余芳.应用设计模式构建数据库操作工厂[J].2006,26:4-6

[45]张组平，肖波.远程抄表系统中实现模式的研究[J].计算机工程，2006,32(19):233-235

[46] Wolfgang,Pree.Framework Development and Reuse Support[M]. USA: Manning

[47] Gamma E等. 设计模式一可复用面向对象软件的基础[M].北京:机械工业出版社,2002

致 谢

经过一年的忙碌，学习、研究、试验以及论文编写，毕业论文终于完成了。首先我要感谢我的指导老师杨邦荣教授。从论文的选题、内容的取舍到论文写作的审阅、修改，杨老师投入了大量的心血。没有杨老师的悉心指导，我不可能如期完成论文。

其次我要感谢所有的授课老师对我的教育和培养。他们严谨的治学态度和孜孜不倦的将治学的方法传授给学生，同时也言传身教给了我很多做人的道理。我要向老师们学习，鼓励自己以实际行动向老师表达敬意。

我还要感谢傅篱教授，他抽出宝贵的时间评阅了我的论文，并诚恳的提出论文中的不足和修改意义。很抱歉的是许多意见由于本人的时间和精力有限，没有能够加以改进和实现。只能放在下一步的研究中加以解决。

最后，我要感谢我的同事，他们在我的研究过程中，提供大量资料和宝贵的意见。

另外，还要感谢我的家人，没有他们的支持，我无法在工作、学习和研究中找到一个平衡点。

攻读硕士学位期间主要的研究成果

设计模式在面向对象软件复用中的应用 邵阳学院学报 2006，3 卷 2 期