Patterns in Action 4.0

A reference Application

Companion document to
Design Pattern Framework™ 4.0

by

Data & Object Factory, LLC
www.dofactory.com

Index

Introduction

Patterns in Action 4.0 is a reference application designed to demonstrate when, where, and how to use design patterns and best practices in building 3-tier, enterprise-quality applications.

What is new in Release 4.0

This latest release is optimized for .NET 4.0 and includes several significant changes and additions. New in his release are patterns and practices for ASP.NET MVC, Silverlight, and the Entity Framework. ASP.NET MVC includes Unit Testing, Dependency Injection, SEO (Search Engine Optimization), and Mocking. Silverlight includes RIA Services, MVVM (Model View ViewModel), and MEF (Managed Extensibility Framework). Please note that the *Silverlight Patterns 4.0* reference application is available as a separate solution with its own documentation.

In addition, significant improvements were made in these areas: the ADO.NET data access layer (including performance and SQL Injection protection), Request/Response patterns, ASP.NET Web Forms with Routing, and the inclusion of the Repository pattern. For users of the previous version (3.5) we have marked the major new or enhanced areas in this release with the following Icon.

Goals and Objectives

The following list of keywords summarizes the goals and objectives for the *Patterns in Action 4.0* reference application:

Educational – the purpose of *Patterns in Action 4.0* is to educate you on the importance of design, and when, where, and how to use them in a modern, 3-tier, enterprise application. In fact, Microsoft's new web platform ASP.NET MVC 2 is a testimony to the importance of design patterns as MVC is one of the oldest patterns in the computing industry.

Productivity – the design pattern knowledge and skills that you will gain from *Patterns in Action 4.0*, combined with the new .NET 4.0 features offer a great opportunity for enhanced productivity. The enhancements in .NET 4.0 are impressive in the areas of ASP.NET MVC, Silverlight, and Entity Framework. Design patterns help you establish architectures for optimal productivity.

Extensibility – extensibility is more or less implicit in applications that effectively use design patterns. Most design patterns promote the idea of coding against interfaces and base classes, which makes it easier to change and enhance your application at a later stage. The word 'extensibility' is overused in development shops, but if you have any experience building and deploying applications, you know that once your application has been released, and is well received, requests for changes and enhancements will be arriving almost immediately. With an extensible design you can easily accommodate these requests.

Simplicity – with simplicity we do not mean simplistic or unsophisticated. What we mean is that the architecture and design are as simple as possible, well thought out, clean, crisp, and easy-to-understand to all developers on the team.

Elegance - we firmly believe in 'elegant' code. Code should be easy-to-navigate, self-documenting, and should '*read like a story*'. In fact, elegance goes beyond code – it applies to all aspects of the application, ranging from the user interface (i.e. intuitive, easy-to-use, and attractive), all the way down to the database (i.e. a robust data model in 3rd normal form). Elegance is hard to quantify, but you know it when you see it. Design patterns, in effect, promote the construction of elegant object-oriented solutions.

Maintainability - building maintainable code goes hand in hand with the two previous points: simplicity and elegance. Code that is simple and elegant is 1) easy-to-navigate,

2) easy-to-understand, 3) easy-to-explain to colleagues, and therefore, much easier to support and maintain.

Vertical Tiers – actually, we at DoFactory coined this term. Please do not confuse vertical tiers with horizontal n-tier design discussed later in this document. Based on our experience, applications that are designed around autonomous functional modules (vertical tiers) are the easiest to understand and maintain. With 'vertical tiers' we mean vertical 'slices' of the application each with their own particular functional focus. Examples include: employee maintenance, account management, reporting, inventory control, and document management. Not only do developers benefit from clearly defined vertical tiers, all other stakeholders will benefit as well, including analysts, designers, programmers, testers, data base modelers, decision makers, and ultimately the end-users.

> Applications frequently do not have clearly marked functional areas. Let's look at an example. Say, you are planning to build a large corporate system that, among other things, manages employees. Without knowing the exact requirements, you already know, ahead of time, that there will be an employee vertical tier. This employee module is where employees can be *searched*, *listed*, *added*, *edited*, *deleted, viewed,* and *printed*. These are all basic operations that apply to any principal entity in an application.
>
> In addition, as a developer you know there will be an employee database table (possibly named 'employee', 'person', or 'party'), an employee business object, and an employee data access component. After reading this document, you will also realize that the application may have an employee façade (or service).
>
> We believe that the best applications (granted, 'best' is subjective) are built by architects who think in vertical tiers and then apply the design patterns to make these 'slices of functionality' a reality.

Interestingly, ASP.NET MVC 2 introduced a new feature called *Areas*. They are designed to help developers better organize their code in terms of functional areas.

These new Areas are very much in line with the concept of coding along Vertical Tiers and we strongly suggest you use them in your own projects.

Enterprise Architecture – building enterprise level applications requires deep understanding of enterprise architecture and design which includes proven design patterns and best practices. Designing multi-user applications (supporting hundreds or perhaps thousands of concurrent users) requires that you consider complex issues such as scalability, redundancy, fail-over, security, transaction management, performance, error handling, logging, and more. If you are involved in building comprehensive, business critical systems, you are expected to bring to the table practical experience as well as familiarity with design patterns and best practice techniques.

What is Patterns in Action 4.0

Functionally, *Patterns in Action 4.0* is an e-commerce solution in which shoppers search and browse a catalog of electronic products. Products are organized by category. Users select products, view their details, and add these to their shopping carts. Shopping carts can be managed by changing quantities and removing items. In the shopping cart, shipping costs are computed based on the shipping method selected. A separate administrative area allows administrators to view and maintain (add, edit, delete) customer records as well as analyze customer orders and order details.

From an architectural perspective, *Patterns in Action 4.0* is many applications in one. A core element in this solution is that it has a central, well-defined service layer (Application Façade) that exposes common e-commerce application tasks as a WCF-hosted service. This service is consumed by 4 separate client applications built on 4 different platforms: ASP.NET MVC, ASP.NET Web Forms, Windows Forms, and WPF. Each client technology consumes the *exact same public service interface*. This design in which client applications consume a Web Service is called Service Oriented Architecture (SOA).

About this document

The best way to read this document is from beginning to end. Each section builds on the previous one and it is best to follow along in a linear fashion. It contains the following six sections:

Setup and Configuration: This section describes how to setup and configure the application. It discusses the .NET solution, the database, and the web.config configuration files.

Documentation: This section presents the documentation and references available for the application. First, there is this document (the one you're reading right now). Additional documentation can be found in:

1) in-line code comments, and
2) class diagrams for each of the Projects in this Solution

Application Functionality: This section presents the functionality of the application by stepping you through the Web e-commerce project in which users shop and where administrators manage customer's records and their orders. It also discusses the web service and the additional web service clients that are coded in Windows Forms and WPF.

Application Architecture: This section provides an overview of the 3-tiers used to construct the application: the Presentation tier, the Business tier, and the Data tier. It discusses how the different tiers communicate (i.e. who calls who) as well as the best places to add any 'non-functional' items, such as, authentication, authorization, data validation, and transaction management.

. NET Solutions and Projects: This section looks at the .NET Solution with its numerous projects. The projects are organized according to the solution's 3-tier architecture and it is important that you have a good grasp of this structure. You will learn how to setup and organize your own projects which will give clarity on where particular parts of the application should reside.

Design Patterns and Best Practices: This section lists and catalogues the numerous design patterns that are used in *Patterns in Action 4.0*. They are organized in 4 separate pattern groups:

1) *Gang of Four Patterns*,
2) *Enterprise Patterns*,
3) *SOA and Messaging Patterns*, and
4) *Model-View Patterns*.

Setup and Configuration

This section discusses how to setup and configure *Patterns in Action 4.0*. The solution consists of 20 projects, including class libraries, ASP.NET MVC, ASP.NET Web Forms, Windows Forms, WPF, and WCF Host and Service projects.

Solution setup:

If you followed the standard installation steps you will find your solution in \Patterns in Action\ under these folders:

On Windows XP:

Files reside in the My Documents folder. The first line is C# and second is VB edition:

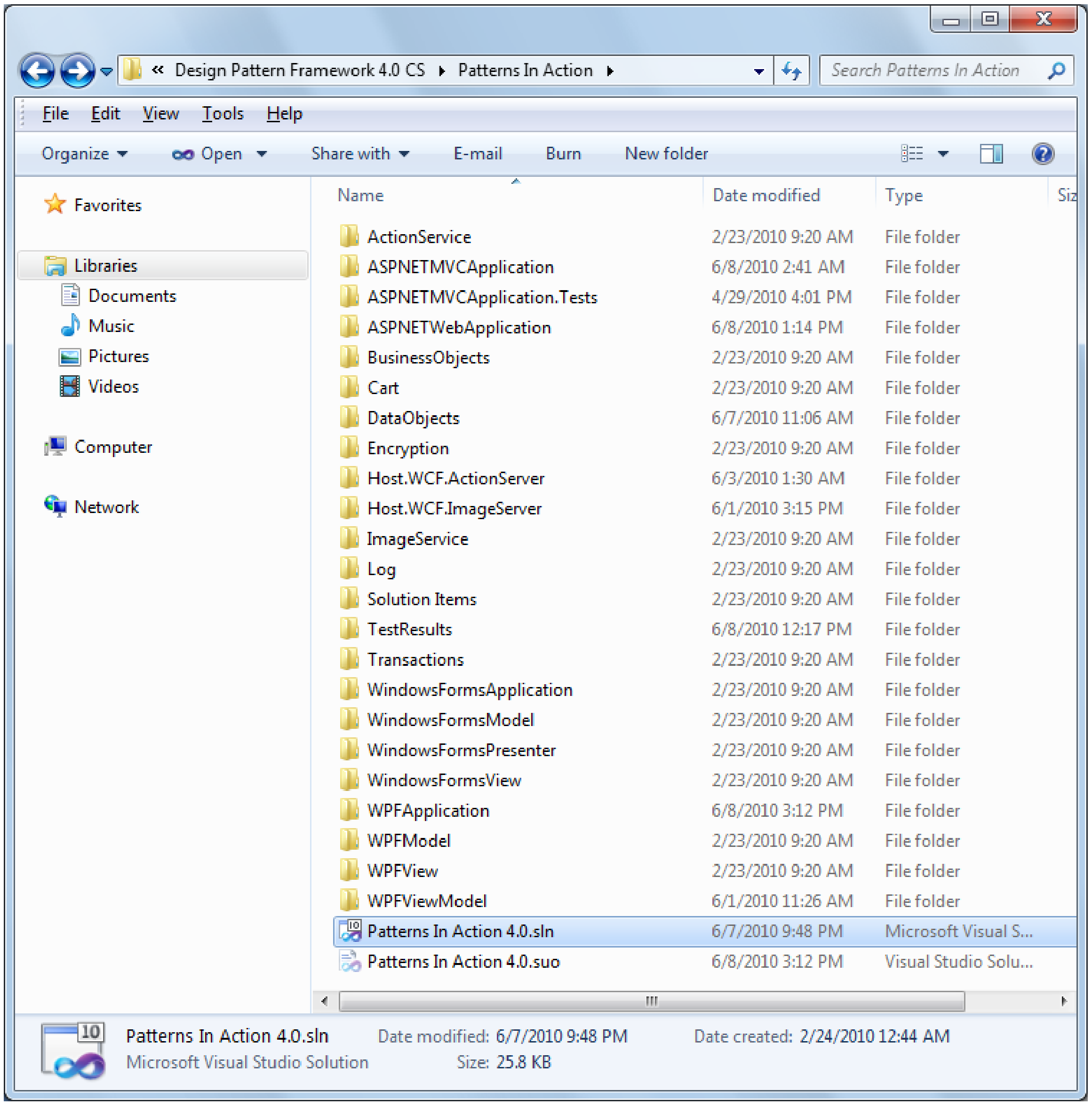
C:\Documents and Settings\%username%\My Documents\DoFactory Patterns 4.0 CS.**
C:\Documents and Settings\%username%\My Documents\DoFactory Patterns 4.0 VB.**

On Vista / Windows 7:

The first line is C# and second is VB edition:

C:\Users\%username%\Documents\DoFactory Patterns 4.0 CS.**
C:\Users\%username%\Documents\DoFactory Patterns 4.0 VB.**

Pattern in Action 4.0 can be launched from your Windows menu or by selection the Patterns in Action 4.0 solution file in the solution folder. Here we go straight to the solution file. Your folder structure should look like this:

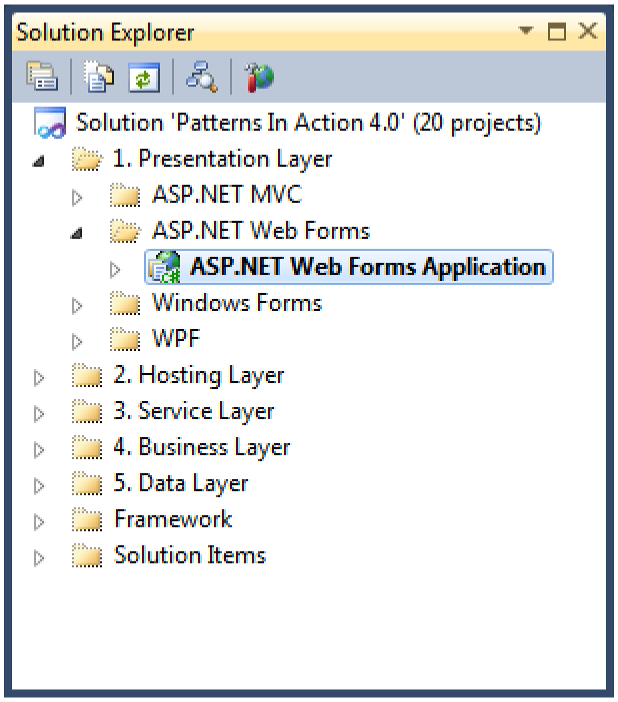

« Design Pattern Framework 4.0 CS ▸ Patterns In Action ▸
Search Patterns In Action
File Edit View Tools Help
Organize ▾ Open ▾ Share with ▾ E-mail Burn New folder
Favorites
Libraries
Documents
Music
Pictures
Videos
Computer
Network

Name	Date modified	Type
ActionService	2/23/2010 9:20 AM	File folder
ASPNETMVCApplication	6/8/2010 2:41 AM	File folder
ASPNETMVCApplication.Tests	4/29/2010 4:01 PM	File folder
ASPNETWebApplication	6/8/2010 1:14 PM	File folder
BusinessObjects	2/23/2010 9:20 AM	File folder
Cart	2/23/2010 9:20 AM	File folder
DataObjects	6/7/2010 11:06 AM	File folder
Encryption	2/23/2010 9:20 AM	File folder
Host.WCF.ActionServer	6/3/2010 1:30 AM	File folder
Host.WCF.ImageServer	6/1/2010 3:15 PM	File folder
ImageService	2/23/2010 9:20 AM	File folder
Log	2/23/2010 9:20 AM	File folder
Solution Items	2/23/2010 9:20 AM	File folder
TestResults	6/8/2010 12:17 PM	File folder
Transactions	2/23/2010 9:20 AM	File folder
WindowsFormsApplication	2/23/2010 9:20 AM	File folder
WindowsFormsModel	2/23/2010 9:20 AM	File folder
WindowsFormsPresenter	2/23/2010 9:20 AM	File folder
WindowsFormsView	2/23/2010 9:20 AM	File folder
WPFApplication	6/8/2010 3:12 PM	File folder
WPFModel	2/23/2010 9:20 AM	File folder
WPFView	2/23/2010 9:20 AM	File folder
WPFViewModel	6/1/2010 11:26 AM	File folder
Patterns In Action 4.0.sln	6/7/2010 9:48 PM	Microsoft Visual S...
Patterns In Action 4.0.suo	6/8/2010 3:12 PM	Visual Studio Solu...

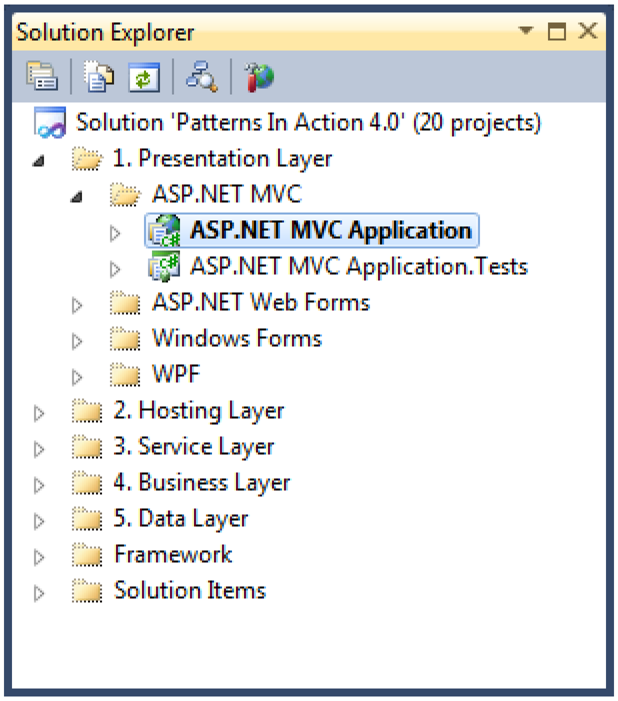
Patterns In Action 4.0.sln Date modified: 6/7/2010 9:48 PM Date created: 2/24/2010 12:44 AM
Microsoft Visual Studio Solution Size: 25.8 KB

Double click or right click on the solution file named "**Patterns in Action 4.0.sln**". Visual Studio 2010 launches and opens the solution. Once in Visual Studio, you can select one of four Startup Projects; they all reside under the *Presentation Layer* folder.

1) To run the ASP.NET Web Application set the **ASP.NET Web Form Application** as the Startup Project (it will display in bold). See below.

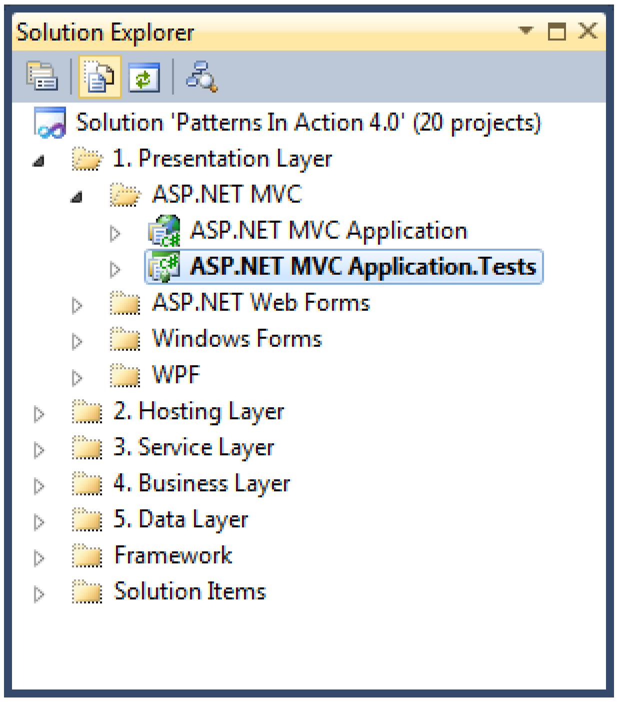


2) To run the new ASP.NET MVC Application set the **ASP.NET MVC Application** as the Startup Project (it will display in bold). See below.



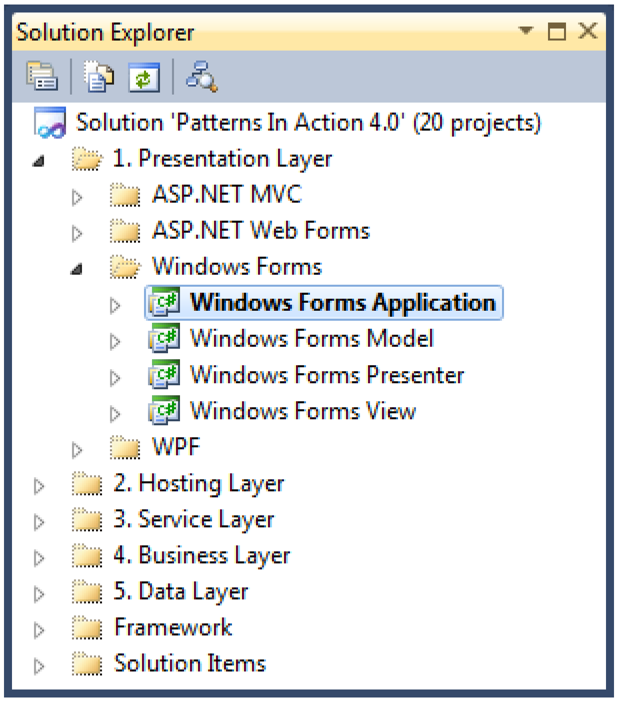
Solution 'Patterns In Action 4.0' (20 projects)
- 1. Presentation Layer
  - ASP.NET MVC
    - **ASP.NET MVC Application**
    - ASP.NET MVC Application.Tests
  - ASP.NET Web Forms
  - Windows Forms
  - WPF
- 2. Hosting Layer
- 3. Service Layer
- 4. Business Layer
- 5. Data Layer
- Framework
- Solution Items

To run Unit tests against the ASP.NET MVC web application select the **ASP.NET MVC Application.Tests** project as our startup project (see below). Testing is discussed later in this document.

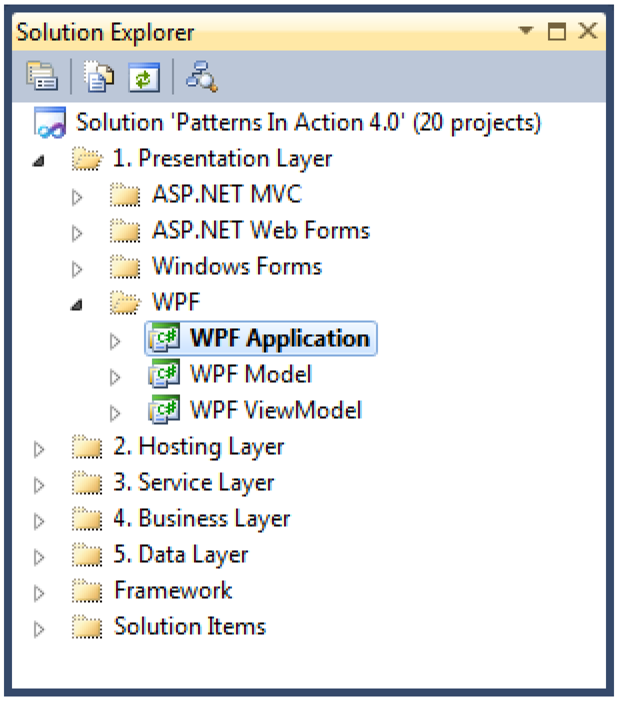


Solution 'Patterns In Action 4.0' (20 projects)
- 1. Presentation Layer
  - ASP.NET MVC
    - ASP.NET MVC Application
    - **ASP.NET MVC Application.Tests**
  - ASP.NET Web Forms
  - Windows Forms
  - WPF
- 2. Hosting Layer
- 3. Service Layer
- 4. Business Layer
- 5. Data Layer
- Framework
- Solution Items

3) To run the Windows Forms Application, select the **Windows Forms Application** project as your Startup Project. See below.



4) Finally, you can run the WPF Application by selecting the **WPF Application** project as your Startup Project. See below.



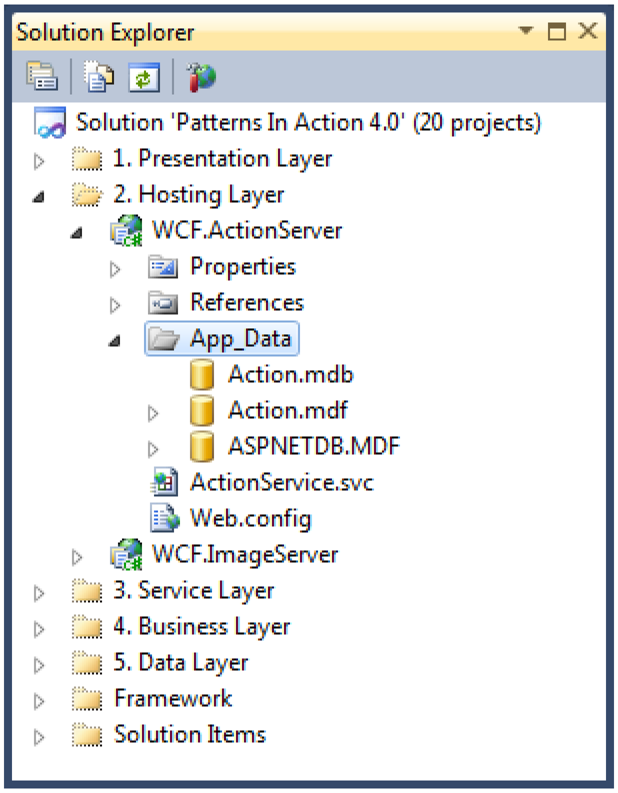
Solution 'Patterns In Action 4.0' (20 projects)
- 1. Presentation Layer
  - ASP.NET MVC
  - ASP.NET Web Forms
  - Windows Forms
  - WPF
    - WPF Application
    - WPF Model
    - WPF ViewModel
- 2. Hosting Layer
- 3. Service Layer
- 4. Business Layer
- 5. Data Layer
- Framework
- Solution Items

Database Setup:

Out of the box, *Patterns in Action 4.0* supports three databases: MS Access, SQL Express, and SQL Server. By default the application uses the Entity Framework to access a local SQL Express database called *Action.mdf*.

The application supports Membership and is configured to use the standard SQL Express membership database named: *ASPNETDB.MDF***.**

The MS Access and SQL Express databases reside under the Hosting layer folder in a project named **WCF.ActionServer** in a folder named *\App_Data*. See image below.

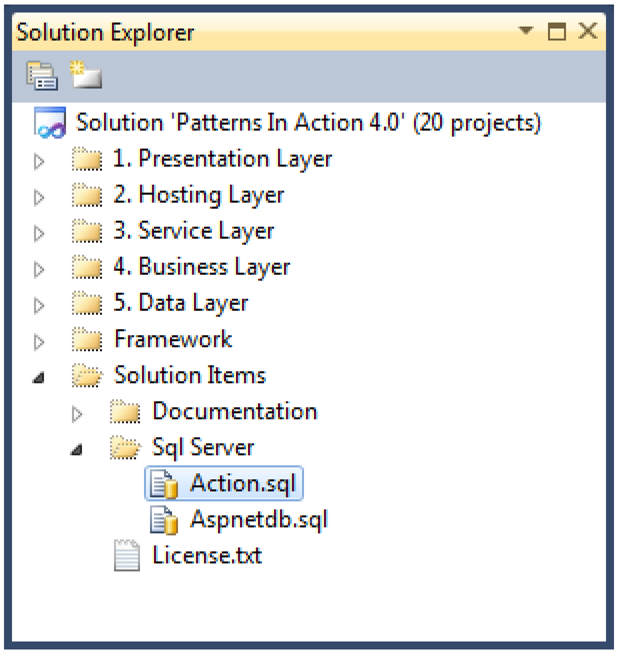


By placing these databases in this hosting project, rather than in the standard ASP.NET *\App-Data* folder, we allow different clients (presentation tiers) to access the same database via the same service layer entry point (the Façade). This way, there is no need to maintain duplicate Web.config files or duplicate databases. Also, connection strings for all databases reside in a single place while supporting 4 different client types (ASP.NET Web Forms, ASP.NET MVC, Windows Forms, and WPF Application).

Important: It should be noted that in your own projects (ASP.NET or other) you should consider placing the Web.config with connection strings and the databases directly in the ASP.NET project folder itself. Of course, it all depends on your specific requirements and deployment environment. In *Patterns in Action 4.0* we are demonstrating the use of a single WCF hosted Service Layer that allows access from several different clients. We have more to say about deployment configurations later in this document.

Using SQL Server

If you wish to use SQL Server, rather than SQL Express for *Patterns in Action 4.0* follow the following simple steps. First, create an empty database named **Action** (any other name is fine too). Second, run the script named *Action.sql* against this new database. This will create the data model as well as enter the required data into the database. The file *Action.sql* is located in the .NET solution under a folder named *\Solution Items\Sql Server.* See image below. As a last step will need to adjust your web.config file (described in the next section on web.config) and you're ready to go.



Using Oracle

For .NET developers using Oracle, some placeholder data access code for Oracle is included (for ADO.NET, but not for LINQ-to-SQL or Entity Framework). The port to Oracle should be relatively easy because the application uses little or no vendor specific database data types or stored procedures. You will need to program the Oracle specific data access components, but again, this will be easy because the SQL is almost identical to the SQL used in SQL Server. Finally, you will need to build and populate the Oracle database.

Using Membership database

To secure the application, *Patterns in Action 4.0* uses Microsoft's built-in Membership services. By default, these services have their own database named ASPNETDB. The script to generate this database (and one user account) can be found in a file named *Aspnetdb.sql* (see image above).

In *Patterns in Action 4.0*, the Membership database always uses SQL Express, whether you run the main application against MS Access, SQL Express, or SQL Server. However, with little effort, you can configure Membership to use SQL Server. If you look at the Membership sections in Web.config (under the *WCF.ActionServer* project) you notice that *connectionStringName* is set to *LocalSqlServer*. By default *LocalSqlServer* is defined in machine.config as:

```
Data Source=".\SQLEXPRESS;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;User
Instance=true" providerName="System.Data.SqlClient"
```

If necessary, you can change the connectionStringName to reference a SQL Server based membership provider (rather than our default: SQL Express).

Web.config setup

Web.config is the configuration file for web sites and web services. In it you configure database connections and other custom application options. In *Patterns in Action 4.0* the database connection strings as well as the membership configuration are maintained at a single place: the web.config file under the.*WCF.ActionServer* project. Note that web.config under the *ASP.NET Web Application* project does not hold any database settings, only a few custom application settings.

First we discuss the web.config file under WCF.ActionServer followed by a discussion on web.config under the ASP.NET Applications.

Web.config under WCF.ActionServer

The most important settings are listed below.

```
<appSettings>
  <add key="DataProvider" value="System.Data.SqlClient"/>
  <add key="ConnectionStringName" value="EntityFramework.SqlExpress"/>
  <add key="LogSeverity" value="Error" />
  <add key="ShippingMethod" value="Fedex" />
</appSettings>

<connectionStrings>

   <add name="EntityFramework.SqlExpress"
connectionString="metadata=res://*/EntityFramework.Action.csdl|res://*/EntityFrame
work.Action.ssdl|res://*/EntityFramework.Action.msl;provider=System.Data.SqlClient
;provider connection string=&quot;Data
Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\Action.mdf;Integrated
Security=True;MultipleActiveResultSets=True&quot;"
providerName="System.Data.EntityClient" />

  <add name="EntityFramework.SqlServer"
connectionString="metadata=res://*/EntityFramework.Action.csdl|res://*/EntityFrame
work.Action.ssdl|res://*/EntityFramework.Action.msl;provider=System.Data.SqlClient
;provider connection string=&quot;Data Source=localhost;Initial
Catalog=Action;Integrated Security=True&quot;"
providerName="System.Data.EntityClient" />

  <add name="ADO.NET.Access"
     connectionString="Provider=Microsoft.Jet.OLEDB.4.0;Data
       Source=|DataDirectory|action.mdb"/>
  <add name="ADO.NET.SqlExpress" connectionString="Data
       Source=.\SQLExpress;Integrated Security=True;User
       Instance=True;AttachDBFilename=|DataDirectory|Action.mdf"/>
  <add name="ADO.NET.SqlServer" connectionString="Data
       Source=Factory;Initial Catalog=Action;Integrated
       Security=True" />
  <add name="ADO.NET.Oracle" connectionString="Data
       Source=MyOracleActionDB;User Id=scott;Password=tiger;
       Integrated Security=no;"/>
  <add name="LinqToSql.SqlExpress" connectionString="Data
       Source=.\SQLExpress;Integrated Security=True;User
       Instance=True;AttachDBFilename=|DataDirectory|Action.mdf"/>
  <add name="LinqToSql.SqlServer" connectionString="Data
       Source=Factory;Initial Catalog=Action;Integrated
       Security=True" />
</connectionStrings>
```

The data access layer is configurable to use ADO.NET, LINQ to SQL, or Entity Framework. Below are the valid Web.config settings:

Web.config values for Entity Framework

Database	ConnectionStringName	Provider
SQL Express	EntityFramework.SqlExpress	n/a
SQL Server	EntityFramework.SqlServer	n/a

Web.config values for Linq-To-Sql

Database	ConnectionStringName	Provider
SQL Express	LinqToSql.SqlExpress	n/a
SQL Server	LinqToSql.SqlServer	n/a

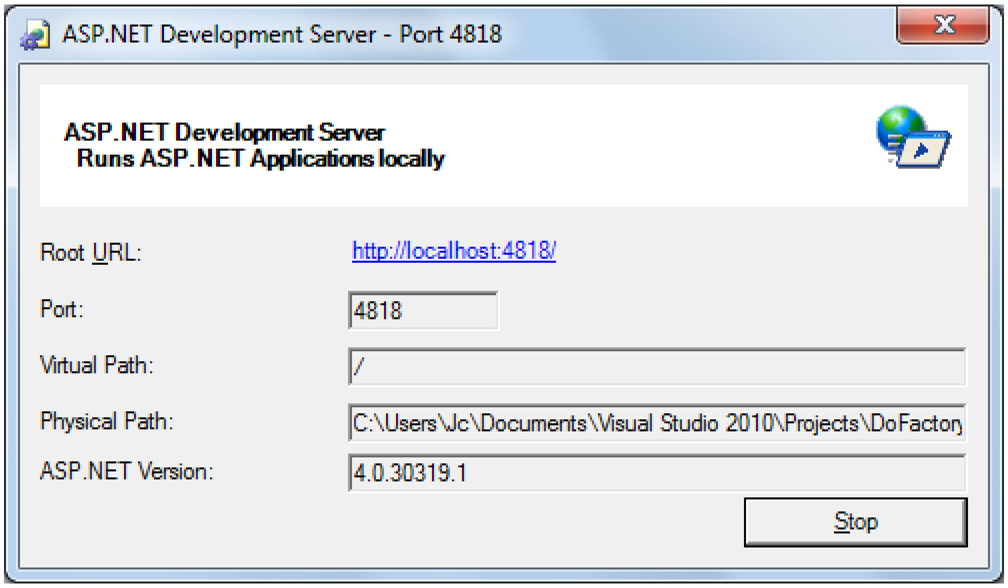
Web.config values for ADO.NET

Database	ConnectionStringName	Provider
MS Access	ADO.NET.Access	System.Data.OleDb
SQL Express	ADO.NET.SqlExpress	System.Data.SqlClient
SQL Server	ADO.NET.SqlServer	System.Data.SqlClient
Oracle	ADO.NET.Oracle	System.Data.OracleClient

Note: For LINQ-to-SQL and Entity Framework the provider setting is not used.

The default ConnectionStringName is EntityFramework.SqlExpress. This indicates that the data access layer uses Entity Framework to access the Sql Express database. As an example: if you want it to change to ADO.NET, change the ConnectionStringName to ADO.NET.SqlExpress and set the Provider to System.Data.Sql.Client. No other changes are required.

When changing the database connections, make sure you stop all ASP.NET development servers before re-running the application. See image below showing you where the stop buttons are located (note: there are other ways to stop these services).


ASP.NET Development Server - Port 4818
ASP.NET Development Server
Runs ASP.NET Applications locally

Root URL: http://localhost:4818/
Port: 4818
Virtual Path: /
Physical Path: C:\Users\Jc\Documents\Visual Studio 2010\Projects\DoFactory
ASP.NET Version: 4.0.30319.1

With LogSeverity you specify at what level error messages should be logged by the Logging system. For example if the log severity level is 'Warning', and a log message of level 'Error' is issued by the application, then it is logged (because the severity for a 'Error' is higher than or equal to 'Warning'). A log message with severity level 'Info' will not be logged. Logging is explained later in this document.

ShippingMethod is also configured in this web.config. It specifies the default shipping method used in every user's shopping cart. Possible values are: Fedex, UPS, and USPS (US Postal Service). Users have the ability to override the default setting and choose their own shipping method. Shipping methods are explained later in this document.

Web.config under ASP.NET Applications

ASP.NET specific settings are configured in the Web.config files in the ASP.NET Web Application. Three application settings are configured under the <appSettings> section.

```
<appSettings>
    <!-- Log Severity. Options are: Debug, Info, Warning, Error,
     Warning, or Fatal -->
    <add key="LogSeverity" value="Error"/>
    <!-- Client tag: identifier or Service communication -->
    <add key="ClientTag" value="ABC123"/>
    <!-- Url for image service -->
    <add key="ImageService"
    value="http://localhost:4818/ImageService.svc/" />
</appSettings>
```

LogSeverity works the same as in the previous web.config.

ClientTag is a unique identifier that is required for every client of a service. For example, Amazon provides each company that is authorized to use its web services a client tag. This identifies you as a legitimate customer. Without a valid ClientTag the WCF service API will not be accessible. For demonstration purposes we use this ClientTag: **ABC123**.

ImageService is the Url of the image services from which Customer and Product images are retrieved. Note: the port number for the VB edition of *Patterns in Action 4.0* is different from the C# edition.

Documentation

Patterns in Action 4.0 comes with several sources of documentation. First, there is this document, the one you are currently reading. It is named *Patterns in Action 4.0.pdf*. This is the primary document that will guide you through the setup, functionality, architecture, and design patterns used in the application.

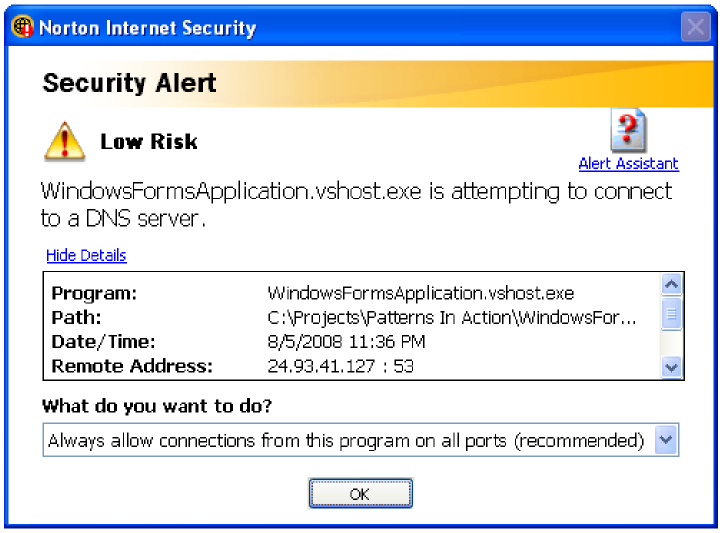
Second, the application code itself is well commented. Each class has XML comments with <summary> information, many with additional <remarks>. Design patterns that are used in these classes are listed. Most public and private members (methods, properties, etc) have comments as well.

Third, each project has a folder named _*UML Diagram*\. It contains a class diagram of the major classes in the project. If you are comfortable reading UML diagrams then these will be helpful in understanding the classes and their relationships.

Just a reminder that the *Silverlight Patterns 4.0* application is available as a separate solution with its own documentation.

Running for the first time

When you run the application for the first time and have enabled Internet Security, say with Norton Internet Security, you may encounter a dialog like the one below. Simply select the option that states ‘always allow connection from this program’. You may see this dialog several times, once for each of the hosted services.



WindowsFormsApplication.vshost.exe is attempting to connect to a DNS server.

Program:	WindowsFormsApplication.vshost.exe
Path:	C:\Projects\Patterns In Action\WindowsFor...
Date/Time:	8/5/2008 11:36 PM
Remote Address:	24.93.41.127 : 53

What do you want to do?

Always allow connections from this program on all ports (recommended)

Similarly, on Windows 7 (or Vista) if you have the Framework installed under the Program Files folders, you may be prompted that you need Administration privileges. Simply accept, Visual Studio will restart, and you should be ready to go.

Application Functionality

This section presents an overview of the functionality of the *Patterns in Action 4.0*. We explain what the application does and how users can navigate through the system The functionality is fairly simple and easy-to-understand so that we maximize learning and minimize unnecessary distractions.

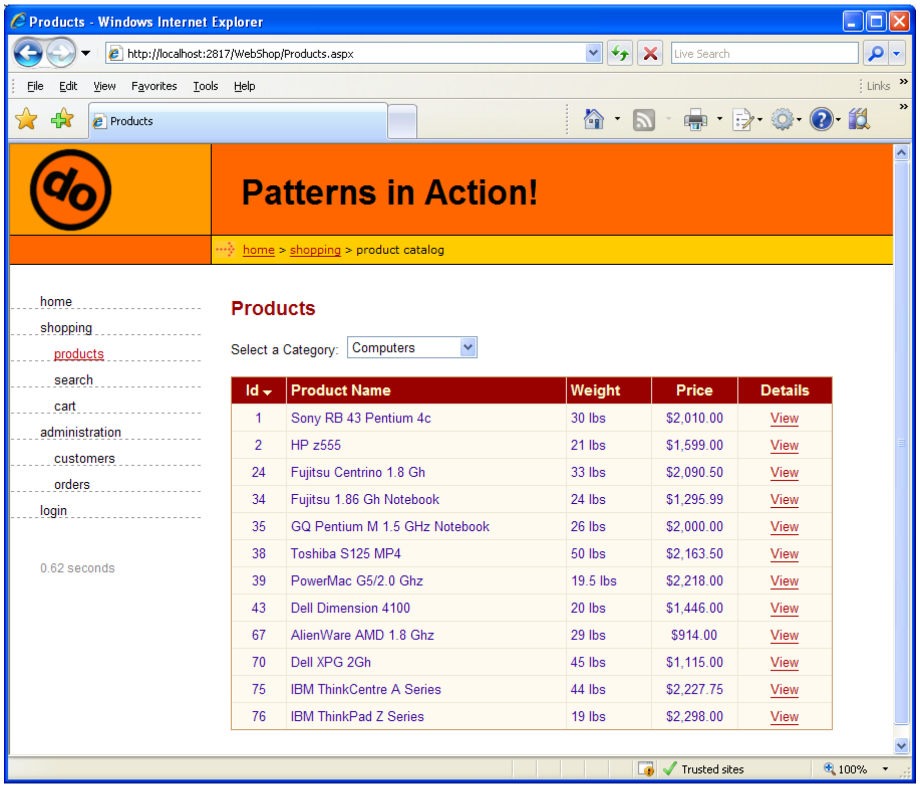
You can *experience Patterns in Action 4.0* in one of 3 ways:

1) as a Web Application (Web Forms or MVC)
2) as a Windows Application
3) as a WPF Application

In this section, we combined ASP.NET Web Forms and ASP.NET MVC because their UIs are essentially the same. So, next, we will next discuss each of the 3 *experiences*.

Web Application (ASP.NET Web Forms and MVC)

First of all, release 4.0 of the Web app has a new look-and-feel. Changing the images, CSS, and Skin files made this remarkably easy (note: only ASP.NET Web Forms supports skin files). To demonstrate the before-and-after UI, here is a screenshot of the original 3.5 version (the 4.0 screenshots are below):

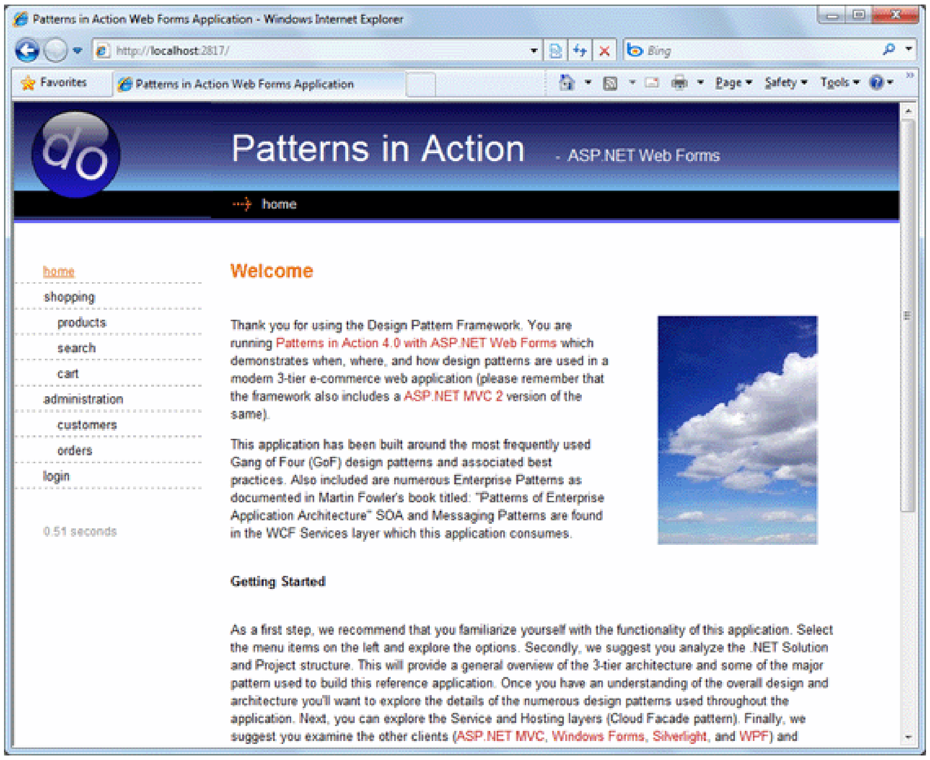

Products - Windows Internet Explorer
http://localhost:2817/WebShop/Products.aspx
Live Search
File Edit View Favorites Tools Help
Links
Products
Patterns in Action!
home > shopping > product catalog
home
shopping
products
search
cart
administration
customers
orders
login
0.62 seconds
Trusted sites
100%

Products

Select a Category: Computers

Id	Product Name	Weight	Price	Details
1	Sony RB 43 Pentium 4c	30 lbs	$2,010.00	View
2	HP z555	21 lbs	$1,599.00	View
24	Fujitsu Centrino 1.8 Gh	33 lbs	$2,090.50	View
34	Fujitsu 1.86 Gh Notebook	24 lbs	$1,295.99	View
35	GQ Pentium M 1.5 GHz Notebook	26 lbs	$2,000.00	View
38	Toshiba S125 MP4	50 lbs	$2,163.50	View
39	PowerMac G5/2.0 Ghz	19.5 lbs	$2,218.00	View
43	Dell Dimension 4100	20 lbs	$1,446.00	View
67	AlienWare AMD 1.8 Ghz	29 lbs	$914.00	View
70	Dell XPG 2Gh	45 lbs	$1,115.00	View
75	IBM ThinkCentre A Series	44 lbs	$2,227.75	View
76	IBM ThinkPad Z Series	19 lbs	$2,298.00	View

The Web Application (both Web Forms and MVC) is basic e-commerce application. To run it, select one of the web projects as your Startup Project and select Run (hit F5). The initial startup may appear a bit slow. This is due to the fact that this application is a client to a couple web services that need to startup before the application can render its pages. Once the services are running, the application will perform well. Details of this architecture are discussed later in this document. The opening screen is as follows:



Welcome

Thank you for using the Design Pattern Framework. You are running Patterns in Action 4.0 with ASP.NET Web Forms which demonstrates when, where, and how design patterns are used in a modern 3-tier e-commerce web application (please remember that the framework also includes a ASP.NET MVC 2 version of the same).

This application has been built around the most frequently used Gang of Four (GoF) design patterns and associated best practices. Also included are numerous Enterprise Patterns as documented in Martin Fowler's book titled: "Patterns of Enterprise Application Architecture" SOA and Messaging Patterns are found in the WCF Services layer which this application consumes.

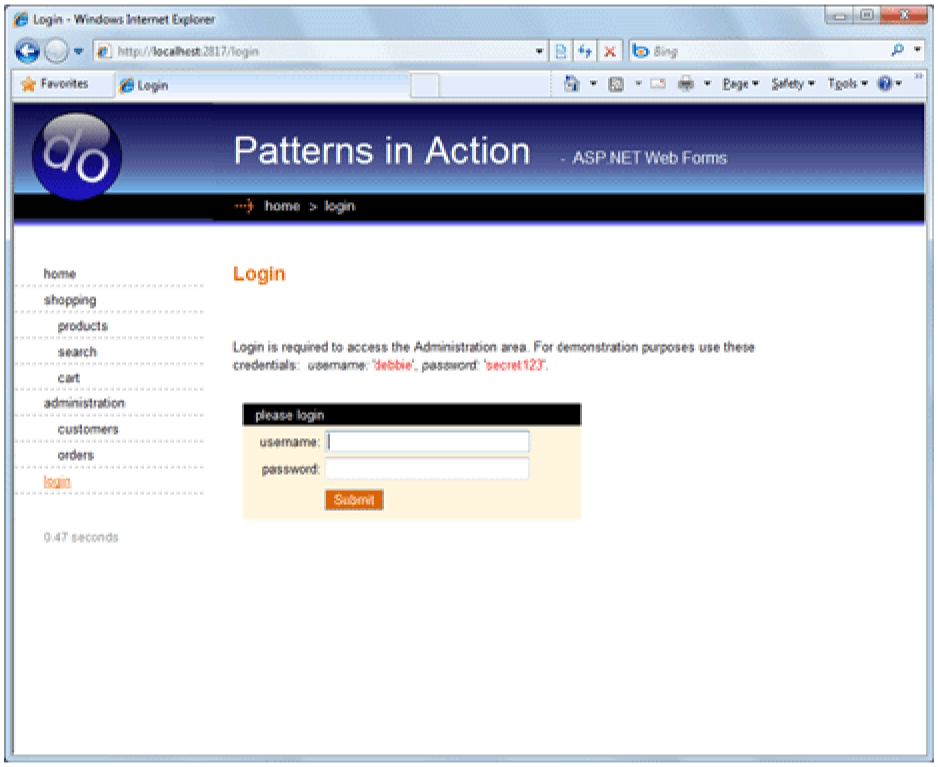
Getting Started

As a first step, we recommend that you familiarize yourself with the functionality of this application. Select the menu items on the left and explore the options. Secondly, we suggest you analyze the .NET Solution and Project structure. This will provide a general overview of the 3-tier architecture and some of the major pattern used to build this reference application. Once you have an understanding of the overall design and architecture you'll want to explore the details of the numerous design patterns used throughout the application. Next, you can explore the Service and Hosting layers (Cloud Facade pattern). Finally, we suggest you examine the other clients (ASP.NET MVC, Windows Forms, Silverlight, and WPF) and

Web App home page

You start off in the role of an unauthenticated user who is shopping for electronic products. Start with the following steps by selecting the menu items on the left under shopping. Browse for products by product category, sort the list, and view details. Then, search for products by entering a partial name and/or price range, sort the resulting list of products, and view product details. In the product detail page add a product to your shopping cart. Add several more items to your cart. In the shopping cart page, remove items, change quantities, recalculate the total and subtotal, etc. Select a different shipping method and notice that the cost of shipping and the total cost are adjusted accordingly.

Next, login as an administrator whose task it is to manage the customers and review their orders and order details. This functionality is available from the menu items on the left under administration. Before entering the administrative area you must be authenticated. To login select the login menu item. Enter the suggested credentials: username: debbie and password: secret123.



Login

Login is required to access the Administration area. For demonstration purposes use these credentials: username: 'debbie', password: 'secret123'.

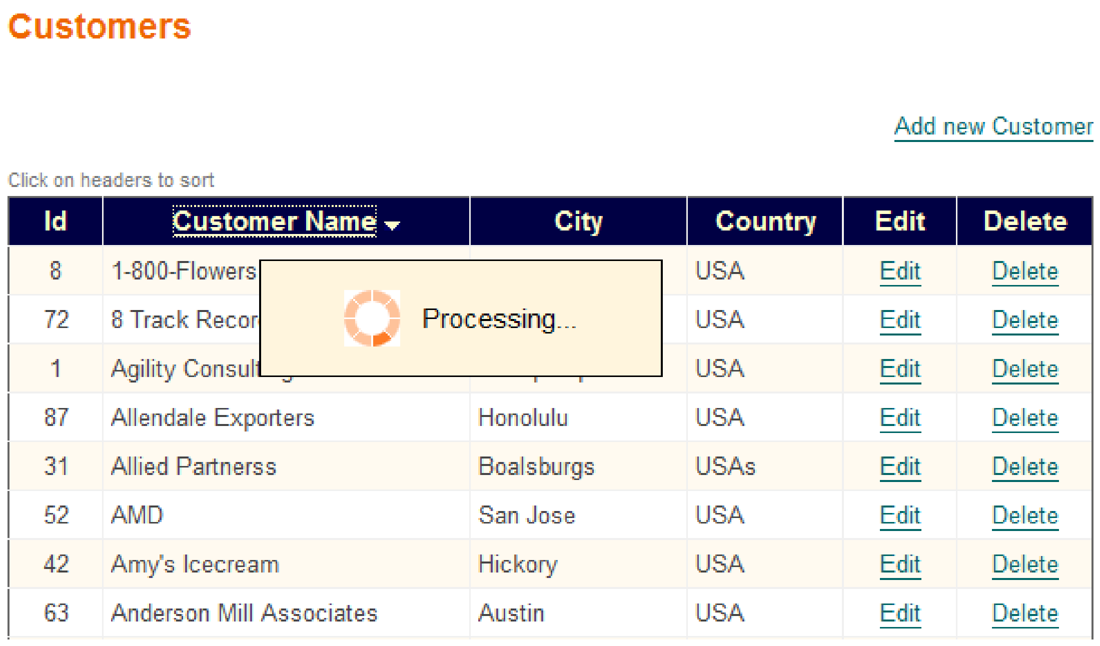
Open the customer page which allows three basic customer maintenance operations: Add, Edit, and Delete. Experiment with these options. A business rule in the application states that customers with orders cannot be deleted. To be able to delete we suggest that you first Add a new customer to the list. After that, sort by Customer Id (descending) and see that your new customer appears on top of the list. Select Edit and change some fields of the new customer and save your changes. Finally, from the customer list, Delete that customer from the database.

Orders can be viewed on the Orders page. It contains a customer list with order totals and last order date. Sort by these order-related fields by clicking on their headers. See who has the most orders placed (the Xio Zing Shoppe). Finally, view all orders for a customer and order details (line items) for one of the orders. All these pages follow the master-detail paradigm, that is, the list of customers is the master and their orders are the details. Each order, in turn, is a master itself because orders have order details (line items). Master-detail is a very common user interface pattern in applications.

A few more items we would like to point out. The app uses the SiteMapPath control (also called ‘bread crumb’ control) just below the header. It displays the current position in the site map (as defined in the Web.sitemap file). This control is used in both the Web Forms and the MVC application.

Notice that the selected menus are highlighted (red text with underscore) depicting the current page selection.

The WebForms application implements AJAX using the ASP.NET AJAX controls. The improvements to the user experience by adding AJAX are quite remarkable. For example, view the shopping cart with several items in it and make some changes. The page updates are flicker free. In fact, it is so smooth, that the change is hardly noticable (which may be a potential disadvantage). Another AJAX feature is that for slow running operations, a yellow ‘Processing’ box with animated image display over the able. See next image for this effect.


Customers

Add new Customer

Click on headers to sort

Id	Customer Name	City	Country	Edit	Delete
8	1-800-Flowers		USA	Edit	Delete
72	8 Track Recor		USA	Edit	Delete
1	Agility Consult		USA	Edit	Delete
87	Allendale Exporters	Honolulu	USA	Edit	Delete
31	Allied Partnerss	Boalsburgs	USAs	Edit	Delete
52	AMD	San Jose	USA	Edit	Delete
42	Amy's Icecream	Hickory	USA	Edit	Delete
63	Anderson Mill Associates	Austin	USA	Edit	Delete

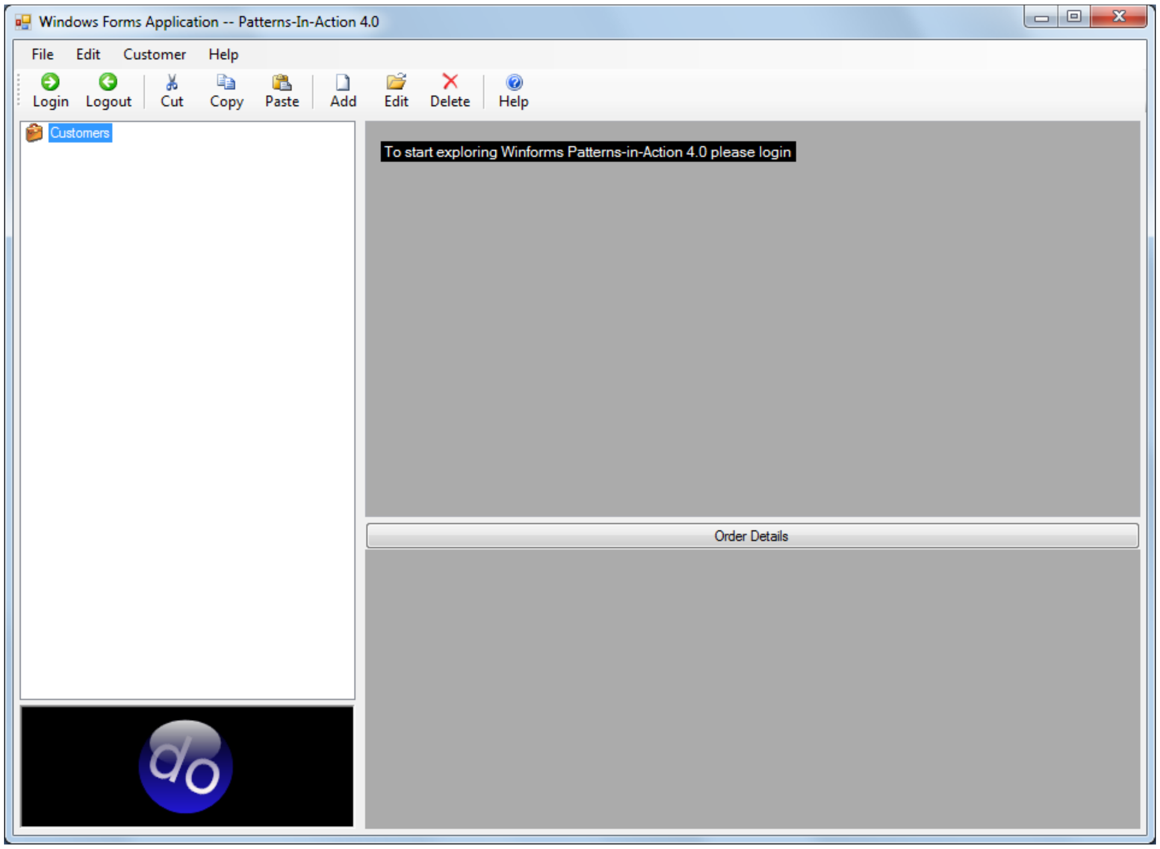
Processing...

ASP.NET AJAX helps display a ‘Processing’ message.

The MVC version of the application uses jQuery as their preferred AJAX/Javascript library. We did not add these AJAX effects to the MVC app. However, once you are familiar with jQuery these features are easy to add.

Windows Forms Application

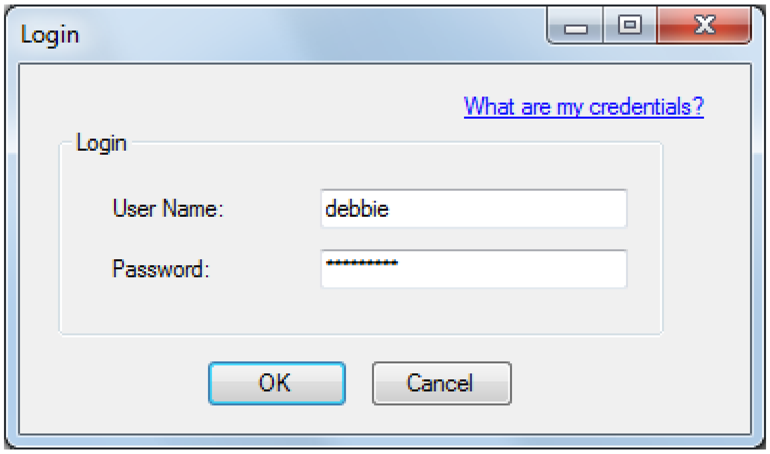
To start the Windows Forms Application set Windows Forms Application as the Startup Project and select Run (hit F5). This application is also a client to the common WCF service. The architecture decisions that went into this application are discussed later in this document.



To start exploring Winforms Patterns-in-Action 4.0 please login

Windows application (not logged in)

When starting the application you see the main form with three empty panes. Select the File->Login menu item or click the Login toolbar button. This opens the login dialog in which login credentials are entered.



User Name: debbie

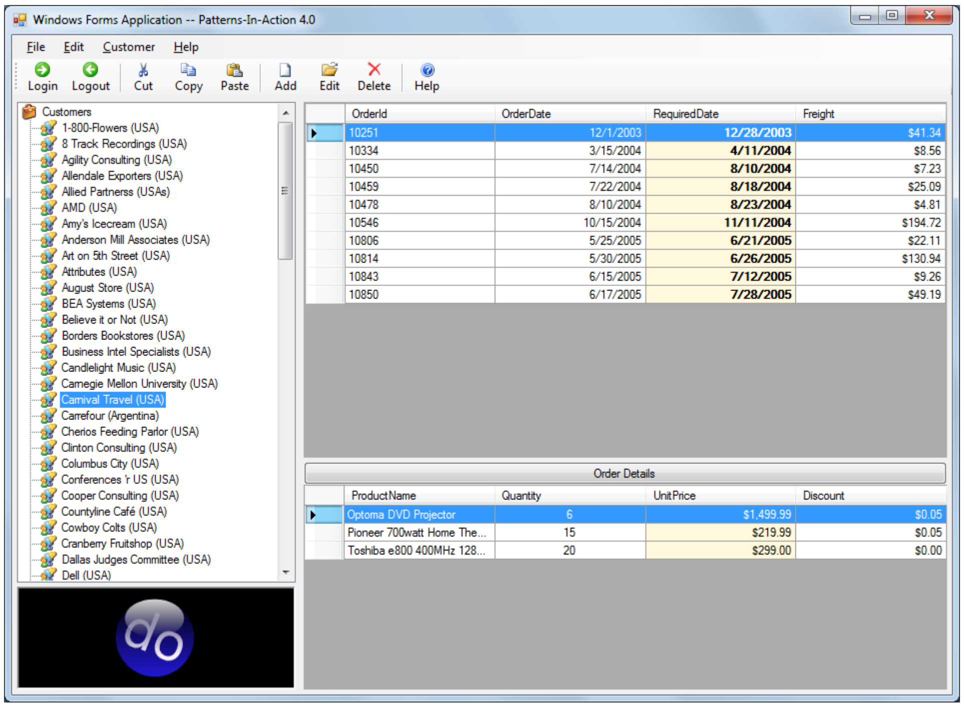
Password: **********

Login form

Patterns in Action 4.0 supports just one set of credentials (of course, in a real application you would have multiple users using the system). Credentials are: username: debbie and password: secret123

Text boxes in the login dialog are pre-populated with valid values and selecting OK will automatically log you in. If you forget your credentials click the "What are my credentials" link label. Note: If you have difficulty logging in (or receive 404 errors) double check that no file named *app_offline.htm* has been created in the Web Service folder. We found that this file is created following VB compile errors. If you see this file, simply delete it.

Once logged in, a list of customers will display in the tree view on the left. Remember that this application is a client to a Web Service and response times may not be 'snappy' (particularly when using MS Access). Click on one of the customers and the customer's orders and order detail data are retrieved and displayed. Highlight an order and order details will display instantly at the bottom pane (both orders and associated order details have been retrieved and are cached for the selected customer).



Customers
- 1-800-Flowers (USA)
- 8 Track Recordings (USA)
- Agility Consulting (USA)
- Allendale Exporters (USA)
- Allied Partnerss (USAs)
- AMD (USA)
- Amy's Icecream (USA)
- Anderson Mill Associates (USA)
- Art on 5th Street (USA)
- Attributes (USA)
- August Store (USA)
- BEA Systems (USA)
- Believe it or Not (USA)
- Borders Bookstores (USA)
- Business Intel Specialists (USA)
- Candlelight Music (USA)
- Carnegie Mellon University (USA)
- Carnival Travel (USA)
- Carrefour (Argentina)
- Cherios Feeding Parlor (USA)
- Clinton Consulting (USA)
- Columbus City (USA)
- Conferences 'r US (USA)
- Cooper Consulting (USA)
- Countyline Café (USA)
- Cowboy Colts (USA)
- Cranberry Fruitshop (USA)
- Dallas Judges Committee (USA)
- Dell (USA)

OrderId	OrderDate	RequiredDate	Freight
10251	12/1/2003	**12/28/2003**	$41.34
10334	3/15/2004	**4/11/2004**	$8.56
10450	7/14/2004	**8/10/2004**	$7.23
10459	7/22/2004	**8/18/2004**	$25.09
10478	8/10/2004	**8/23/2004**	$4.81
10546	10/15/2004	**11/11/2004**	$194.72
10806	5/25/2005	**6/21/2005**	$22.11
10814	5/30/2005	**6/26/2005**	$130.94
10843	6/15/2005	**7/12/2005**	$9.26
10850	6/17/2005	**7/28/2005**	$49.19

ProductName	Quantity	UnitPrice	Discount
Optoma DVD Projector	6	$1,499.99	$0.05
Pioneer 700watt Home The...	15	$219.99	$0.05
Toshiba e800 400MHz 128...	20	$299.00	$0.00

Windows forms app with customers, order, and order details

Once order and order details have been retrieved from the Web Service, they are cached on the client. Over time the performance of the application improves as more and more customer orders are being cached.

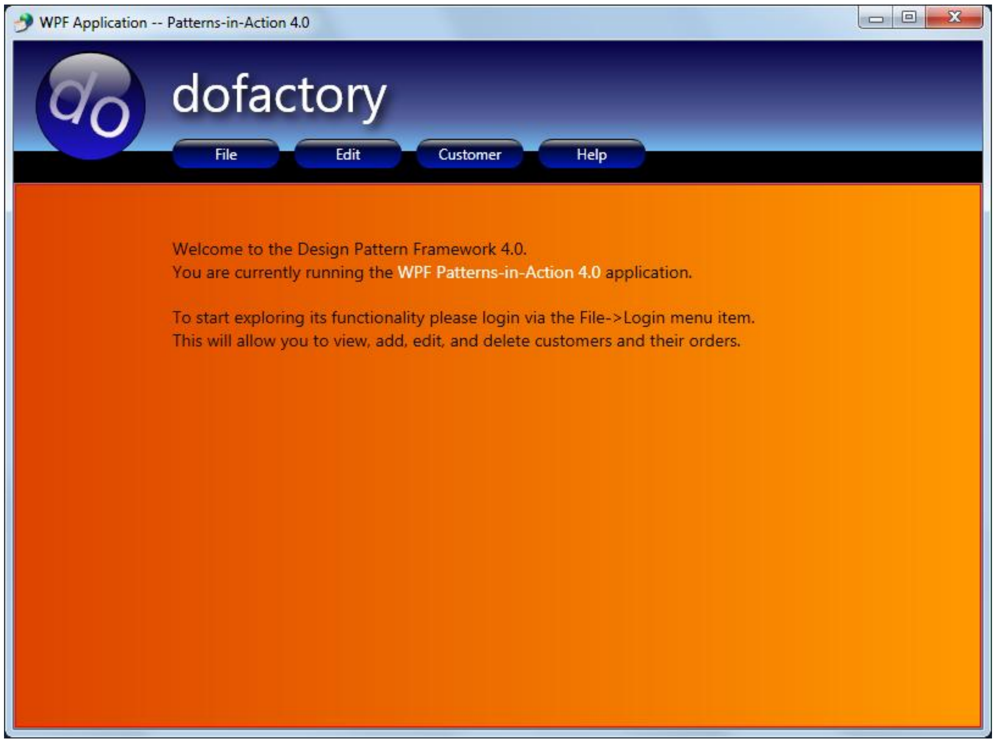
The customer maintenance functionality is the same as it is in the Web Application: you can Add, Edit, and Delete customer records. To reiterate: not only are they functionally the same, they also share the same Service Layer, Business layer, and Data layer. The details of this are explained later in this document.

Experiment with the customer maintenance options from the Customer menu or by clicking the Add, Edit, Delete toolbar icons. The functionality is straightforward and self-explanatory. Again, a business rule states that customers with orders cannot be deleted. To explore the delete functionality you will first have to create a new customer (note: new customers are added to the bottom of the customer tree), edit it if you want, and then delete it from the list of customers.

As a last step select Logout, which disconnects you from the Web Service.

WPF Application

To start the WPF Application set the WPF Application Project as the Startup Project and select Run (hit F5). Like all others before, this application also is a client to the WCF Service, which, when running from within Visual Studio 2010 is automatically launched. This service architecture is discussed later in this document.

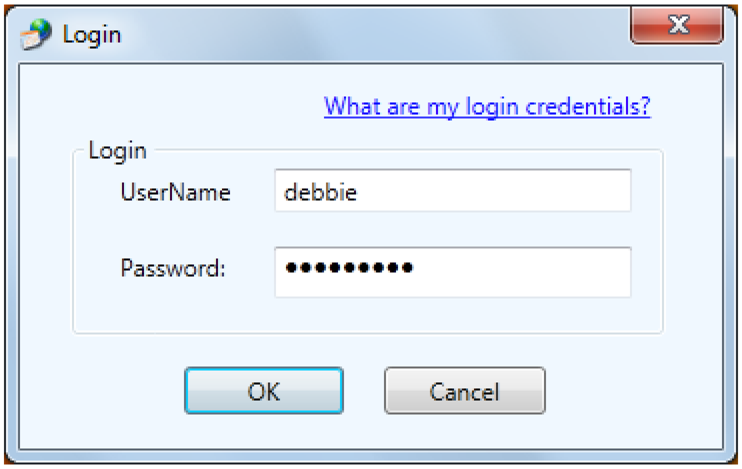


Welcome to the Design Pattern Framework 4.0.
You are currently running the WPF Patterns-in-Action 4.0 application.

To start exploring its functionality please login via the File->Login menu item.
This will allow you to view, add, edit, and delete customers and their orders.

WPF Application

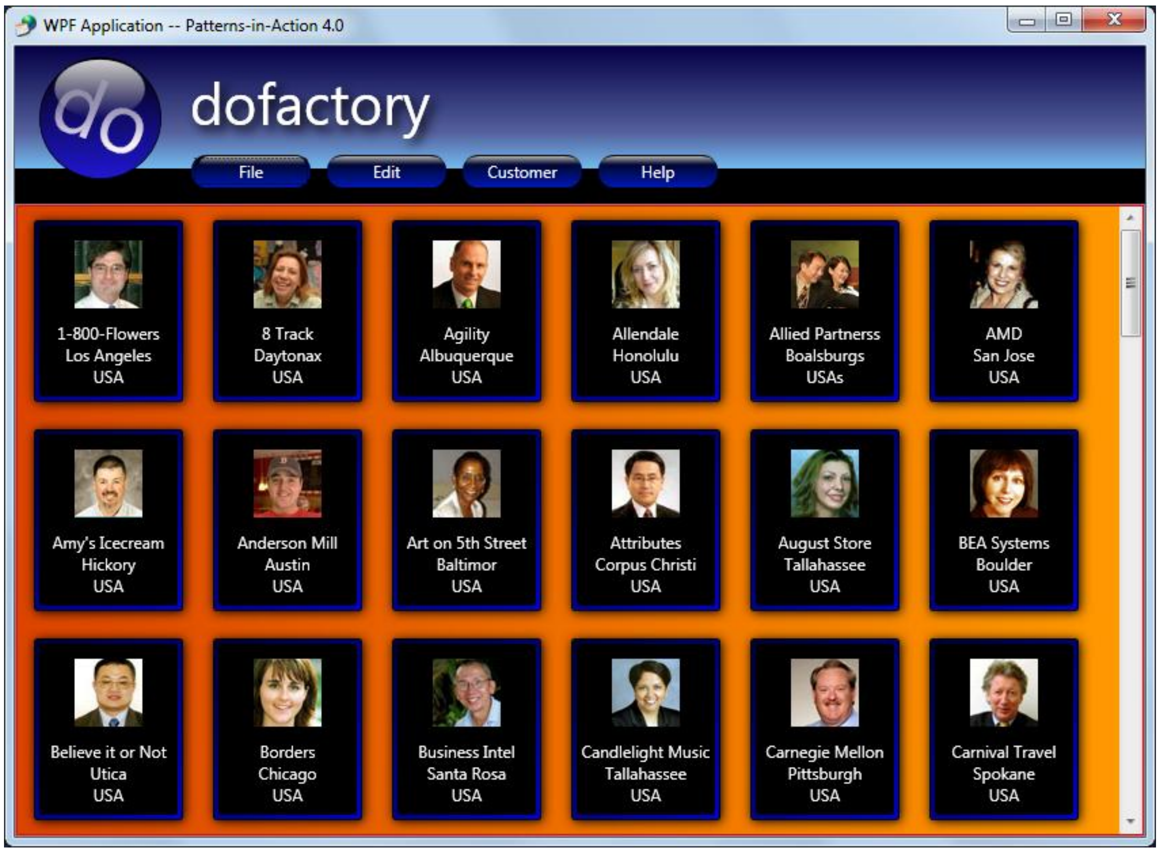
Upon startup you see the main form with four top level menus. Notice the menu button glow when the mouse moves over them. Select File->Login menu item or enter Ctrl-I. This opens the login dialog in which you enter login credentials.



WPF Login dialog

The text boxes in this login dialog are pre-populated with the appropriate values and selecting OK will log you in. If you forget your credentials click the "What are my login credentials" link label. The credentials are the same as all prior clients: username debbie and password secret123.

Once logged in, a series of customers, including their images, will display.

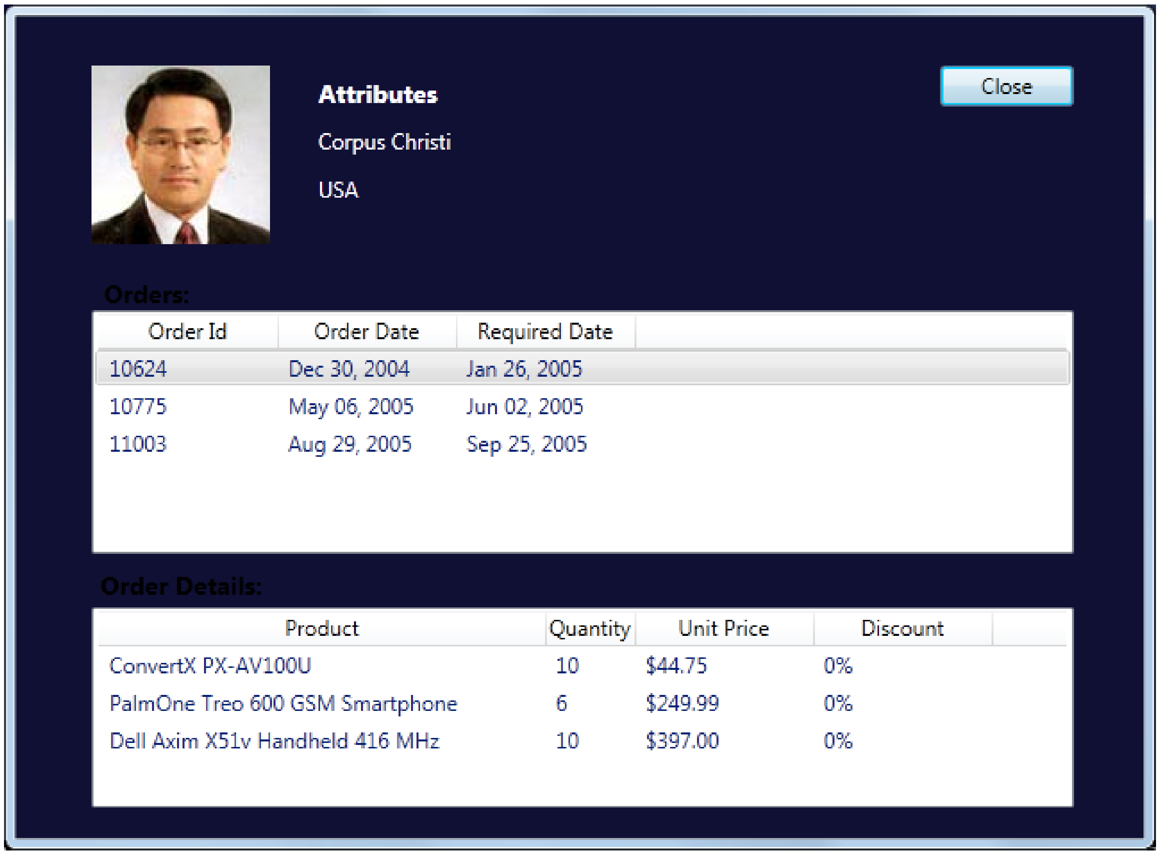

WPF Application -- Patterns-in-Action 4.0
dofactory
File
Edit
Customer
Help
1-800-Flowers Los Angeles USA
8 Track Daytonax USA
Agility Albuquerque USA
Allendale Honolulu USA
Allied Partnerss Boalsburgs USAs
AMD San Jose USA
Amy's Icecream Hickory USA
Anderson Mill Austin USA
Art on 5th Street Baltimor USA
Attributes Corpus Christi USA
August Store Tallahassee USA
BEA Systems Boulder USA
Believe it or Not Utica USA
Borders Chicago USA
Business Intel Santa Rosa USA
Candlelight Music Tallahassee USA
Carnegie Mellon Pittsburgh USA
Carnival Travel Spokane USA

WPF Application with customers retrieved.

New in release 4.0 is that the images are stored locally (rather than being retrieved from a central image service). This significantly improved the performance because the app had to retrieve many images at the same time (the image service became a bottleneck).

You can scroll up and down the customer list using the right-hand side scrollbar. Moving the cursor over the client boxes will trigger an animation that enlarges the box (using the new ‘easing’ animation options). When the cursor moves away it animates back to normal size. A selected customer box (simply click on it) is indicated by a blue backdrop.

Double click on one of the customers and the customer’s orders and order detail data are retrieved and displayed. See next image.



Attributes

Corpus Christi

USA

Orders:

Order Id	Order Date	Required Date
10624	Dec 30, 2004	Jan 26, 2005
10775	May 06, 2005	Jun 02, 2005
11003	Aug 29, 2005	Sep 25, 2005

Order Details:

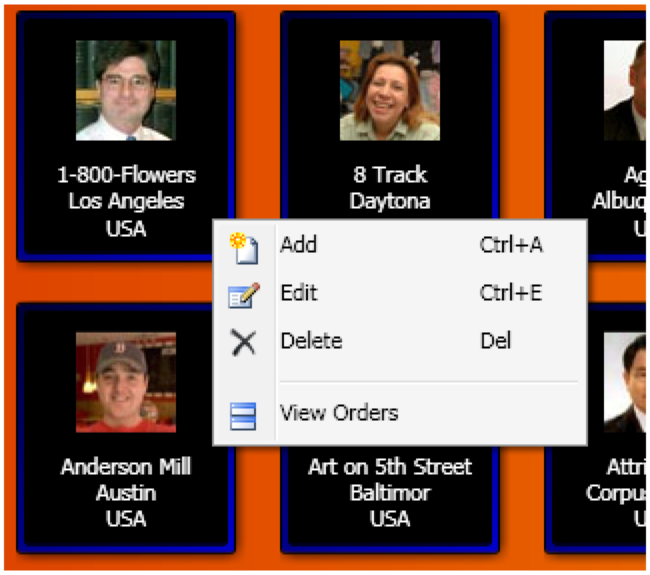
Product	Quantity	Unit Price	Discount
ConvertX PX-AV100U	10	$44.75	0%
PalmOne Treo 600 GSM Smartphone	6	$249.99	0%
Dell Axim X51v Handheld 416 MHz	10	$397.00	0%

WPF Application, customer details, orders, and order details.

Select an order and order details will display instantly at the order detail list (both orders and associated order details have been retrieved for the selected customer). Notice also that the customer image is larger than the ones displayed on the customer list boxes. Selecting the Close button will close this window.

The customer maintenance functionality is the same here as it is in the Web Application and Windows Forms Application; it allows you to Add, Edit, and Delete customer records. Again, not only are they functionally the same, they also share the same Service Layer, Business layer, and Data layer. The details of this are explained later in this document.

Experiment with the customer maintenance options from the Customer menu or by right clicking on a customer box and selecting the Add, Edit, Delete menu items. See image below.



Right clicking on customer box opens Context menu

The operations themselves are straightforward. One thing to remember is the business rule that states that customers with orders cannot be deleted. To explore the delete functionality it is best to first create a new customer (note: new customers are added to the bottom of the list of customer boxes), edit it if you want, and then delete it from the list of customers.

As a last step select Logout under the File menu button, which logs you out from the WCF Service. Then select File->Exit.

Application Architecture

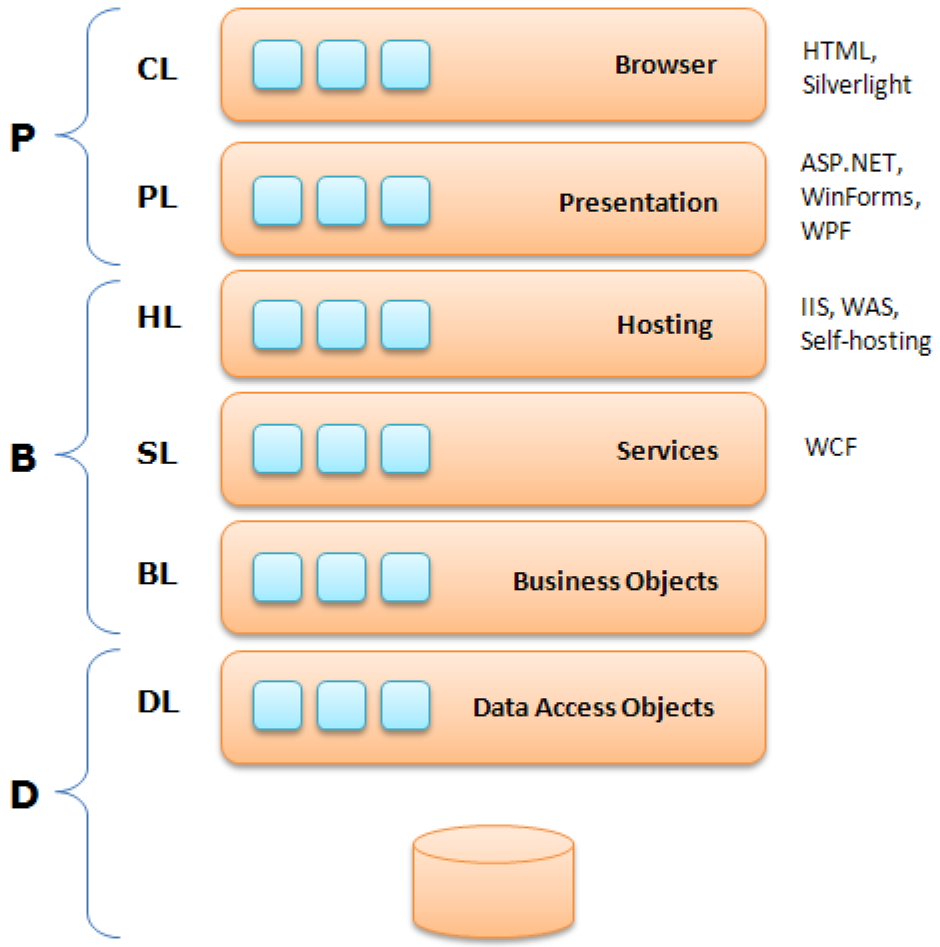
In this section we discuss general application architectural design decisions that go into building a robust solution. We will also highlight the specific architectural decisions that went into building the *Patterns in Action 4.0* reference application.

In general, you should have a good understanding of the functional (and non-functional) requirements before you can make decisions on the design and architecture of a new application. Functional requirements include the business processes that the application will perform. An example may be: the user can login or register to setup a new account; he/she should be able to search for a product from a catalog of products, etc. Functional requirements are often captured in use-case artifacts or user stories. Non- functional requirements (i.e. operational requirements) are harder to pin down – they determine the desired levels of scalability, availability, maintainability, and security of the application. Non-functional requirements provide the environmental details that are necessary to run the system effectively and efficiently.

Furthermore, designing business applications involves making decisions about the logical and physical architecture. A layered approach is a generally accepted best practice because it logically separates the major concerns of the application. The advantage is that the key components of the application will be loosely coupled (i.e. more flexible), more easily maintained, easy to enhance, and it provides the option to physically distribute the layers across multiple dedicated servers.

Layered Architecture

Patterns in Action 4.0 is built using a 3-tier architecture (i.e. horizontal tiers, which are different from the vertical tiers discussed earlier in this document). The next figure shows a cross-sectional view of the application layering.

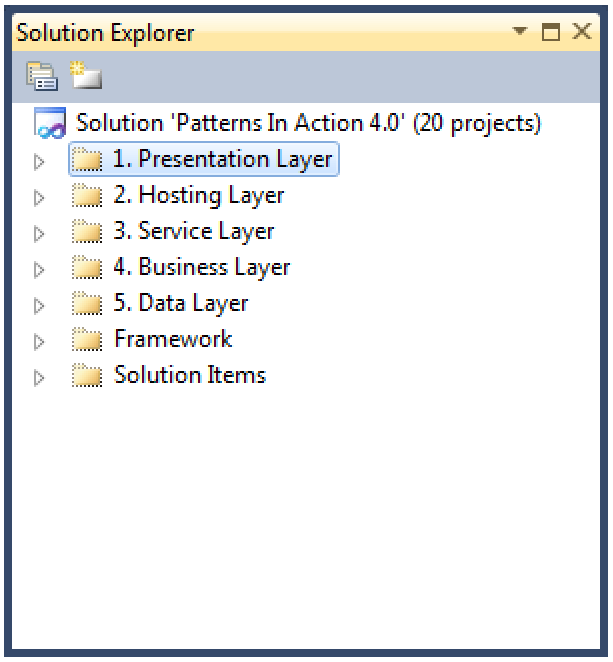


Cross Section of Tiers

First of all, you'll notice that this cross section depicts 6 layers and a database at the bottom. However, it is still referred to as a 3-tier architecture. On the left the letters **P, B**, and **D** represent these 3 tiers: **P**resentation, **B**usiness, and **D**ata tier respectively.

Some of these tiers consist of more than one layer which in the figure above is indicated by two letter acronyms: **CL** (**C**lient **L**ayer), **PL** (**P**resentation **L**ayer), **HL** (**H**osting **L**ayer), **SL** (**S**ervice **L**ayer), **BL** (**B**usiness **L**ayer), and **DL** (**D**ata Access **L**ayer) respectively. In this document, the word 'tier' is used more conceptually, as opposed to 'layer' which represents a physical structure (just so that you know, other publications sometimes use the words 'tier' and 'layer' to mean the same thing, whereas others reverse our usage, that is, tier as physical and layer as conceptual).

When viewing he Solution Explorer you see the 5 layers represented by numbered sub-folders (see image below). The difference between the cross section above and the Solution folders is that CL (Client Layer) is not present in the latter. However, if we had included Silverlight in this solution it would probably have resided in a Client Layer folder (possibly like 0.Client Layer).



Solution Explorer with Layers

Some architects refer to the architecture depicted in the cross-section as 4-tier, 5-tier, 6-tier, and sometimes even 7-tier. In reality they are talking about the same basic 3-tier model, except that they choose to include sub-systems that are required to run the applications and declare these to be another tier.

Some consider the client tier (browser) to be the 1st tier and the database to be the last tier. In fact, they have a point, particularly when the browser contains application logic (using Javascript, Ajax, Silverlight) and the database contains business logic (using triggers and stored procedures). However, we will stick with common terminology and continue referring to our architecture as 3-tier, but realize that more than 3 layers or subsystems are involved.

Analyzing the layers

Let’s examine the cross-section from the bottom up. At the very bottom we have the database. Next we find the data access layer (**DL**) who’s main responsibility is to provide access to the database and map database tables to business objects and vice versa. The **DL** is the only place in the application where data access occurs, including database connections and executing dynamic SQL and/or stored procedures in the database. Data access object (DAO) interfaces represent the data access functionality required to run the application. These interfaces are implemented by database specific data access classes (one set for MS Access, one for SQL Server, one for Oracle, one for LINQ-to-SQL, and one for Entity Framework).

This data access model makes it easy to switch to a different database and/or different data access technology just by changing a web.config configuration setting. To summarize: *Patterns in Action 4.0* supports MS Access, SQL Express and SQL Server databases. Data access technologies supported include ADO.NET, LINQ-to-SQL, and Entity Framework. Membership however, always uses a local SQL Express database in this application (but can be changed easily to SQL Server).

The business layer (**BL**) contains business objects. Business objects represent the domain logic that is relevant to the application, that is, the data and the business rules that are important to the business. Each business object has the ability to validate its data using a simple business rule engine that is built into the base class for each business object.

The service layer (**SL**) implements a single point of entry (the Façade pattern) throught which all communication with the layers below must occur. The Façade is the entry point into the Business layer and exposes a very simple, course-grained API. The Façade is a busy place because it manages business objects and coordinates several common operational tasks, including data validation, user authorization, and transaction management.

The **SL** also coordinates the interaction between business objects and data access objects by saving and retrieving business objects using DAOs (data access objects) to and from the database. Throughout the service layer you will find WCF features, such as, DataContract, DataMember, and ServiceContract attributes. Because the SL is a single point of entry for the application, we call this the Application Façade Design Pattern. It supports a simple, but very powerful API that can be exposed on the cloud and be accessed by any kind of client technology. It is the only entry point into the application and its data.

The hosting layer (**HL**) exposes the service layer (WCF Services) to the outside world. Setting up an HL is primarily a configuration exercise in which the hosting environment is established, such as IIS, Windows Services, WAS, or others. The choice of selecting the optimal hosting model and configuration is largely driven by performance requirements, the operating platform, and the clients (i.e. the **PL**), and the location of the clients (Internet, Intranet, etc) that are going to consume the services. When building an architecture around WCF services, the WCF hosting layer is the perfect place to hold singleton resources (that is, resources of which only one instance can exist), such as, databases, images, documents, etc.

The interface (or contract) between the service layer (**SL**) and the service consumer (**PL**) is built around a messaging model in which messages contain Data Transfer Objects. Data Transfer Objects is an Enterprise design pattern in which you have objects that contain business data only and no behavior (methods or properties).

Patterns in Action 4.0 demonstrates the use of four different **PL** technologies: ASP.NET Web Forms, ASP.NET MVC, Windows Forms, and WPF. As mentioned before, each calls into the exact same service layer.

The client layer (**CL**) is relevant only to Web Applications and respresents the browser which can render HTML and/or Silverlight / Flash, etc.

The different UI platforms (Web, Windows, WPF) are all very different with respect to deployment, user experience, coding methods, etc. However, with the 3-tier architecture we have clearly demonstrated the ability to build a single foundation with service layer, business layer, and data layer that is accessible by any kind of UI technology with several different hosting options (offered by WCF hosting). Whatever UI platform or hosting options, no code changes are required to the bottom 4 layers.

WCF Hosting and Deployment

Important: This section is important for those who, after studying the *Patterns in Action 4.0* reference architecture, have decided to build a similar 3-tier .NET pattern model.

The design and architecture of *Patterns in Action 4.0*, with the single-point-of-entry service layer (Application Façade pattern), is a powerful model that you can use in all your own .NET projects. However, when it comes to deployment you may want to configure your WCF Hosting in a different way from *Patterns in Action 4.0*. As mentioned earlier, the performance of the application may be somewhat sluggish which is related to the way we configured WCF Hosting.

Our goal for *Patterns in Action 4.0* is to demonstrate the ability to support multiple UI platforms ‘out-of-the-box’ without any special hosting and/or installation requirements, such as, IIS 7, Windows Services, or self-hosting. Frankly, we would have preferred to use IIS 7, including WAS (Windows Activation Services), and we would be able to demonstrate great performance. However, IIS 7 is not (yet) readily available to most.NET developers and therefore we are hosting using ASP.NET’s development server over HTTP.

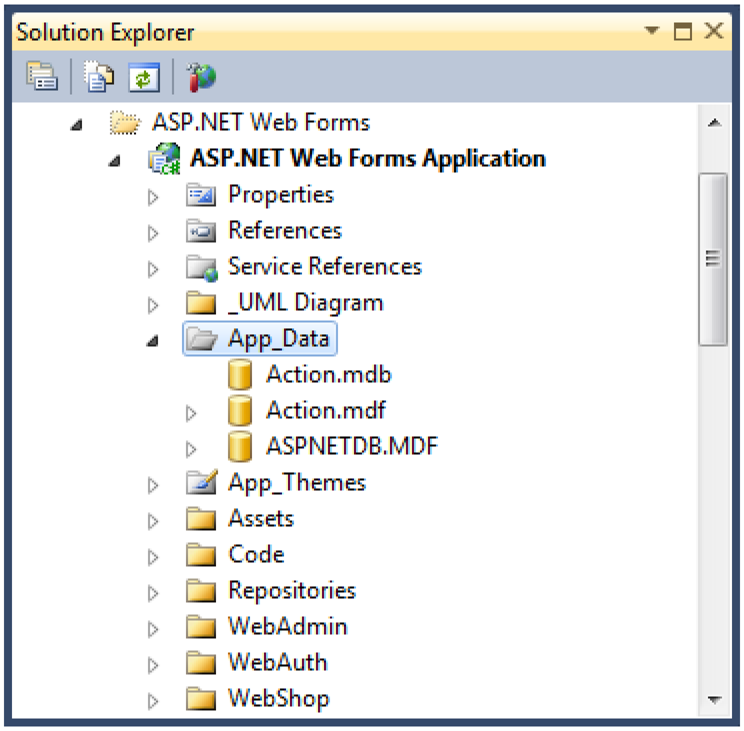
The optimal WCF Hosting and deployment model for any application depends on your particular situation and requirements. The first question to ask is: are there any external consumers of the service API in addition to the main application you are developing? If you have multiple consumers each one will need to be evaluated with respect to optimal hosting configuration. External applications that access your Web services over the Internet probably need IIS over HTTP.

Internal applications (i.e Intranet apps) may use any one of the hosting environments available: IIS, Windows Service, or self hosting. In addition you should determine the optimal 'endpoint' configuration, which includes decisions on your *ABC*: Address, Binding, & Contract. The WCF is a large platform and the details about endpoints is outside the scope of this document. Fortunately, there are some very good reference books are available.

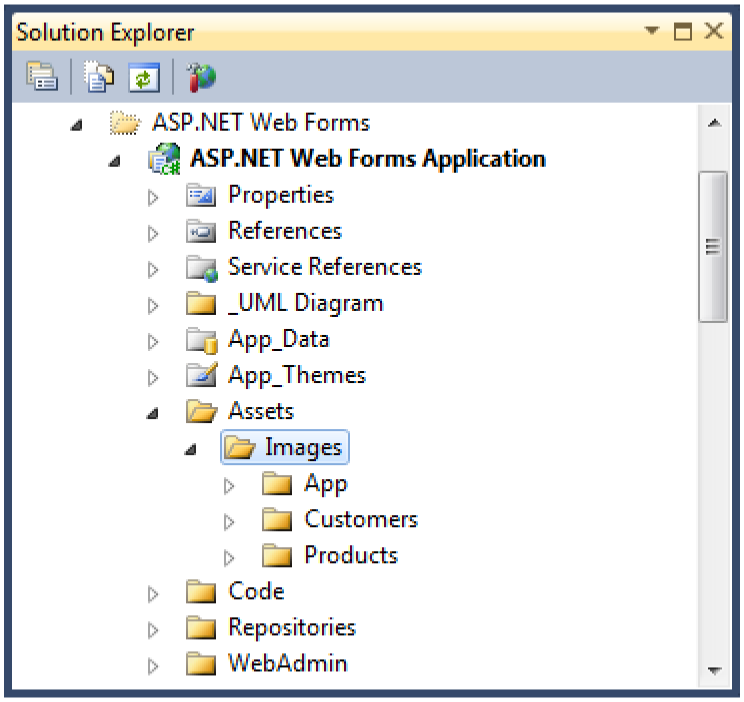
Building standalone ASP.NET applications

If you are developing an ASP.NET application (Web Forms or MVC) and there are no external consumers of the service API (other than your application), you should consider bypassing the WCF hosting layer alltogether. Simply reference the service layer assemblies directly from your UI layer, just like you would any other assembly. Because many .NET projects involve building standalone ASP.NET applications, we are including the steps necessary to skip WCF hosting and reference the service layer assemblies directly. Here are the steps for ASP.NET Web Forms (but MVC would be similar):

1) Copy the App_Data folder from project *Host.WCF.ActionServer* to project *ASP.NET Web Application*. You will only be able to copy these database files when all databases are properly closed. The application looks like this:



2) Copy the \Images\Customers and \Images\Products folders from project *Host.WCF.ImageServer* to the \Images folder in project *ASP.NET Web Application*. The application looks like the image below.

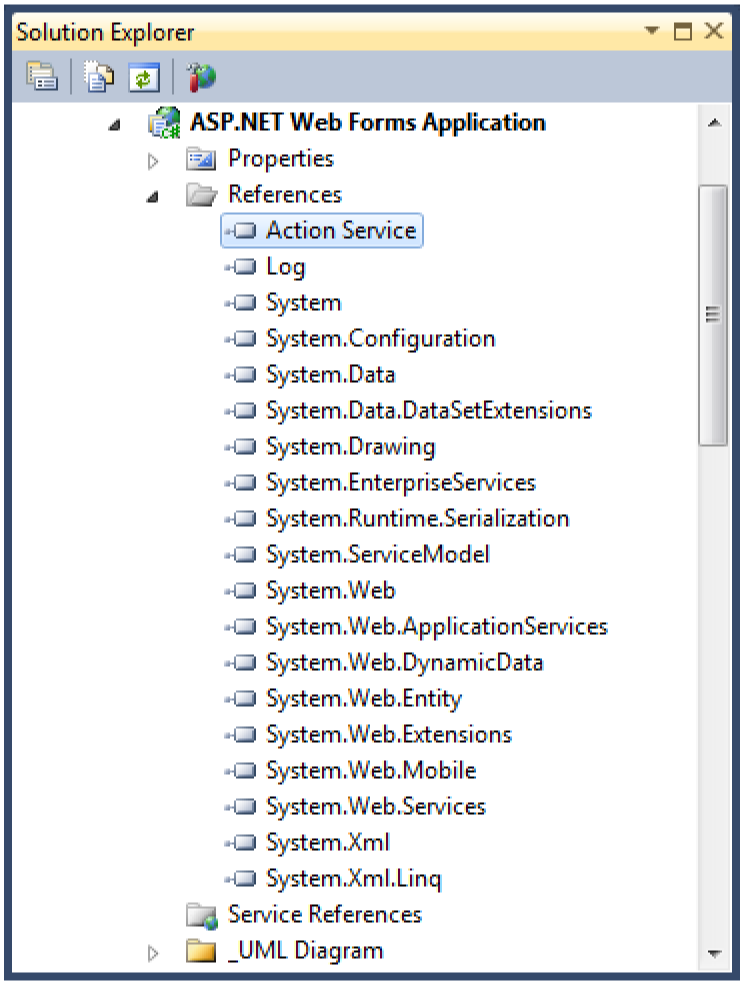


3) Copy the *<appsettings>, <connectionStrings>,* and *</membership>* sections from the web.config file in project *WCF.ActionServer* to the web.config file in project *ASP.NET Web Forms Application*. Note: do not immediately overwrite the original <appsettings> section, because you need to copy the following line from the old into the new <appsettings> section. Without the ClientTag value, the system will *not* work.

```
<!-- Client tag: identifier or Service communication -->
<add key="ClientTag" value="ABC123"/>
```

Finally, the *<system.ServiceModel>* section can be removed.

4) Add a Reference to the assembly of project Action Service. Remove the two Service References. The project now looks like the following image:



5) Replace all these `using` statements in all files:

```
using ASPNETWebApplication.ActionServiceReference;
```

with

```
using ActionService;
```

Include the appropriate namespaces in all Repository classes and also several code- behinds, for example:

```
using ActionService.Messages;
using ActionService.MessageBase;
using ActionService.Criteria;
using ActionService.DataTransferObjects;
```

In page Product.aspx also replace

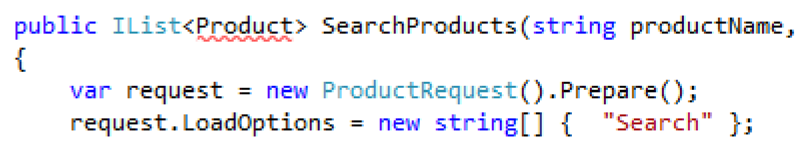
```
<%@ Import
       Namespace="ASPNETWebApplication.ActionServiceReference" %>
```

with

```
<%@ Import Namespace="ActionService.DataTransferObjects" %>
```

and change Category to CategoryDto in the middle of the page.

Replace all business object names, such as, Customer, Product, ShoppingCartItem with the DTO versions, as in, CustomerDto, ProductDto, ShoppingCartItemDto, etc. (in this setup, WCF is not available anymore to rename these to the non-dto names). The editor will help out as the original names are underlined in red (see below). Once you append Dto, this will dissappear.



```
public IList<Product> SearchProducts(string productName,
{
    var request = new ProductRequest().Prepare();
    request.LoadOptions = new string[] {  "Search" };
```

Then in the Repository base class, change the Client property like so:

```
/// <summary>
/// Lazy loads ActionServiceClient and stores it in Session object.
/// </summary>
protected ActionService.ServiceImplementations.ActionService Client
{
   get
   {
      if (HttpContext.Current.Session["ActionServiceClient"] == null)
          HttpContext.Current.Session["ActionServiceClient"] =
          new ActionService.ServiceImplementations.ActionService();

      return HttpContext.Current.Session["ActionServiceClient"] as
           ActionService.ServiceImplementations.ActionService;
   }
}
```

6) Finally, change how customer images and product images are referenced in the pages: For example, in the Customer.aspx code-behind change:

From:

```
// Set image
ImageCustomer.ImageUrl =
      imageService + "GetCustomerImageLarge/" +  CustomerId
```

To

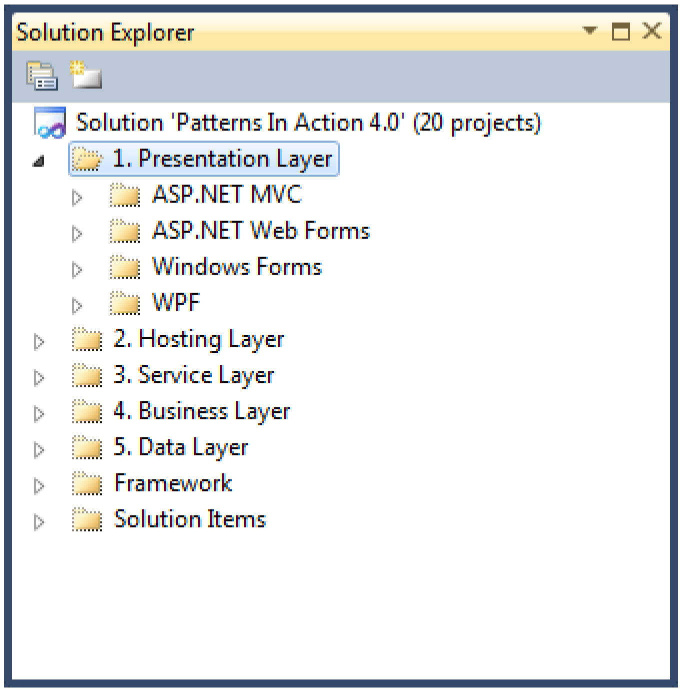
```
// Set image
ImageCustomer.ImageUrl =
       "~/Assets/Images/Customers/Large/" + CustomerId + ".jpg";
```

Make a similar ImageUrl change in the Product.aspx code-behind.

You are ready to go! Start the ASP.NET application and confirm that you are running without the WCF hosting layer. Thanks to the 3-tier service oriented architecture, we needed to make only few changes to get to a completely different deployment model.

The .NET Solution and Projects

This section explores the *Patterns in Action 4.0* Solution and its 20 Projects. Open the application in Visual Studio 2010. Collapse all projects, all folders, and then open the first level folders without seeing any projects. Don't open the folders just yet. Your Solution Explorer should look like the image below:



.NET solutions and projects

The Solution Explorer shows there are 20 projects. They are spread over 5 layers and a folder named Framework that contains several utility projects. The layers are numbered so that they display in a logical top-to-bottom order.

The Presentation Layer contains 4 application types: an ASP.NET MVC application, an ASP.NET Web Forms application, a Windows Forms application, and a Windows WPF application.

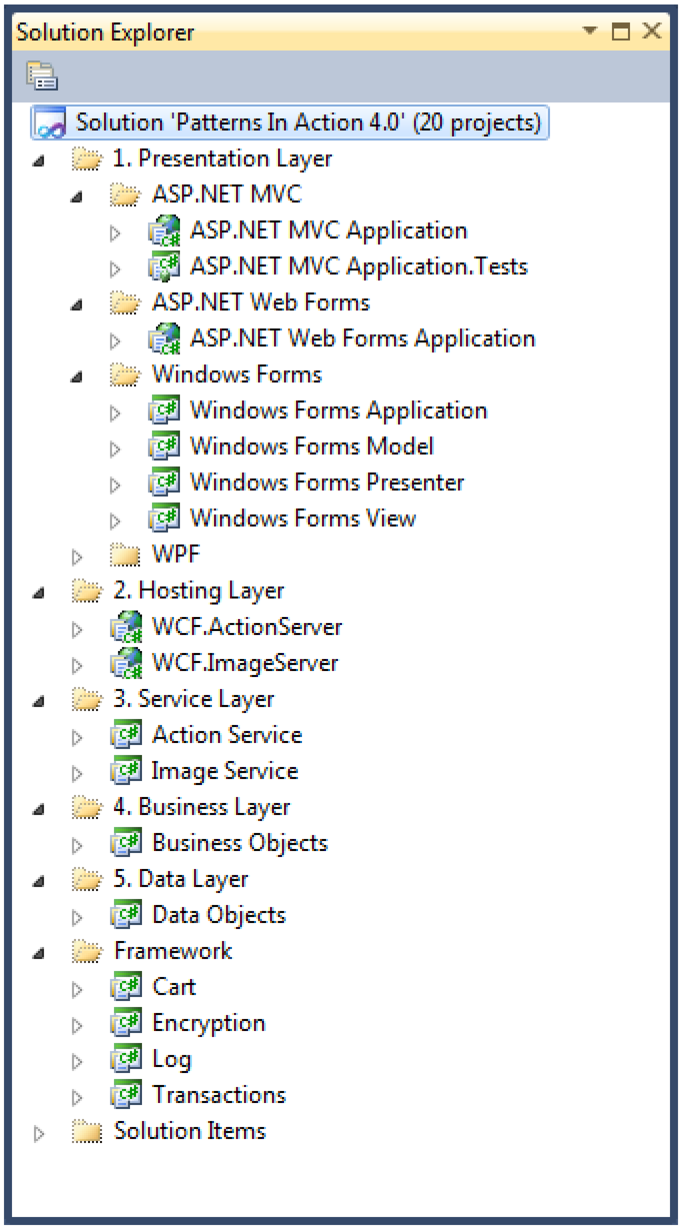
All 4 applications consume the same application services in the lower levels: that is, layers 2, 3, 4, and 5. The Service Layer represents a single point of entry into the application services, meaning that to get to the Business Layer and Data Layer you must go through the Service Layer. The services in the Service Layer are exposed to the outside world via the Hosting Layer.

It is the Service Layer that invokes the Business and Data Layers. The Data Layer is the only place with access to the database. The Framework folder contains a few supporting class libraries, including, shopping cart, logging, and encryption.

Finally, the folder named \Solution Items contains several project artifacts, including a copyright statement, a class reference help file, and two sql files: action.sql and adonetdb.sql – these have the scripts to create and populate the databases used in this solution.

20 Projects

To view the projects, expand the folders. A total of 20 projects will be exposed.


Solution Explorer

Solution 'Patterns In Action 4.0' (20 projects)
- 1. Presentation Layer
  - ASP.NET MVC
    - ASP.NET MVC Application
    - ASP.NET MVC Application.Tests
  - ASP.NET Web Forms
    - ASP.NET Web Forms Application
  - Windows Forms
    - Windows Forms Application
    - Windows Forms Model
    - Windows Forms Presenter
    - Windows Forms View
  - WPF
- 2. Hosting Layer
  - WCF.ActionServer
  - WCF.ImageServer
- 3. Service Layer
  - Action Service
  - Image Service
- 4. Business Layer
  - Business Objects
- 5. Data Layer
  - Data Objects
- Framework
  - Cart
  - Encryption
  - Log
  - Transactions
- Solution Items

The 20 projects in *Patterns in Action 4.0*

Note: Two projects (Controls and ViewState) that were originally listed under Framework in *Design Pattern Framework 3.5* have been moved to the ASP.NET Web Forms application (which is the only place where they were used anyhow).

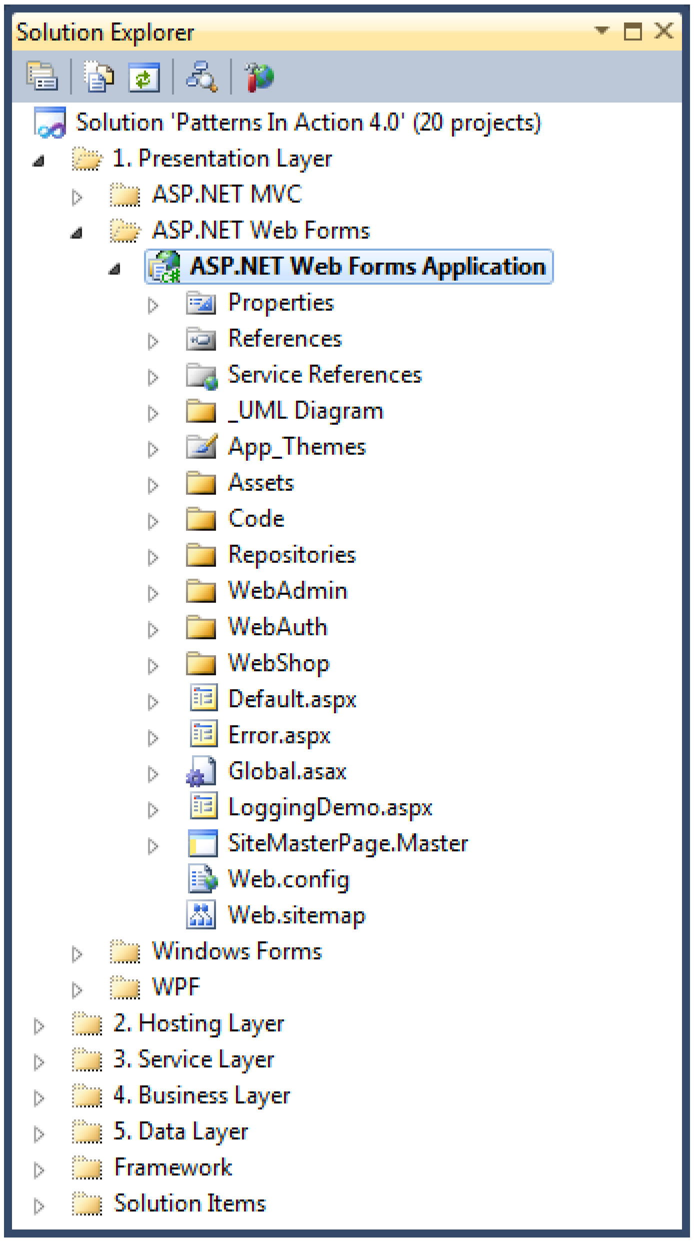
Here is a brief summary of each project.

Project	**Summary**
ASP.NET MVC Application	The MVC web application
ASP.NET MVC App. Tests	The MVC unit testing application
ASP.NET Web Forms App.	The Web Forms web application
Windows Forms Application	The forms of the windows forms application
Windows Forms Model	The model in the Model-View-Presenter pattern
Windows Forms Presenter	The presenter in the Model-View-Presenter pattern
Windows Forms View	The view in the Model-View-Presenter pattern
WPF Application	The WPF forms and views in the MVVM pattern
WPF Model	The model in the Model-ViewModel-Model pattern
WPF ViewModel	The viewmodel in the Model-ViewModel-Model pattern
WCF ActionServer	Host for the Patterns in Action 4.0 application service
WCF ImageServer	Host for the Image service
Action Service	The Patterns in Action 4.0 application service
Image Service	The customer image service
Business Objects	Business objects with business rules
Data Objects	Data Access objects (ADO.NET and LINQ-to-SQL)
Cart	An e-commerce shopping cart
Encryption	A security library with encryption/decryption functionality
Log	An error logging system
Transactions	Manages database transactions for different databases

Next, we will examine each project in more detail:

ASP.NET Web Forms Application

Expand the ASP.NET Web Forms Application project until you see the view below. The ASP.NET Web Application has been set as the default Startup Project (in bold).



```
Solution 'Patterns In Action 4.0' (20 projects)
  1. Presentation Layer
    ASP.NET MVC
    ASP.NET Web Forms
      ASP.NET Web Forms Application
        Properties
        References
        Service References
        _UML Diagram
        App_Themes
        Assets
        Code
        Repositories
        WebAdmin
        WebAuth
        WebShop
        Default.aspx
        Error.aspx
        Global.asax
        LoggingDemo.aspx
        SiteMasterPage.Master
        Web.config
        Web.sitemap
    Windows Forms
    WPF
  2. Hosting Layer
  3. Service Layer
  4. Business Layer
  5. Data Layer
  Framework
  Solution Items
```

The ASP.NET Web Forms Application is an e-commerce web application. It communicates exclusively with two WCF Web services (referenced under the Service

References folder). There is no database under the \App-Data folder and neither will you find a folder with customer or product images. Application data and images are served via the aforementioned WCF web services.

The Default page, Login and Logout pages, Global.asax, and Master page are located in the project root. The shopping module and related pages are located in the \WebShop folder. This is where users search and shop for products. The administration module and related pages reside in the \WebAdmin folder; this is where the administrator manages the web site. Access to the web pages in this folder is restricted and requires a login. Security is handled by Microsoft's Membership services.

A folder named Repositories contains several repository objects that communicate with the WCF services. This method of accessing data services is called the *Repository Pattern* which will be discussed later in this document. A RepositoryBase base class provides basic plumbing for all repositories and is the ancestor to all Repository classes. A new RequestHelper class facilitates the construction of RequestMessages (each Request messages contain a set of 3 fields that need to be populated for every request)..

A folder named \Code contains several utility classes. One is a file named PageBase.cs, which is the base class to all pages in *Patterns in Action 4.0*. PageBase supports functionality that is shared throughout the application and includes 1) page render timings, 2) gridview sorting, 3) shopping cart access, 4) viewstate management, and 5) javascript registration. Another class is UrlMaker, which helps you build consistent Urls that are used throughout the application for routing purposes.

Under the \Code folder are two more folders: \ViewState and \Controls which in prior versions, were separate libraries. ViewState manages the ViewState on the server side (it demonstrates the Provider design pattern discussed in more detail at the end of this section). The \Controls folder holds the custom menu control. This control is placed on the ASP.NET Master page and is therefore rendered on every web page. It demonstrates the use of the Composite design pattern, which builds tree-like data structures. The menu control is a two-level tree hierarchy.

Folder \App_Themes contains themes that control the overall appearance and look and feel of an application. *Patterns in Action 4.0* has a new theme, named 'DoFactorySky'. DoFactorySky.css is a style sheet which assists in formatting of the HTML elements. DoFactorySky.skin contains skin definitions to control the appearance of several of the ASP.NET controls.

Several application images are located in the \Assets\Images\App\ folder.

A lesser known feature that may be of interest to ASP.NET developers is the SiteMapPath. A file named *Web.sitemap* defines the hierarchical structure of the web site and is the data source for the SiteMapPath control displayed along the top of each page (i.e. breadcrumbs).

When using Master pages, .NET developers usually require a) access a control on the Master page from the content pages, and b) access an item on the content page from the Master page. *Patterns in Action 4.0* does both; the code behind page of the Master page demonstrates how this bi-directional access is handled.

The latest version of ASP.NET offers a couple of new SEO tools (SEO = Search Engine Optimization): they are *Meta tags* and *Routing*. The built-in MetaKeywords and MetaDescription properties on the Page class are used to add meta-tag data to each page (which is picked up by search engines), which is a nice convenience. Even better is the support for Routing. Routing lets you configure an application to accept request URLs that do not map to physical files. This allows you to build search-engine-friendly and meaningful URLs. For example, instead of this: www.company.com/users/user.aspx?cid=3340d99ss you can now have this: www.company.com/users/john-travolta. The routing system is fantastic, but remember that *designing* a consistent set of SEO friendly URLs can be tricky. This application demonstrates one way of designing a consistent and predictable set of URLs. Two places to explore are global.asax (where the routes are registered) and UrlMaker.cs (where routes are built – preventing the use of *magic strings* throughout the application).

The ViewState in ASP.NET is a feature that makes keeping track of page state very easy. The downside is that it may add a considerable amount of data to every page (stored in a hidden field named "__VIEWSTATE"). Particularly when working with list-type controls, such as, GridView and ListView, the increase in page size can be very significant. Select View->Source on your browser and you can actually see the data that travels with these pages.

The ViewState set of classes in this project offers an alternative by keeping the Viewstate data on the server rather than sending it over to the client with every page. So, where then is this ViewState data kept on the server? Alternatives include: in Cache, in Session, or in a globally accessible HashTable data structure. All three providers are implemented in *Patterns in Action 4.0* using Microsoft's Provider design pattern.

ViewState and the Provider design pattern

The built-in Provider design pattern is a very comprehensive pattern. It offers an infrastructure in which several elements need to be implemented. We'll step through the different elements and the classes that implement them.

A 'provider' is a pluggable and configurable component that extends or replaces current system functionality. As an example, ASP.NET offers a built-in Session management system that uses the provider design pattern. It is configurable in web.config where you can setup a different provider (for example, store Session data in SQL Server). Alternatively, you can write your own provider and customize the way Session data is stored. The provider model offers considerable flexibility.

Providers require their own configuration section in web.config (or app.config). This custom section must contain a list of registered providers, one of which is marked as the default. Below is our web.config, which demonstrates how the ViewState provider is configured.

```
 <!--  Declare viewstateService as a valid section in this file
       Note: this section must be first element under <configuration>
 -->
 <configSections>
  <sectionGroup name="system.web">
   <section name="viewstateService"
   type="DoFactory.Framework.ViewState.ViewStateProviderServiceSection,
         ViewState"
         allowDefinition="MachineToApplication"
         restartOnExternalChanges="true"/>
  </sectionGroup>
 </configSections>
...
<system.web>
  <!--  Custom viewstate provider service  -->
   <viewstateService defaultProvider="ViewStateProviderGlobal">
    <providers>
      <add name="ViewStateProviderCache"
        type="DoFactory.Framework.ViewState.ViewStateProviderCache"/>
      <add name="ViewStateProviderGlobal"
        type="DoFactory.Framework.ViewState.ViewStateProviderGlobal"/>
      <add name="ViewStateProviderSession"
        type="DoFactory.Framework.ViewState.ViewStateProviderSession"/>
    </providers>
   </viewstateService>

  ...
```

The top half demonstrates how a new <viewstateService> section is defined that will be referenced later in the configuration file. The bottom half shows how the three viewstate providers are registered. The ViewStateProviderGlobal is set as the default.

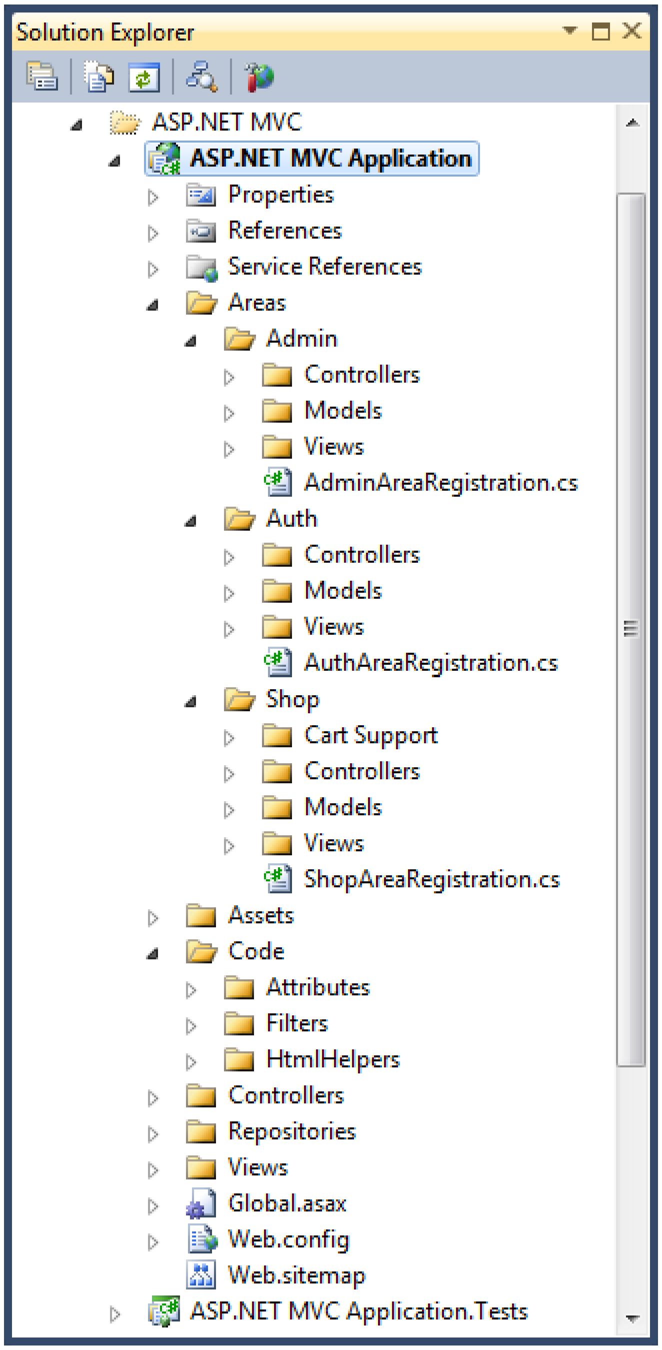
Folder \ProviderBase contains the classes that perform the 'plumbing' for the viewstate provider. Abstract class ViewStateProviderBase declares abstract methods that need to be implemented by the ViewStateProvider instances; they are SavePageState and LoadPageState. ViewStateProviderCollection is a collection of ViewStateProviders that are read into memory from the web.config. A static class ViewStateProviderService ensures that the viewstate providers are loaded and that the default provider is set correctly. Finally, ViewStateProviderServiceSection represents the custom section in the web.config file.

The three viewstate providers are ViewStateProviderCache, ViewStateProviderSession, and ViewStateProviderGlobal. They all derive from ViewStateProviderBase and implement (override) the two abstract methods SavePageState and LoadPageState. The Global provider has a helper class named GlobalViewStateSingleton.

Performance improvements by keeping ViewState data on the server can be rather significant. You could consider using this in your own applications. However, we need to point out that ViewState replacement is a complex topic and different scenarios and page sequences need to be thoroughly tested. Before you take this route, please know that the code in *Patterns in Action 4.0* is written for educational purposes only and may or may not work under all scenarios and configurations.

ASP.NET MVC Application

Expand the ASP.NET MVC folder, select the ASP.NET MVC Application and expand the \Areas folder and the \Code folder. This gives you an overview of the MVC project structure. Your explorer should look like the image below:



```
ASP.NET MVC
  ASP.NET MVC Application
    Properties
    References
    Service References
    Areas
      Admin
        Controllers
        Models
        Views
        AdminAreaRegistration.cs
      Auth
        Controllers
        Models
        Views
        AuthAreaRegistration.cs
      Shop
        Cart Support
        Controllers
        Models
        Views
        ShopAreaRegistration.cs
    Assets
    Code
      Attributes
      Filters
      HtmlHelpers
    Controllers
    Repositories
    Views
    Global.asax
    Web.config
    Web.sitemap
  ASP.NET MVC Application.Tests
```

The MVC and previous Web Forms ASP.NET application have a lot in common. Functionality and look-and-feel are the same. Also, they access the same two WCF web services (referenced under \Service References) for their data and images. Neither application has a database under their \App-Data folders or any folders with customer or product images. So, both access the bottom tiers (numbered 2-5) in the same way.

But on the Presentation tier things are radically different. In ASP.NET MVC there is no concept of code behind. Instead, the code lives in Controller classes. Furthermore, the aspx pages have no code behind and are called Views. The business objects are collectively referred to as the Model. What we have here is the MVC pattern with Models, Views, and Controllers classes. Microsoft has built an entire Web platform based on the popular MVC Design Pattern. Next, we'll review several details of this application.

Areas

When starting a new ASP.NET MVC project it will automatically create \Models, \Views, and \Controllers folders and you place the appropriate components in their respective folders, that is, controller classes in the Controllers folder; view pages in the Views folder; and all model classes in the Models folder. However, when your MVC project starts getting more than, say, 20 pages you will discover that maintaining a large number of files in a single folder becomes unwieldy. To solve this, MVC 2 introduced the concept of *Areas*. Areas are simply a way to organize and partition the application in logical / functional areas. Each area has its own set of \Models, \Views, and \Controllers folders. We suggest you always use Areas in your MVC applications.

In *Pattern in Action 4.0* we have 3 *Areas*. Admin, Auth, and Shop: these reflect the Administration, Authentication and Shopping functional areas. Each has one controller class and several Model and View classes. The Controller class contains a number of Action methods, each of which corresponds to a View. For example, the Cart action method renders the Cart View and the Customers action method renders the Customers page. In some instances you will see two action methods for a View, an Http GET and Http POST version. GET will display the page, POST will handle post-backs in which form data is validated and sent back to the database.

There are a couple of controller and view classes outside the \Areas folder and are located at the root level \Views and \Controllers folders. In \Controllers you find two controller classes: BaseController (ancestor to all Controller classes) and HomeController. The HomeController has action methods for home page and error page: their views reside under \Views\Home and \Views\Shared folders respectively.

The important Site.Master master page, which is a master of all Views, is located in the same \Shared folder. It provides the standard look-and-feel for the application and includes a header, breadcrumbs, and menu for every page. In Site.Master you find two custom Html helpers: Html.BreadCrumbs and Html.Menu. Breadcrumbs and menus are discussed next.

Breadcrumbs

In the Site.Master master page locate the Html.BreadCrumb helper:

Breadcrumb helper:

```
<%-- Subheader --%>

<div id="subheader">
    <div id="subheader-left"></div>
    <div id="subheader-right">
      <img src="/assets/images/app/arrow.gif" alt="" />
        <span><% = Html.BreadCrumbs() %></span>
    </div>
</div>
<div style="height:4px;background:#6666ff;"></div>
```

The code for this helper can be found under the \Code\HtmlHelpers folder with the BreadCrumb and BreadCrumbHelper classes. The breadcrumb system is based on the SiteMapPath control which allows most breadcrumbs to be built automatically using the web.sitemap configuration file. However, in some cases, the views do not follow the hierarchy defined in web.sitemap and the breadcrumbs require customization. For example, the Login action method in AuthController demonstrates an example of this in which the BreadCrumbs entry in ViewData dictionary is explicitly defined.

Custom breadcrumbs

```
[Menu(MenuItem.Login)]
public ActionResult Login()
{
    ViewData["BreadCrumbs"] = new List<BreadCrumb> {
        new BreadCrumb { Url = UrlMaker.ToDefault(), Title = "home" },
        new BreadCrumb { Title = "login" } };

    return View(new LoginModel());
}
```

Menu

The Site.Master page displays the left-hand-side menus using several Html.Menu helpers, one for each menu item.

Menu helpers

```
<%-- Home --%>

<li><% = Html.Menu("home", "Index", "Home", new { area = "" }) %></li>

<%-- Shopping --%>

<li><% = Html.Menu("shopping", "Index", "Shop", new { area = "Shop" } ) %
<li class="indented"><% = Html.Menu("products", "Products", "Shop", new {
<li class="indented"><% = Html.Menu("search", "Search", "Shop", new { are
<li class="indented"><% = Html.Menu("cart", "Cart", "Shop", new { area =
```

The custom menu helper code is simple. It needs to know for every view which menu is selected and for this we use a custom Filter attribute, called MenuAttribute which is located under the \Filters folder. It assists in setting the selected Menu item for each action method. Below is an example of setting the Cart menu item in the Checkout view. Each menu item has its own enumerated value (defined in the MenuItem enumerator).

Menu filter attribute

```
[Menu(MenuItem.Cart)]
public ActionResult Checkout()
{
    return View();
}
```

Sorting

The user has the ability to sort on three views with these collections: Products, Customers, and Customer Orders. On the customers view, for example, you can see the custom Html.Sorter helpers in action.

Sort helpers

```
<tr class="table-header">
<td align="center"><%= Html.Sorter(Model, "Id", "CustomerId",
<td align="left"><%= Html.Sorter(Model, "Customer Name", "Comp
<td align="left"><%= Html.Sorter(Model, "City", "City", "asc")
<td align="center"><%= Html.Sorter(Model, "Country", "Country"
<td align="center">Edit</td>
<td align="center">Delete</td>
</tr>
```

Three items under \HtmlHelpers folder provide the basis for the custom sorting: an ISortable interface, a generic SortedList, and an Html helper named SorterHelper. SortedList, which implements ISortable, contains a list of objects and two properties: Sort and Order. These properties hold the current Sort Column and Sort Order (ascending or descending). The code below demonstrates how the SortedList is populated and passed to the View:

SortedList in action

```
[Menu(MenuItem.Orders)]
public ActionResult Orders(string sort = "customerId", string order = "asc")
{
    var models = _customerRepository.GetCustomerListWithOrderStatistics(
        new Criterion(sort, order)).ToModel();

    return View(new SortedList<CustomerModel>(models, sort, order));
}
```

First, the action method receives sort and order variables. If none are specified then their default values are assigned. Next, customers are retrieved in the given sort order. Then a typed SortedList item is populated with the sorted list and the current sort and order. This information is passed to the view where it is used to display the list and configure the different soft controls. The Sorter extension method in the Html Sorters is interesting as it demonstrates how to issue postbacks using jQuery. Notice also the use of two hidden controls (named Sort and Order) on the page.

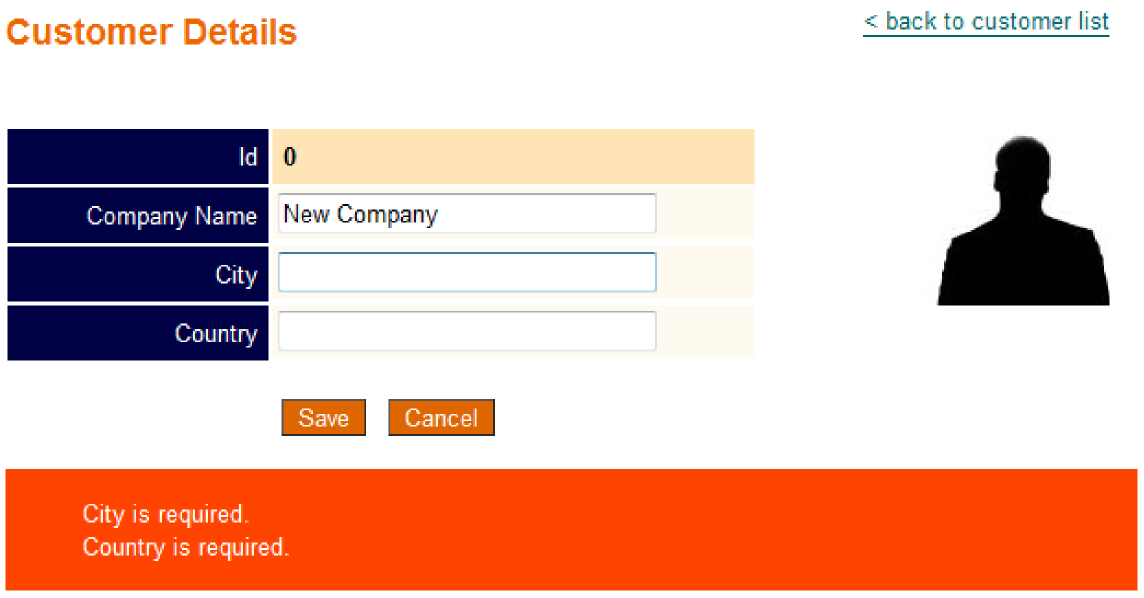
Validation

MVC makes extensive use of Data Annotation. To see an example, open up the CustomerModel class located under \Areas\Admin\Models folder.

Data Annotation

```
public class CustomerModel
{
    /// <summary>
    /// The Customer Identifier.
    /// </summary>
    [DisplayName("Id")]
    public int CustomerId { get; set; }

    /// <summary>
    /// Customer Company name.
    /// </summary>
    [DisplayName("Company Name")]
    [Required(ErrorMessage = "Company Name is required.")]
    [StringLength(30, ErrorMessage = "Company Name can be at
    public string CompanyName { get; set; }
```

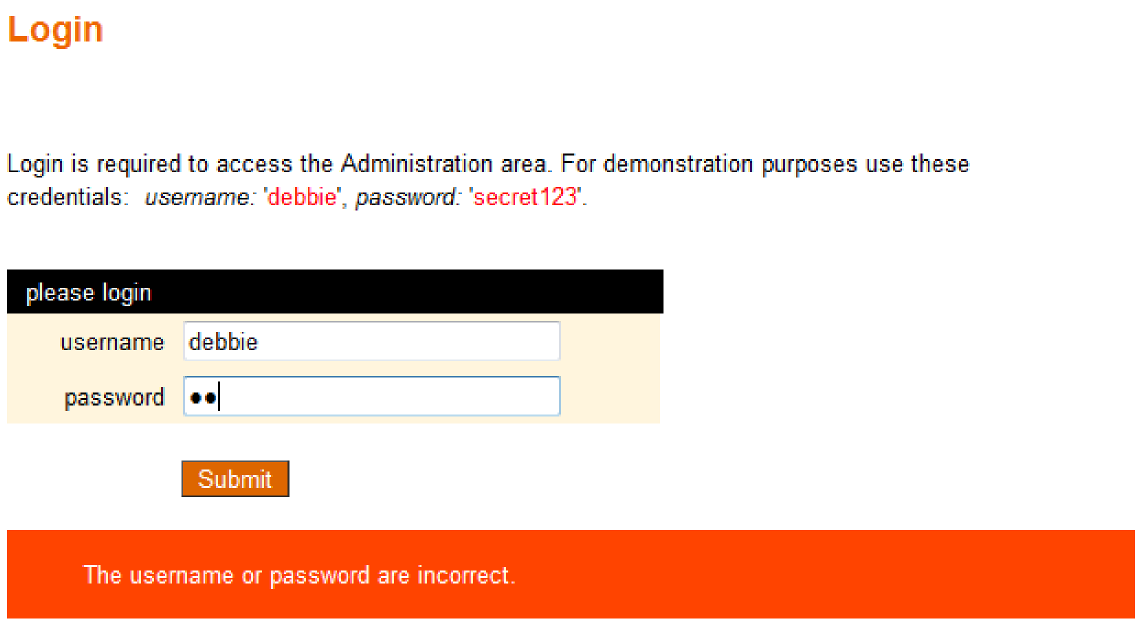
Several attributes decorate the properties in this model. [DisplayName] indicates the label name of the field on the page. [Required] indicates that a value is required. And [StringLength] specifies the maximum string length. When editing a customer and not providing the required entries, you will automatically get the appropriate error message. This validation happens behind the scenes through a process called model-binding. Here is what the error display looks like in *Patterns in Action 4.0*:



Id	0
Company Name	New Company
City	
Country	

City is required.
Country is required.

You can also build custom data annotation validation attributes. Under the \Attributes folder are two custom attributes: one that validates email addresses and one that validates the required minimum length of passwords. Open the LoginModel class and you will see it used to tag the Password property. So now, when you enter a password that does not meet the minimum length you will get the following error.



Login is required to access the Administration area. For demonstration purposes use these credentials: *username:* 'debbie', *password:* 'secret123'.

username	debbie
password	●●

The username or password are incorrect.

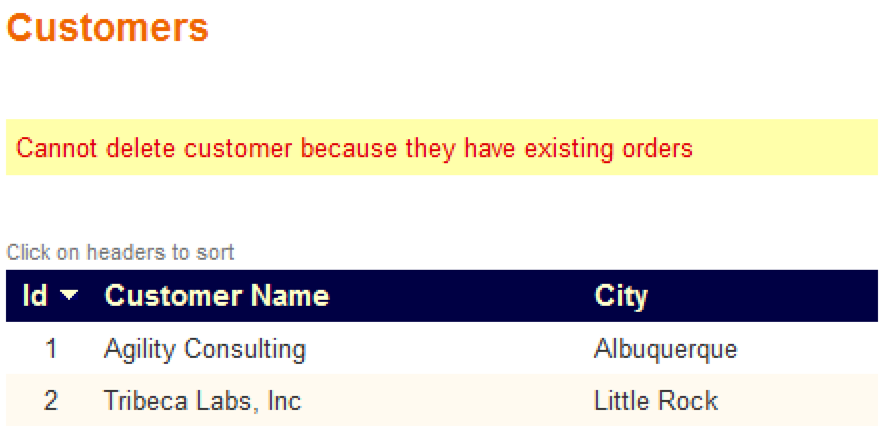
The validation occurs in the Login action method using the ModelState.IsValid validation property. It returns false when not valid. In MVC, data annotations make field level validation very easy. There are numerous built-in data annotation attributes, and we shown just a few of these and how to use them.

Result summaries

An Html.ResultSummary helper displays data access results. It is used only on the Customers page where it displays the outcome of delete actions. The code for this helper can be found in the ResultHelper class. Below you see the ResultSummary helper in action and the resulting view display.

Result Summary helper

```
<h1>Customers</h1>

<% = Html.ResultSummary() %>
```



Customers

Cannot delete customer because they have existing orders

Click on headers to sort

Id ▾	Customer Name	City
1	Agility Consulting	Albuquerque
2	Tribeca Labs, Inc	Little Rock

Repositories

A folder named \Repositories contains repository classes, which are similar to the Repositories in the previously discussed ASP.NET Web Forms application. The difference is that under \Core you will find several Repository interface files. These interfaces are necessary for testing as they allow the repositories to be swapped out by Mock versions of the Repository classes (this is demonstrated in the MVC Test project discussed next).

Have a look at the IRepository interface which generically represents all basic CRUD operations (Insert, Update, etc). This interface allows you to build a consistent repository API for all entities in the system. For example, inserting a new Category is similar to inserting an Order as both have the same Insert method signature: void Insert(T t), where T is the object type being inserted. Using a generic interface like this gives you the opportunity to enforce coding conventions and keep the code simple and easy-to-understand for all members on the team. In *Patterns in Action 4.0* all repository interfaces derive from the IRepository base interface. Repository classes may or may not implement all the interface methods, depending on their needs.

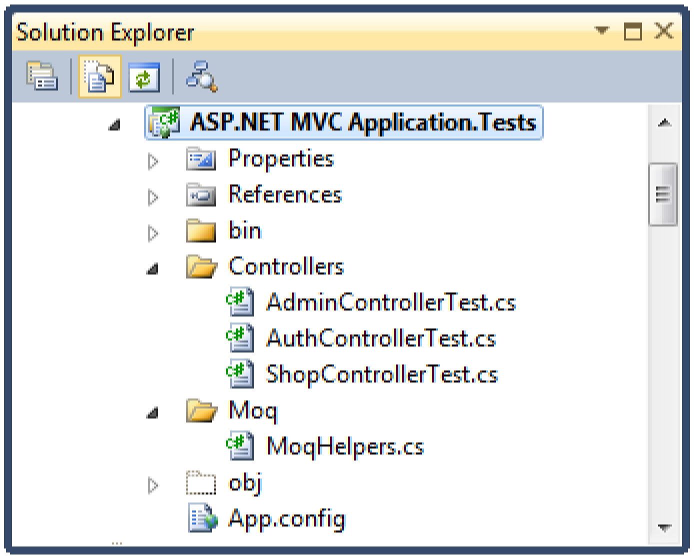
Assets

Finally, the \Assets folder contains application images, style sheet and jQuery script files each in their respective folders.

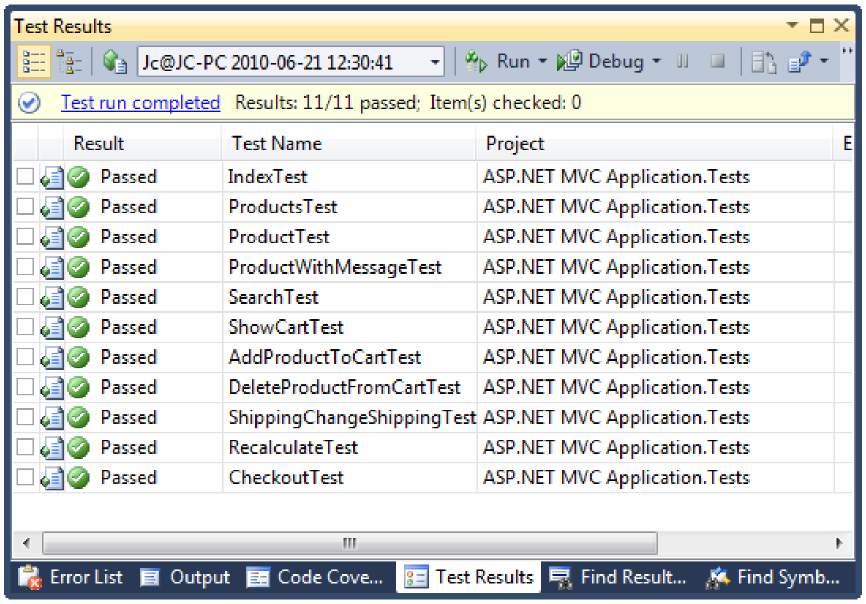
Next we will explore testing the MVC application.

ASP.NET MVC Test Application

Select ASP.NET MVC Application.Tests as your startup project. This project is designed to perform unit testing on the ASP.NET MVC application. Here is the project:



This project uses Microsoft's built in unit testing platform called *mstest* and a 3rd party tool called *Moq* (Moq can be downloaded from here: http://code.google.com/p/moq/ and is available under a new BSD license). If you run the project now you should see that all tests pass.



Test run completed Results: 11/11 passed; Item(s) checked: 0

Result	Test Name	Project
Passed	IndexTest	ASP.NET MVC Application.Tests
Passed	ProductsTest	ASP.NET MVC Application.Tests
Passed	ProductTest	ASP.NET MVC Application.Tests
Passed	ProductWithMessageTest	ASP.NET MVC Application.Tests
Passed	SearchTest	ASP.NET MVC Application.Tests
Passed	ShowCartTest	ASP.NET MVC Application.Tests
Passed	AddProductToCartTest	ASP.NET MVC Application.Tests
Passed	DeleteProductFromCartTest	ASP.NET MVC Application.Tests
Passed	ShippingChangeShippingTest	ASP.NET MVC Application.Tests
Passed	RecalculateTest	ASP.NET MVC Application.Tests
Passed	CheckoutTest	ASP.NET MVC Application.Tests

MVC testing involves testing controller classes. Essentially the Views are taken out of the picture and user actions are ‘simulated’ by calling the Action methods in the Controllers directly from our testing platform. Views (i.e. web pages) are notoriously difficult to test, so this makes systematic testing easier and faster.

The main idea of unit testing is that these are tests that you run quickly to verify that a change you made does not accidentally break something that used to work before. We’ve taken the View out of the testing. However, there still is the Model which involves data base access which is relatively slow and is non-repeatable.

Here is an example of this non-repeatability. Let’s say that in your application users register themselves to purchase products. If your database is designed correctly, it will not let you insert the same user twice. This is enforced through a unique composite key or a unique index on, for example the email column. Either way, no duplicate records can be inserted. The business reasons for this restriction are rock solid, but it makes running repeatable units tests hard. To resolve this we *mock* the Model.

Mocking

To take the database out of the equation, mock objects are used. A mock object simulates or mimics the functionality of a real object in a predictable and controlled way. To simulate the Model we will create mock objects for each of the Repository objects.

Mocking is a technique in which classes are stubbed out and mocked by a mock class. They behave just like the real objects. Mock objects have the same interface as the objects they simulate. This is one of the reasons that the Repository classes in the MVC project are implemented with interfaces, such as, IRepositoryCategory, IRepositoryCustomer, etc. The controller class being tested is unaware that it executes mock objects rather than the real Repository objects.

Dependency Injection

So, how do we ‘trick’ the Controller class into running mock objects rather than the real objects? The answer lies in a technique called *Dependency Injection* (also referred to as IOC – Inversion of Control). In the MVC application Dependency Injection occurs in the Controller’s constructor. The following code snippet of the ShoppingController demonstrates how this works.

```
public class ShopController : BaseController
{
    private ICategoryRepository _categoryRepository;
    private IProductRepository _productRepository;
    private ICartRepository _cartRepository;

    /// <summary>
    /// Default Constructor for ShopController.
    /// </summary>
    public ShopController()
        : this(new CategoryRepository(),
               new ProductRepository(), new CartRepository())
    {
    }

    /// <summary>
    /// Overloaded 'injectable' Constructor for ShopController.
    ///
    /// Pattern: Constructor Dependency Injection (DI).
    /// </summary>
    /// <param name="categoryRepository"></param>
    /// <param name="productRepository"></param>
    /// <param name="cartRepository"></param>
    public ShopController(ICategoryRepository categoryRepository,
        IProductRepository productRepository,
        ICartRepository cartRepository)
    {
       _categoryRepository = categoryRepository;
       _productRepository = productRepository;
       _cartRepository = cartRepository;
    }
```

First of all, notice that the controller has 3 private fields with repository interfaces: ICatetoryRepository, IProductRepository, and ICartRepository. These fields are initialized in the controller’s constructor (either the default or the overloaded constructor). Normally, when running the ASP.NET MVC application, the default constructor is called which creates 3 standard repository objects. However, in the Test project the overloaded constructor is called with 3 mocked repository objects.

The relevant code where mocked repository objects are passed into the constructor can be found in the Test project in the ShopControllerTest class:

```
// Private helper. Creates shop controller.
// This is a Factory Method.
private ShopController CreateShopController()
{
    // Note: This is where DI (Dependency Injection) takes place.
    // The repositories are injected (via the constructor) into the controller.
    return new ShopController(mockCategoryRepository.Object,
                              mockProductRepository.Object,
                              mockCartRepository.Object);
}
```

The Initialize method in ShopControllerTest is where the mock objects are created and being prepared for the test. Moq uses on a pair of *Setup* and *Return* extension methods with which you configure the mock object's behavior. It's like telling the mock objects "if they call you in this way, this is how you respond". Essentially, Setup is the process where you 'teach' the mock objects to respond in a certain way when called in a certain way

Here is an example:

```
// Setup getting the shopping cart
var cartItems = new[] { new ShoppingCartItem {
     Id = 1, Name = "test-product", Quantity = 1 } };
var cart = new ShoppingCart { CartItems = cartItems, ShippingMethod = "UPS" };
mockCartRepository.Setup(c => c.GetCart()).Returns(cart);
```

It states that when you call the GetCart() method on the mocked CartRepository you return the shopping cart object as defined in the first two lines of code. There is far more to Moq than we are able to demonstrate here, but Moq offers a lot of flexibility by its use of lambda methods.

One additional item we'd like to mention. When designing and developing tests you may run into IOC (Inversion of Control) containers which are a popular technique to control which classes to instantiate at runtime (typically you configure this with XML files). Here we could have used an IOC container to control which of the repositories (the real or mocked ones) to instantiate. It uses the same technique (i.e. *Dependency Injection*) to dynamically assign object references to the repositories. We opted not to use any IOC container to keep the tests relatively simple, but at the end the results are the same.

Arrange-Act-Assert Testing Pattern

Test classes are tagged with testing attributes. The test controller class is decorated with the [TestClass] attribute. Test initialization is done in a common initialization step decorated with the [TestInitialize] attribute. Its counterpart is [TestCleanup] which cleans up after running all tests, but this not used in our example. Finally, each test method is tagged with the [TestMethod] attribute.

Test methods normally have a very narrow focus and each should test one thing or one behavior only. Testing methods are normally arranged according to a 3 step pattern, which is called AAA (Arrange-Act-Assert). Arrange is where you create and initialize the items involved in the test. The Act is where you execute the test, and Assert is where you evaluate the outcome of the test. A simple example is listed below:

```
[TestMethod]
public void ProductTest()
{
    // Arrange
    var controller = CreateShopController();

    // Act
    var result = controller.Product(1) as ViewResult;

    // Assert
    Assert.IsInstanceOfType(result, typeof(ViewResult));
    Assert.IsInstanceOfType(result.ViewData.Model, typeof(ProductModel));
    Assert.AreEqual((result.ViewData.Model as ProductModel).Name, "test-product");
}
```

The Controller classes being tested are located under the Controllers folder. Please note that of the 3 controller classes only the ShopController is implemented. The purpose of this project is to demonstrate MVC unit testing, dependency injection and mocking, and not to get full code coverage. To reach adequate test coverage would involve a significant effort and include many dozens of unit test cases. Just as an aside: automatic testing may give you a sense of security, but the seemingly simple question of ‘how much testing is enough?’ turns out to be hard to answer. It involves code coverage metrics and code coverage analysis that require some ‘deep thinking’.

Faking contexts

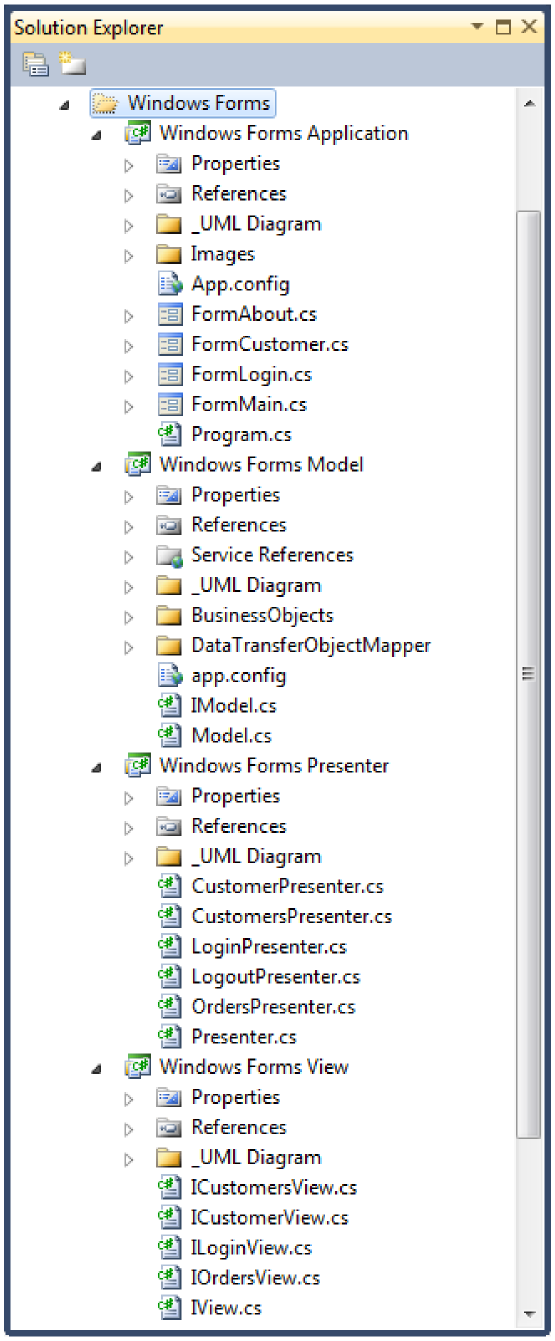
We have discussed earlier that some controller action methods only respond to Http GET requests or Http POST requests. The [HttpGet] and [HttPost] attributes are used for this purpose. Simulating POST and GET requests in a testing environment requires some extra code. We used a post by Microsoft’s Hanselman on how to create fake HttpContext or fake ControllerContext objects to make this possible (by the way, ‘fake’ is an official testing term). We used this in the SearchTest method to indicate that we are testing a POST in which a search is to be performed. The MoqHelper class is located under the \Moq folder. You also find it online here:
http://www.hanselman.com/blog/ASPNETMVCSessionAtMix08TDDAndMvcMockHelpers.aspx

As the MVC test application demonstrates it is important that you construct your controller classes and repository classes in such a way that they can be used in unit testing frameworks. Testing MVC web applications can get complex quickly. Entire books have been written on TDD (Test Driven Development) and related topics. Tools like IOC containers (not implemented here) and Moq are powerful, but they have a learning curve which means that if your organization is committed to testing they need to allocate the necessary resources (skills, money, time, etc). In this example we have offered a solid starter foundation for your MVC unit testing efforts.

Next we discuss the Windows Forms Application.

Windows Forms Application

Expand the Windows Form folder as well as the 4 projects within this folder. Your explorer should look like the image below:



Windows Forms
- Windows Forms Application
  - Properties
  - References
  - _UML Diagram
  - Images
  - App.config
  - FormAbout.cs
  - FormCustomer.cs
  - FormLogin.cs
  - FormMain.cs
  - Program.cs
- Windows Forms Model
  - Properties
  - References
  - Service References
  - _UML Diagram
  - BusinessObjects
  - DataTransferObjectMapper
  - app.config
  - IModel.cs
  - Model.cs
- Windows Forms Presenter
  - Properties
  - References
  - _UML Diagram
  - CustomerPresenter.cs
  - CustomersPresenter.cs
  - LoginPresenter.cs
  - LogoutPresenter.cs
  - OrdersPresenter.cs
  - Presenter.cs
- Windows Forms View
  - Properties
  - References
  - _UML Diagram
  - ICustomersView.cs
  - ICustomerView.cs
  - ILoginView.cs
  - IOrdersView.cs
  - IView.cs

The Windows Forms Project

The Windows Forms Application is a standard Windows application. Like the two web projects discussed before, this application also communicates exclusively with WCF web services and there is no data stored locally, such as a local database.

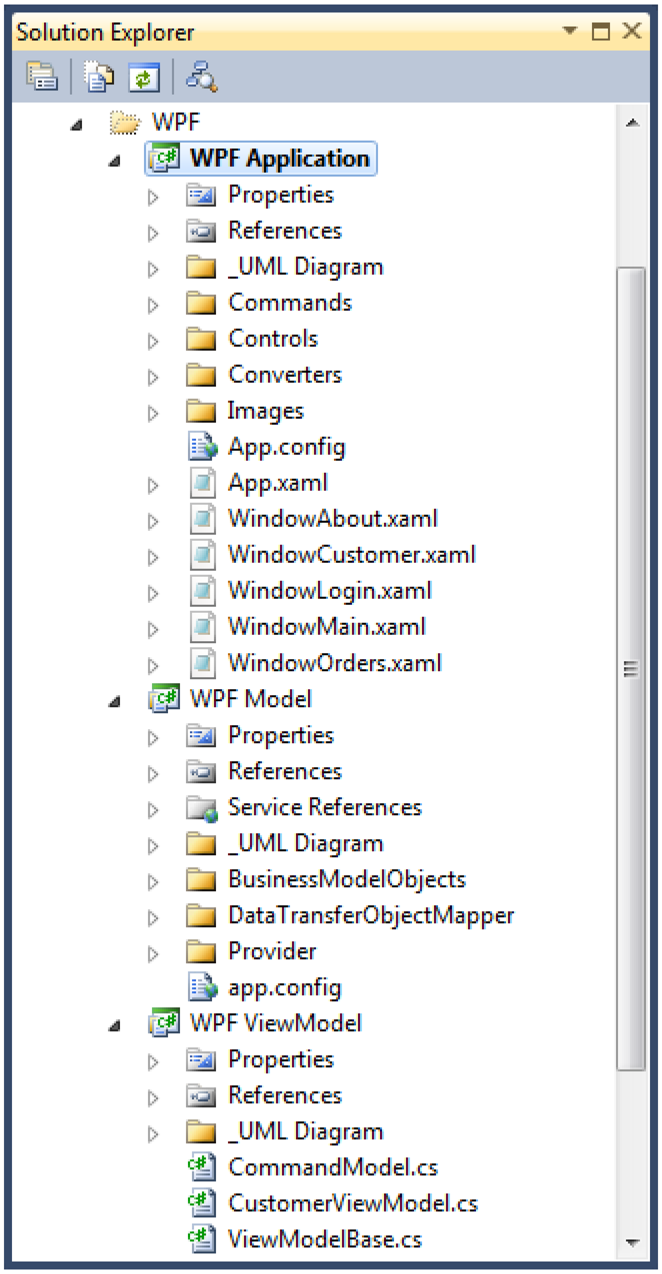
The Windows Forms application is built around a Model-View-Presenter design pattern. Three projects: Model, View, and Presenter represent each of the parts. The forms in the Application derive from the View interfaces defined in the View project.

FormMain is the main form. All other forms are supporting dialogs that pass data back and forth between FormMain and the dialog windows. Dialog result values determine whether FormMain processes the dialog data or not. Just a heads up: this model of placing all logic in FormMain works for this particular application because the application is very much centered on the main form. This may or may not be the optimal approach for applications with more complex child forms.

This application has its own set of local Business Objects (in Model project) in parallel to the ones in the Business Layers on the server side. The client view of a Business Object is not necessarily the same as the server view. The Mapper class in folder \DataTransferObjectMapper maps data transfer objects to the local business objects and vice versa. The three client-side business objects are located in the \BusinessObjects folder.

WPF Application

Expand the WPF Application folder until you see the view below:



- WPF
  - WPF Application
    - Properties
    - References
    - _UML Diagram
    - Commands
    - Controls
    - Converters
    - Images
    - App.config
    - App.xaml
    - WindowAbout.xaml
    - WindowCustomer.xaml
    - WindowLogin.xaml
    - WindowMain.xaml
    - WindowOrders.xaml
  - WPF Model
    - Properties
    - References
    - Service References
    - _UML Diagram
    - BusinessModelObjects
    - DataTransferObjectMapper
    - Provider
    - app.config
  - WPF ViewModel
    - Properties
    - References
    - _UML Diagram
    - CommandModel.cs
    - CustomerViewModel.cs
    - ViewModelBase.cs

The WPF Application

Three projects make up the WPF Application. The application is built around the Model ViewModel View (MVVM) design pattern. Model and ViewModel classes reside in similarly named projects. The View is represented by the Forms in the WPF Application, that is, the Forms are the Views.

The application has five windows (the Window*.xaml files). Several WPF specific classes exist in the \Commands and \Converters folders. The \Controls folder contains a control that facilitates the glowing menus.

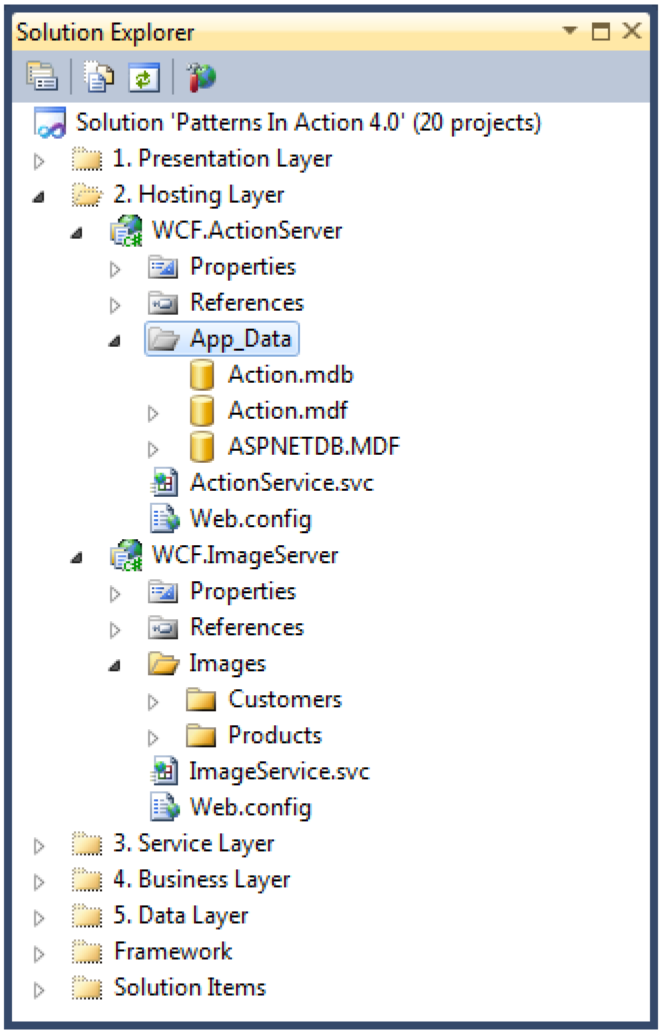
The WPF Model has three business model objects in the \BusinessModelObjects folder. They are CustomerModel, OrderModel, and OrderDetailModel, each of which derives from base class BaseModel. BaseModel ensures that the methods are called on the UI thread (a WPF requirement). The Mapper class maps DTOs (Data Transfer Objects) to Business Model Objects and vice versa. The WPF client receives DTOs from the WCF service. Finally, the \Provider folder contains the classes where the actual calls are made into the WCF services.

The WPF ViewModel project is relatively small, but it contains the 'command and control' classes of the MVVM pattern. The ViewModel is where Model and View events are coordinated and processed. This pattern relies heavily on WPF's command system as well as its data binding facilities. CommandModel is an abstract class that encapsulates routed UI commands. CustomerViewModel is the place where all customer-related events and requests are coordinated and processed.

Details on MVVM are discussed later in this document.

WCF.ActionServer and WCF.ImageServer

Expand the hosting layer and its two projects until you see the view below.


```
Solution Explorer
Solution 'Patterns In Action 4.0' (20 projects)
  1. Presentation Layer
  2. Hosting Layer
    WCF.ActionServer
      Properties
      References
      App_Data
        Action.mdb
        Action.mdf
        ASPNETDB.MDF
      ActionService.svc
      Web.config
    WCF.ImageServer
      Properties
      References
      Images
        Customers
        Products
      ImageService.svc
      Web.config
  3. Service Layer
  4. Business Layer
  5. Data Layer
  Framework
  Solution Items
```

The hosting layer in *Patterns in Action 4.0*

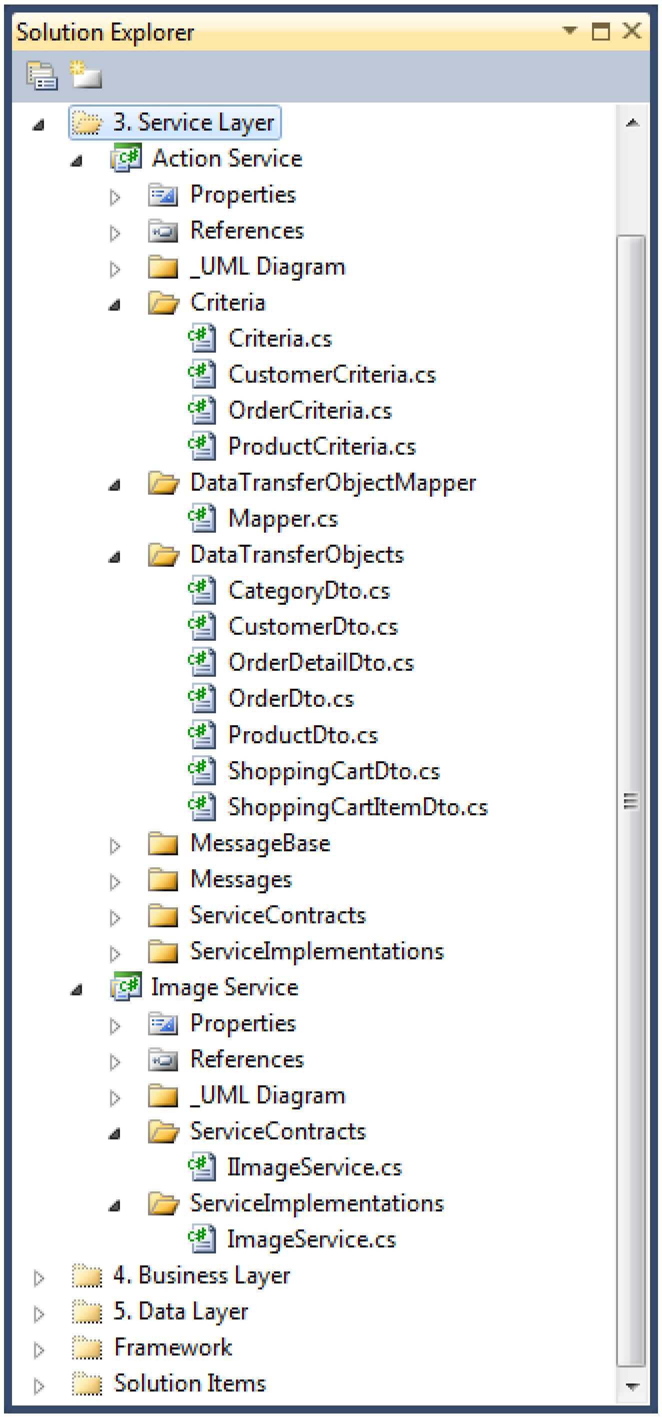
Two WCF Services are exposed in the hosting layer. First is ActionService which represents the complete *Patterns In Action 4.0* application services. Notice that the hosting layer is also the place where copies of SQL Express and MS Access databases (under \App_Data) are kept. This is the perfect place to physically keep the databases (that is, for SQL Express and MS Access) as it is the only point of entry for all clients. Of course, SQL Server can be anywhere and, in fact, usually runs on a dedicated server anyhow (but you would only have one instance supporting 4 different client applications).

(product images are new)
The second hosting project is WCF.ImageServer which hosts a service named ImageService. It is designed to serve customer and product images to its clients. Having all images at a central place prevents you from having to maintain duplicate images. All image files are stored in sub-folders under the \Images folder.

Both hosting projects are simple and have little or no code. They reference the service layer from the ActionService.svc and ImageService.svc files respectively. In your own hosting projects, when building a hosting layer, expect to spend most of you time configuring the hosting environment in the <system.serviceModel> sections in the web.config files (although WCF 4 configuration has been made simpler than prior versions).

ActionService and ImageService

Expand the service layer and its two projects until you see the view below.



```
3. Service Layer
  Action Service
    Properties
    References
    _UML Diagram
    Criteria
      Criteria.cs
      CustomerCriteria.cs
      OrderCriteria.cs
      ProductCriteria.cs
    DataTransferObjectMapper
      Mapper.cs
    DataTransferObjects
      CategoryDto.cs
      CustomerDto.cs
      OrderDetailDto.cs
      OrderDto.cs
      ProductDto.cs
      ShoppingCartDto.cs
      ShoppingCartItemDto.cs
    MessageBase
    Messages
    ServiceContracts
    ServiceImplementations
  Image Service
    Properties
    References
    _UML Diagram
    ServiceContracts
      IImageService.cs
    ServiceImplementations
      ImageService.cs
4. Business Layer
5. Data Layer
Framework
Solution Items
```

The service layer with Action Service and Image Service projects

Two projects are located in the service layer: ActionService, which represents all of the *Patterns in Action 4.0* application services and ImageService which serves up customer and product images.

The Service layer concept is fundamental to most modern 3-tier architectures. Our Service layer is an implementation of the Façade design pattern. It is called an Application Façade because it can be deployed on the cloud, and any person, any device, or any application can consume its services from anywhere in the world. All clients consume the exact same service API. Clients can include: command line tools, full blown WPF applications, mobile devices, web sites; you name it.

The Façade pattern is a simple, but important pattern that will help you build clean APIs. It is hard to overstate the importance of the Façade design pattern in modern .NET architectures

As an aside, notice that our service API is <u>not</u> very object-oriented. Instead, it is more functional or transaction oriented in which each method encapsulates a single transaction (i.e. a complete unit-of-work). You don't see methods or properties like OpenResultSet, GetNextResult, GetLastResult, etc. This would require the service layers to maintain state between calls which would put the client in control of completing the transaction, which is undesirable. Also, the use of the Request/Response design pattern (discussed later) nicely facilitates this kind of 'chunky' transaction-oriented API. We will revisit API design later in this section.

Façades are frequency grouped by functionality, so you may have Membership Façade, Employee Façade, Reporting Façade, and others. We usually organize these Façades so that they match the 'Vertical Tiers' or modules described earlier in this document. Having said that, *Pattern in Action 4.0* has just one façade because it is small and we have tried to keep things simple. The service interface is embodied by the IActionService service contract. We could have partitioned this interface into three logical Façades matching the main modules in the application: a Membership Façade which supports login and logout, a Product Façade which supports the shoppers and their product search activities, and a Customer Façade which supports the administrative tasks of maintaining customers and their orders.

What is the Service Layer's responsibility? In addition to calling and managing business objects, data access objects, and processing the standard business logic, every interface method in the Façade should validate the incoming arguments, authorize the action, and handle and coordinate database transactions, possibly covering numerous business object changes. Every data item that enters or leaves the façade must be checked and validated – even when, for example, data validation and user authentication have already taken place at the UI (for example, the UI could have checked for required fields). The reason for this is twofold: security and reusability. Different clients will be accessing the façade, and these clients are possibly written by different developer teams with different skill levels. Nothing can be assumed about how a client is implemented and therefore, facades must take a very *conservative position* in order to maintain the integrity of the system.

Let's look at the classes and types in the Action Service project. Under folder \ServiceContracts, the file IActionService defines the public web service interface. The implementation of this interface is found in ActionService located under \ServiceImplementations. ActionService is the 'central hub' of the service based 3-tier architecture and it is important that you study and understand its role and responsibilities.

In ActionService you will find references to instances of the DAOs (Data Access Objects). Also, there is fairly extensive parameter validation, mapping from DTOs (Data Transfer Objects) to BOs (Business Objects) and functionality that supports the SOA and Messaging patterns, discussed later. Transactions are 'auto-committed', but you would explicitly manage transactions here for updates that involve multiple database entities. Finally, a couple of private helper methods are included to support identifier encryption, but they are not currently used (to keep things simple).

The Data Transfer Object (DTO) design pattern is an Enterprise pattern that creates objects for the sole purpose of transferring business data – there is no behavior (methods or properties) associated with these objects. Seven such objects are used in passing data to and from the Web Service. They are located in folder \DataTransferObjects. A mapper class in the \DataTransferObjectMapper folder transforms DTOs to BOs (business objects) and vice versa.

Folder \MessageBase contains two base classes, RequestBase and ResponseBase. They contain important members that are used in all derived RequestMessages and ResponseMessages respectively. AcknowledgeType and PersistType are enumerations that facilitate accurate communication between clients and service via messages.

The folder named \Messages contains pairs of Response and Request objects -- one pair for every web method. For example, the method GetCustomers has a single argument named CustomerRequest and a return value named CustomerResponse. CustomerResponse includes an array of CustomerTransferObjects in which each element represents a customer.

Criteria objects hold search criteria coming from the client (Presentation layer). They pass through the layers and arrive in the Data Layer where they are used construct the' query clauses in the dynamic SQL statements. Different business object types have different criteria objects.

As mentioned before, communication between the service layer and presentation layer is message based. Messages are classes that contain query parameters and query results packaged up as a message. Two different message types in *Patterns in Action 4.0* include: Request messages and Response messages. These messages help in the creation of simple, clean, symmetrical APIs. Request messages are passed in as arguments and Response messages are returned as return values.

Their usage pattern in C# is as follows:

```
[OperationContract]
SomethingResponse GetSomethings(SomethingRequest request);

[OperationContract]
SomethingResponse SetSomethings(SomethingRequest request);
```

Have a look at IActionService under \ServiceContracts and see how simple and elegant the API really is. By the way, this is the entire application services API for *Patterns in Action 4.0*.

Here is the API in C#:

```
[ServiceContract(SessionMode = SessionMode.Required)]
public interface IActionService
{
    [OperationContract]
    TokenResponse GetToken(TokenRequest request);

    [OperationContract]
    LoginResponse Login(LoginRequest request);

    [OperationContract]
    LogoutResponse Logout(LogoutRequest request);

    [OperationContract]
    CustomerResponse GetCustomers(CustomerRequest request);

    [OperationContract]
    CustomerResponse SetCustomers(CustomerRequest request);

    [OperationContract]
    OrderResponse GetOrders(OrderRequest request);

    [OperationContract]
    OrderResponse SetOrders(OrderRequest request);

    [OperationContract]
    ProductResponse GetProducts(ProductRequest request);

    [OperationContract]
    ProductResponse SetProducts(ProductRequest request);

    [OperationContract]
    CartResponse GetCart(CartRequest request);

    [OperationContract]
    CartResponse SetCart(CartRequest request);
}
```

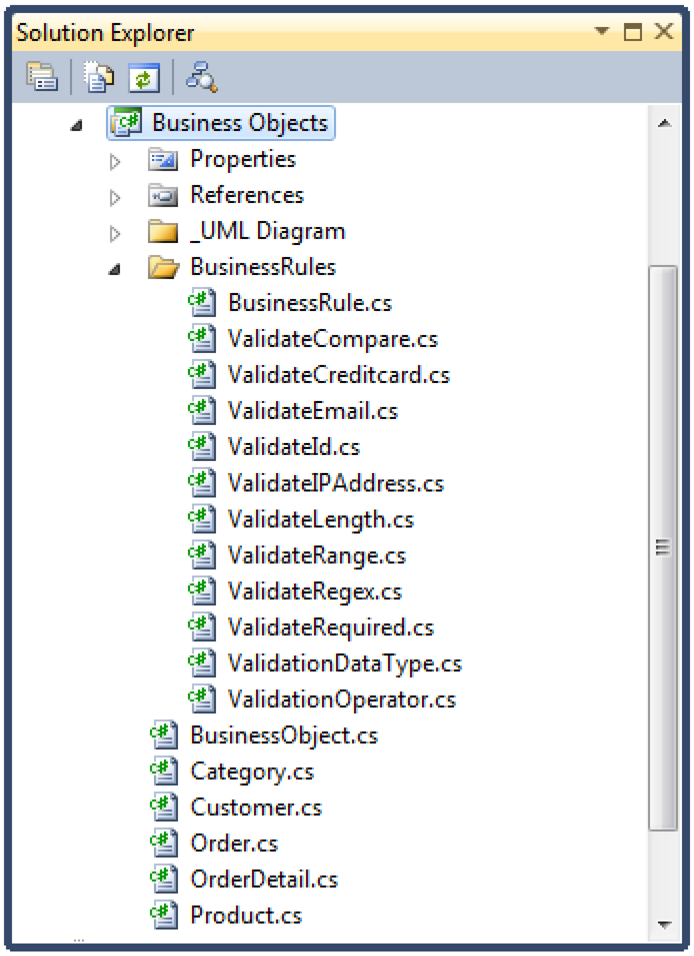
Here is the same API in VB:

```
<ServiceContract(SessionMode := SessionMode.Required)> _
Public Interface IActionService
      <OperationContract> _
      Function GetToken(ByVal request As TokenRequest) As TokenResponse

      <OperationContract> _
      Function Login(ByVal request As LoginRequest) As LoginResponse

      <OperationContract> _
      Function Logout(ByVal request As LogoutRequest) As LogoutResponse

      <OperationContract> _
      Function GetCustomers(ByVal request As CustomerRequest) As _
            CustomerResponse

      <OperationContract> _
      Function SetCustomers(ByVal request As CustomerRequest) As _
            CustomerResponse

      <OperationContract> _
      Function GetOrders(ByVal request As OrderRequest) As _
            OrderResponse

      <OperationContract> _
      Function SetOrders(ByVal request As OrderRequest) As _
            OrderResponse

      <OperationContract> _
      Function GetProducts(ByVal request As ProductRequest) As _
            ProductResponse

      <OperationContract> _
      Function SetProducts(ByVal request As ProductRequest) As _
            ProductResponse

      <OperationContract> _
      Function GetCart(ByVal request As CartRequest) As CartResponse

      <OperationContract> _
      Function SetCart(ByVal request As CartRequest) As CartResponse
End Interface
```

Project ImageService is a simple WCF service returning customer and product images (product images are new in this latest release). It is accessible as a web service from all Presentation tier applications. The current implementation does not include functionality to add, update, and remove images, which is what you would expect in a full-blown Image service application. ImageService simply serves as an example of how to create and host a WCF service for managing any kind of *singleton* resource (i.e. a resource of which you can have only one, such as, database, images, documents, etc).

BusinessObjects

Expand the BusinessObjects Project until you see the view below:



```
Business Objects
    Properties
    References
    _UML Diagram
    BusinessRules
        BusinessRule.cs
        ValidateCompare.cs
        ValidateCreditcard.cs
        ValidateEmail.cs
        ValidateId.cs
        ValidateIPAddress.cs
        ValidateLength.cs
        ValidateRange.cs
        ValidateRegex.cs
        ValidateRequired.cs
        ValidationDataType.cs
        ValidationOperator.cs
    BusinessObject.cs
    Category.cs
    Customer.cs
    Order.cs
    OrderDetail.cs
    Product.cs
```

The business layer with Business Objects project

Project BusinessObjects is a class library that contains business objects (also called domain objects). Business objects are objects that encapsulate data with associated behavior that is relevant to the business at hand. Business objects in *Patterns in Action 4.0* are: Category, Product, Customer, Order, and Order Detail.

Depending on the requirements, business rules may be encoded in these classes. In *Patterns in Action 4.0* the objects are kept simple and focus on validation-type business rules. In a more complex application you may have interdependent business rules encoded outside these objects (for example, if the bank account is over $5000 and more than 3 transactions have taken place in the last month, and the account holder's age is > 35 and is on the same job for more than 2 years, then accept the application).

A simple business rules engine is built in *Patterns in Action 4.0*. It is implemented by a combination of the BusinessObject class (ancestor to all business objects) and the BusinessRule class (ancestor to all business rules). Several rule implementations can be found in the \BusinessRules folder.

Let's look at the Customer class and see how the business rules work. The customer's validation-type rules are: Id must be greater or equal to zero, Company name is required and must be between 1 and 20 characters, City name is required and must be between 1 and 15 characters, Country name is required and must be between 1 and 15 characters. These rules are specified in the default constructor. Once the object is populated with data, the application can call Validate() on the Customer object. If it returns false, which means validation fails, you check the ValidationErrors property which will have a list of error messages available. The application can then decide how to handle these errors.

Here is an example of a more complex rule. There may be a rule that states that customers are ranked according to their recent order history. Customers with 10 or more orders over the last 3 months receive gold status, customers with 5 to 10 orders receive silver status, and customers with fewer than 5 orders are given bronze status. These statuses can be used to determine the relative priority treatment they receive when calling a 1-800 customer support number. When an order is placed, the rule in the business object re-evaluates the customer's ranking. These kind of rules require (in most cases) custom validation code.

Let's now discuss the issue of business object persistence (i.e. saving the object data to a database). Business objects themselves are not involved with data access, so you won't find any Load, Update, or Save methods on business objects. Data access is handled entirely by the Data Layer, which accepts and returns values from and to business objects. The Data Layer accepts business objects, then gets and sets their data via object properties. The Data Layer has knowledge about the object model, as well as the data model, and it is therefore the intermediary between the two models (object model and data model).

Clearly, there are different ways to implement and enforce business rules. It should be mentioned that for simple property based validation, a viable alternative would be to use DataAnnotations. DataAnnotations were introduced by Microsoft when they released their Dynamic Data controls. ASP.NET MVC platform has taken the ball and ran with it, by making it very easy to use DataAnnotations for validation and rendering purposes.

POCOs

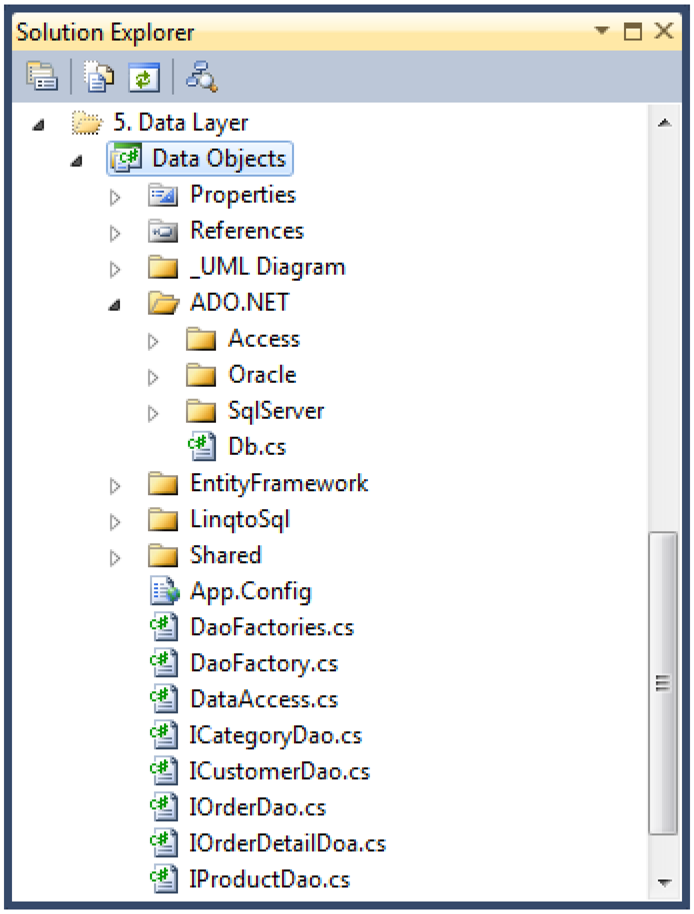
The word POCO is an abbreviation of '*Plain Old CLR Object',* meaning objects that are just normal C# or VB .NET objects and that have no dependency or relationship to any other system or platform. Our business objects are POCOs, as are the DTOs (Data Transfer Objects) used in the Framework. A good example of non-POCO objects are the Entities in the Entity Framework. Many developers use Entities are their business objects, but the problem with that it will tie your system to the Entity Framework. Once you change your persistence technology to, say, NHibernate, you are stuck with Entities (i.e. Business Objects) that have payload that is totally meaningless in this new context.

In the latest release of the Entity Framework, Microsoft has added support for POCOs. The only requirement is that the Business Objects have some notion of a unique key which is done by tagging one or more properties with the [Key] attribute. As mentioned, our Business Objects are POCOs, but they do not use this Key attribute, primarily because they travel through several layers and are transformed to and from DTOs. Depending on your requirements you may want to evaluate and consider using POCOs and the [Key] attributes when using business objects and the Entity Framework.

We will not further discuss POCOs, but we know that many .NET developers are confused and wanted to point out that there is nothing magical about them, and, in fact, that we have been using them all along in the Design Pattern Framework.

DataObjects

Expand the DataObjects project until you see the view below:



The data layer with Data Objects project

The Data Tier resides in a class library project named DataObjects. This project is located in the \DataLayer folder. Here you have to option to use one of three data access platforms: ADO.NET, LINQ TO SQL, or Entity Framework. In web.config you specify which platform to use (how you do this was discussed earlier in this document).

Five Dao (Data Access Object) interfaces, ICategoryDao, IProductDoa, ICustomerDao, IOrderDao, and IOrderDetailsDao define the interface between Business Layer and Data Layer. The Business Layer does not know anything about data access and these interfaces facilitate the persistence of the business objects. In our example, each business object type has its own Dao. As mentioned before, the business objects themselves are not involved with databases or data access. Business object persistence is entirely handled by the Data Layer which maps the object model to the relational data model and vice versa.

ADO.NET as the data access platform

ADO.NET data access classes are located in the \AdoNet folder. Each of the three databases supported in *Patterns in Action 4.0*, MS Access, SQL Server, and Oracle, require their own ADO.NET implementations of the aforementioned DAO interfaces. Three subfolders, named \Access, \SqlServer, and \Oracle contain these implementations. For example, the Access folder contains implementation classes named AccessCategoryDoa, AccessCustomerDoa, AccessCustomerDao, AccessOrderDao, and AccessOrderDetailDao. These are database specific implementations of the general DAO interfaces. Each database has its own implementation but you will find that In *Patterns in Action 4.0* the actual SQL statements for each of the supported databases are, in fact, very similar.

Microsoft's' built-in Data Provider design pattern participates in handling the database differences. At the highest level there is DaoFactories (a Factory of Factories), which, given a data provider name, returns a database specific Factory. Database specific factories derive from a base class named DaoFactory. Database specific implementations of these factories are found in the aforementioned \Access, \SqlServer, and \Oracle folders. For example, the Access version is called AccessDaoFactory.

DaoFactories return database specific implementations of the five IDao interfaces. For example, the AccessDaoFactory, returns AccessCategoryDao, AccessProductDao, AccessCustomerDao, AccessOrderDao, and AccessOrderDetailDao. These classes are used in the Service layer which coordinates the persistence of business objects by managing both business objects and data access objects. The Business Layer uses a static class named DataAccess, which shields it from the Factory and other data base specific access details. The DataAccess class is used in the ActionService class in the Service Layer.

Finally, the important helper class named **Db** is the real workhorse of the ADO.NET Data Layer. It handles all ADO.NET 4.0 data access calls and shields the rest of the system from low level database differences, such as, retrieving newly generated identifiers (identities or auto-numbers) from the database.

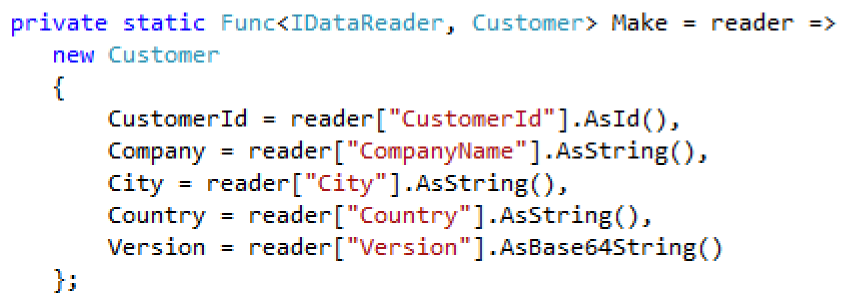
The **Db.cs** class has gone through some significant changes since earlier versions. It now uses the very fast DbDataReader class (often referred to as a 'firehose' data stream) to get the data from the database. It uses DbParameters, a significant improvement as it helps preventing SQL Injection attacks. Please note that we pass the DbParameters in as a linear object[] array. We prefer it this way because the client does not have to know anything about DbParameters. However, you can easily change this to an array of DbParameters being passed into the Db methods. Generic 'make' delegate methods are passed into the Read and ReadList methods which makes creating and populating business objects very easy and flexible.

There are several extension methods in the data access layer that you need to be aware of. They are of great help in getting data in and out the database. First there are two methods in the Db class: AppendIdentitySelect and SetParameters. And, secondly, there is a group of extensions methods that help getting data in and out of the DbDataReader.

The AppendIdentitySelect extension method appends the database specific identity select to the sql string (always an INSERT statement). For example, for SQL Server the identity select is ";SELECT SCOPE_IDENTITY();". By appending the sql insert and the identity select, you prevent the need for an extra database request. It inserts the new record and returns the newly created identity value in one and the same request. Please note that MS Access does not support this concatenation of SQL statements.

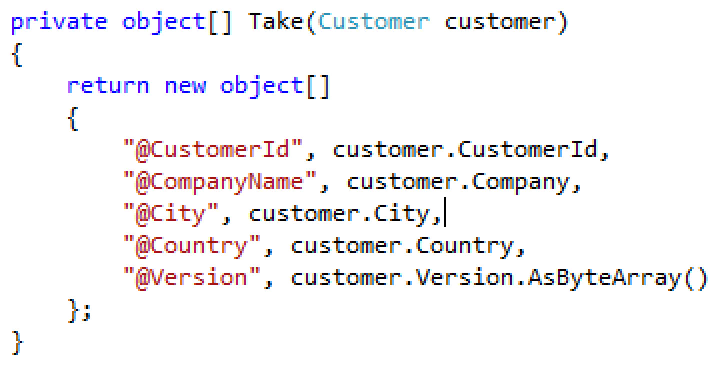
Another extension method is SetParameters. It iterates over parameter name/value pairs and creates DbParameter objects that are assigned to the Command Parameters collection. Again, if you choose to use DbParameters directly in your Dao classes then this is entirely valid and possible. In that case you would not need the SetParameters extension method.

A second set of extension methods is available in Extensions.cs under the \Shared folder. This contains several conversion methods that are very helpful in getting data out of the DbDataReader class and converts these into the proper types in the Business objects. To see an example, open up SqlServerCustomerDao and find the Make lambda expression. In it, you will see the AsId(), AsString(), and AsBase64String() extension methods in action. There is an extension method for each data type used in the application.



```
private static Func<IDataReader, Customer> Make = reader =>
    new Customer
    {
        CustomerId = reader["CustomerId"].AsId(),
        Company = reader["CompanyName"].AsString(),
        City = reader["City"].AsString(),
        Country = reader["Country"].AsString(),
        Version = reader["Version"].AsBase64String()
    };
```

Now scroll down a bit and you will see a Take method that extracts property values from the business object and adds these to an object array. This is the counterpart of Make which extracts data from the datareader and adds these to the properties of the business object. In *Pattern in Action 4.0* you will see that each Dao implementation has exactly one Make and one Take. The Take is executed in the Dao itself, whereas the Make executes in the Db class. Here is the code for the Take method:



```
private object[] Take(Customer customer)
{
    return new object[]
    {
        "@CustomerId", customer.CustomerId,
        "@CompanyName", customer.Company,
        "@City", customer.City,
        "@Country", customer.Country,
        "@Version", customer.Version.AsByteArray()
    };
}
```

Some of our users asked how the static **Db** class can be used to support multiple database connections. In situations like this you could simply duplicate the Db class, say DbX and DbY, each of which would read their own connectionStringX and connectionStringY from web.config. This works well, except that you end up with duplicate code which feels like a code smell.

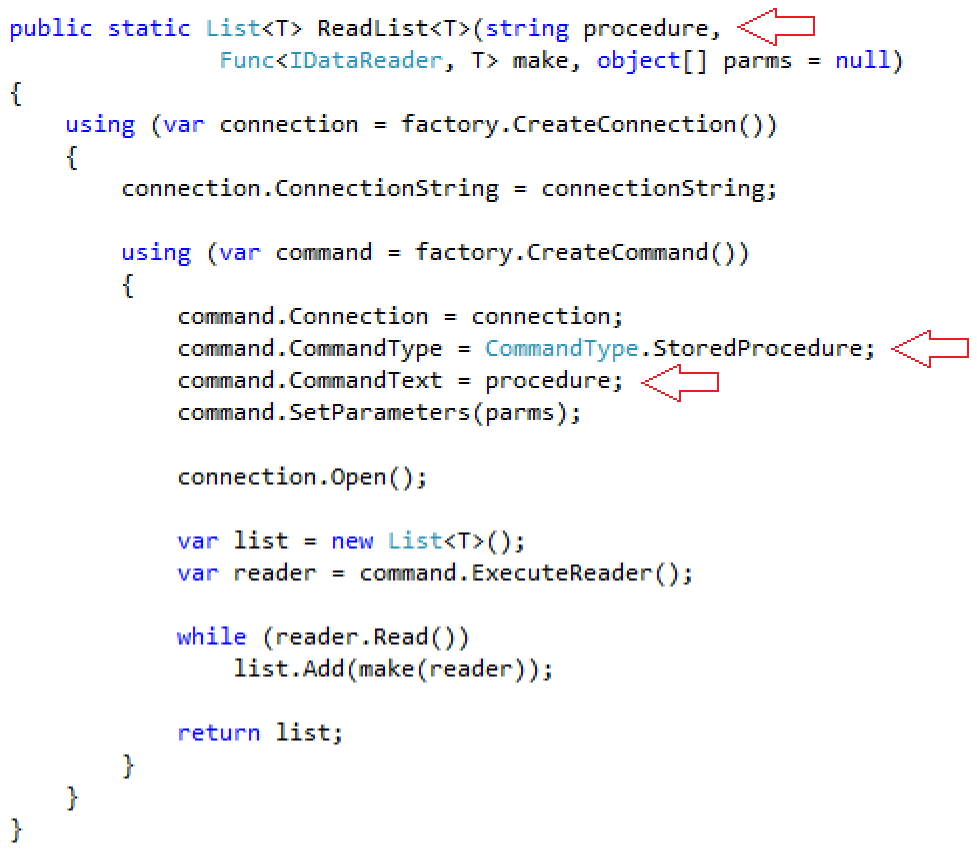
A better solution would be to get one of our trusted GoF patterns involved, specifically the Adapter pattern. Let's say we need to access 2 different databases. Here is how to approach this:

1) Change all methods in Db.cs to accept a connectionstring argument. Use this argument to set the connectionstring on the Connection object. Remove the two connectionstring related fields located at the top of this class.
2) Create two new static classes DbAdapter1 and DbAdapter2. These are wrappers, or Adapters, around the Db.cs class. Their primary purpose is to read the appropriate connectionstring once from web.config and store it statically. Then pass the stored connectionstring, which is database specific, into each Db method call.

We have included a sample file name DbAdapter.cs to demonstrate how to do this.

The Data Layer uses several commonly used design patterns, including, Factory, Singleton, and the Provider pattern. Finally, notice the significance of interfaces and abstract classes to these patterns. When working with Design Patterns you rarely see classes that are designed without interfaces, abstract base classes, and/or inheritance.

Another question that has come up frequently is how to use **Db** with stored procedures. Simply replace the sql string with a procedure string (the stored procedure name) and set the command type in the command object to *StoredProcedure*. Here is some skeleton code to get you started.



```
public static List<T> ReadList<T>(string procedure,
                 Func<IDataReader, T> make, object[] parms = null)
{
    using (var connection = factory.CreateConnection())
    {
        connection.ConnectionString = connectionString;

        using (var command = factory.CreateCommand())
        {
            command.Connection = connection;
            command.CommandType = CommandType.StoredProcedure;
            command.CommandText = procedure;
            command.SetParameters(parms);

            connection.Open();

            var list = new List<T>();
            var reader = command.ExecuteReader();

            while (reader.Read())
                list.Add(make(reader));

            return list;
        }
    }
}
```

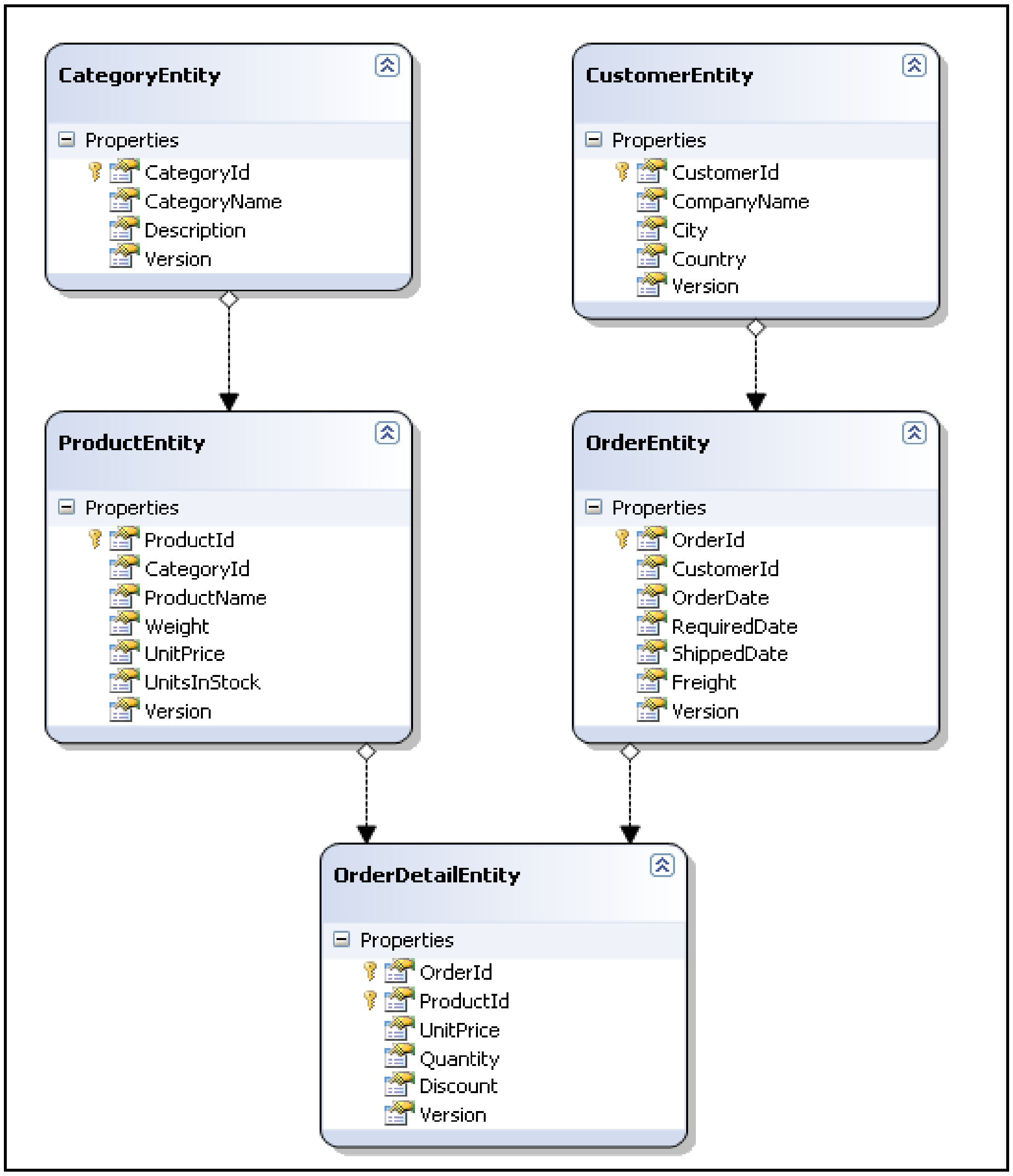
LINQ-to-SQL as the data access platform

LINQ (Language INtegrated Query) was introduced in .NET 3.5. It adds data querying and iteration services to all .NET languages. Its syntax is similar to that of SQL.

LINQ-to-SQL is an object relational mapping that is used to effectively query databases, but it also supports the ability to insert/update/delete data back to the database. Transactions, views, and stored procedures are all fully supported by LINQ-to-SQL. Currently, LINQ-to-SQL is only available for SQL Server and SQL Express.

Classes that support LINQ-to-SQL data access are found under folder \LinqToSql. Just as with ADO.NET, the LINQ-to-SQL data access classes implement the five Dao interfaces: ICategoryDao, IProductDao, ICustomerDao, IOrderDao, and IOrderDetailDao. Similarly, LINQ-to-SQL also implements the IDaoFactory interface. You find the implementations under the \LinqImplementation folder.

The file Action.dbml represents the object-relational model for LINQ-to-SQL. It contains the entities that are created by dragging and dropping tables from the database onto the work-area. This modeling tool allows you to very easily and quickly create entities (similar to business objects). See figure below.



The Object Relational Modeler with *Patterns in Action 4.0* entities

One issue you will need to consider when using LINQ-to-SQL is that DataContexts are short-lived. They are created and destroyed for each method call in the service layer. When a message arrives and a database update is required, the DataContext with the original values of the record does not exist anymore and therefore LINQ–to-SQL cannot perform an update. You could re-retrieve a fresh copy of the original record from the database before the update, but that would be inefficient as it would effectively double the network traffic and database update and insert activity. LINQ-to-SQL offers a solution that requires that each record has a unique row version number or timestamp.

To support LINQ-to-SQL every table in our database has a column named Version which is of type 'timestamp'. Every entity has also a property called Version (see previous figure of the object relational modeler). Version numbers are used to support so-called *optimistic concurrency*. Optimistic concurrency is a technique that prevents two users from interfering with each other's work while editing the same database record. It is called 'optimistic' because it is hopeful (or optimistic) that no other users make changes while a record is being changed (on the UI). This way, there is no need to place locks on database records. The system can detect any concurrency issues (i.e. interference between users) by comparing the original version number against the version number in the database. If something did change, then the version numbers will be different and the application can take appropriate action by canceling the last change and informing the user.

So how does this work? A record is selected from the database including its version number and is sent to the presentation layer and rendered on a page or form. This version number needs to be stored somewhere between requests. In Web applications, this is usually ViewState. In Windows applications the version number is simply stored in the client-side business objects. Similarly, primary keys are typically stored in a business objects (in Windows) or ViewState (in ASP.NET). By the way, ASP.NET controls, such as GridView and ListView can automatically store keys for each record with the help of ViewState. Version numbers nicely fit within that same model. You simply send the primary key and version number as pairs to the page or form and the server will get it back when the user issues a post-back.

In SQL Server, the timestamp data type is binary so we need some binary-to-string conversion and back. This is implemented in the static VersionConverter class. If you use LINQ-to-SQL with timestamps in the database tables, you will use this class, or a similar one, frequently.

Another issue when using LINQ-to-SQL for web applications is how to manage DataContext objects effectively. DataContexts are fairly expensive to create and yet for every service method call they are created and destroyed (the service layer implies a stateless model which means it does not keep DataContext instances floating around between page requests).

Patterns in Action 4.0 offers a solution encoded in the DataContextFactory class. This Factory class rapidly manufactures DataContext objects by caching the Connectionstring and the MappingSource. The MappingSource is essentially an XML file that is loaded for every DataContext instance. This is expensive, but most likely the MappingSource is the same for every instance you create. Therefore, our solution is to load a MappingSource, keep it in memory and make it available to every new DataContext created by the Factory. This is a good example of a simple and useful Factory pattern implementation.

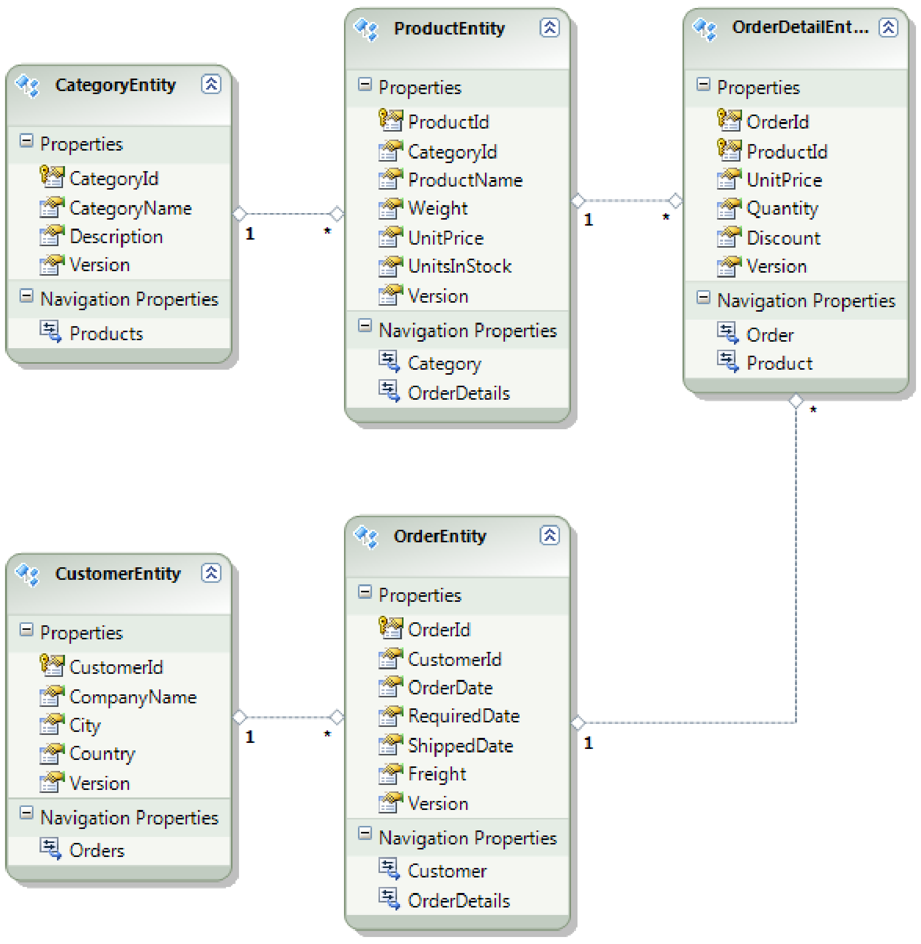
Finally, the Mapper class in the \ModelMapper folder maps entities to business objects and vice versa. In *Patterns in Action 4.0* the entities are seen as 'data receivers' that receive data from the database and pass it on to business objects. Some .NET developers using LINQ-to-SQL have decided that entities are, in fact, business objects. Indeed, LINQ-to-SQL allows the addition of custom business logic to entities and, therefore, this seems like a reasonable approach. When doing this, just be careful not to lose any business logic when regenerating entities.

Entity Framework as the data access platform

Among other things, the Entity Framework was designed to address the object-relational 'impedance mismatch', the mismatch between object model and relational data models. Microsoft is totally committed to the Entity Framework and is putting enormous resources in its development and refinement. In fact, Microsoft is 'nudging' developers away from LINQ-to-SQL and pointing to Entity Framework as the preferred data access platform for now and into the future. Design Pattern Framework 4.0 is the first release in which we include the Entity Framework.

Entity Framework data access is located under folder named \EntityFramework. Just as with the other data access platforms, Entity Framework also implements the five Dao interfaces: ICategoryDao, IProductDao, ICustomerDao, IOrderDao, and IOrderDetailDao. Similarly, Entity Framework also implements the IDaoFactory interface. You find the implementations in the \EntityFramework\Implementation folder.

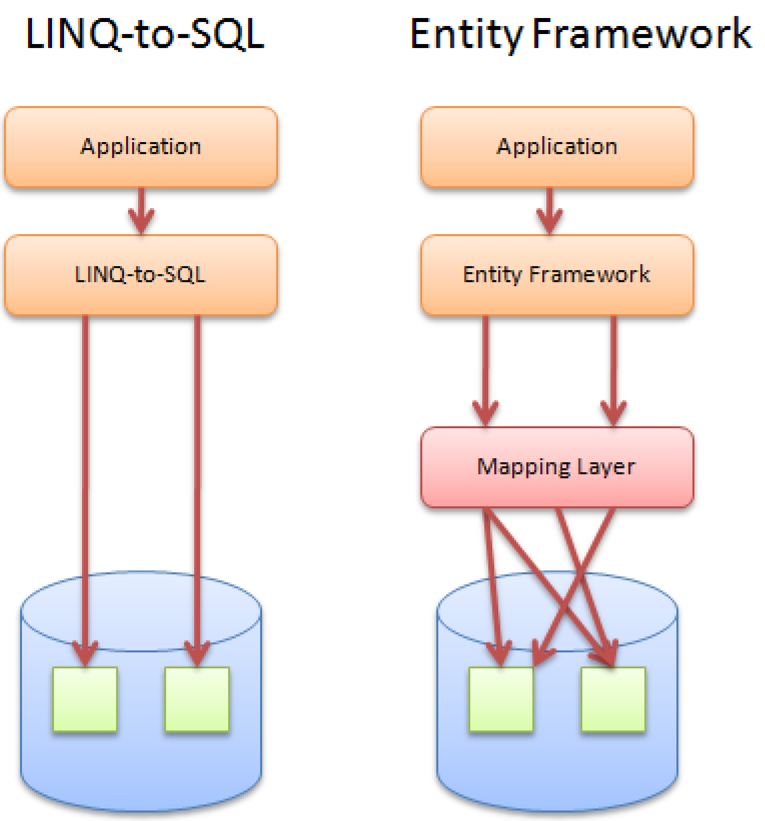
The file Action.edmx represents the object-relational model and mappings for the Entity Framework. It contains the entities that are created by dragging and dropping tables from the database onto the work-area. This modeling tool allows you to very easily and quickly create entities (similar to business objects). We appended the word Entity to entity names to avoid conflicts with the Business objects which are named similarly See figure below.



Patterns in Action 4.0 uses the DataObjectFactory class to cache the connection string (read from web.config just once) and rapidly create instances of the ActionEntities object context. Each database call uses a Factory method called CreateContext(). Just as in LINQ-to-SQL, a Mapper class exists in the \ModelMapper folder which maps entities to business objects and vice versa.

Looking at the Entity Framework implementation versus LINQ-to-SQL you may see more similarities than differences. Indeed, there are strong similarities; in particular the LINQ expressions appear pretty much the same. In reality, however, LINQ-to-SQL is only a light-weight version of the Entity Framework. It only runs against SQL Server whereas Entity Framework supports multiple data base vendors. Also, LINQ-to-SQL works best in situations where you have a relatively straightforward one-to-one mapping between business objects (perhaps with a few joins and some aggregate functions like sums, counts, etc). However, if your object model is very different from your data model (because legacy reasons, reluctant DBAs, etc), then the Entity Framework will most likely be your preferred data access platform. The Entity Framework has a mapping layer that is specifically designed for these situations in which you map the business object model to the data object model and this layer will then automatically issue the appropriate SQL to insert, update, and select statements to the appropriate tables.

Here is a pictorial overview of the differences between LINQ-to-SQL and the Entity Framework.



What this image shows is the addition of a mapping layer in the Entity Framework. This allows objects to maintain property values and/or collections that come from different places in the database. For example, a Customer object could get its data from a Party and Client table (note: Party is a common data model table that handles people, companies, departments, etc). Subsequent selects, updates, and deletes of the Customer object take all these tables into consideration and the appropriate records are automatically updated. Please note that there is a lot more to the Entity Framework and this is just a conceptual view.

Microsoft has had a long history of frequent changes to their database access technologies: DAO, ODBC, OLE DB, ADO, Jet, MDAC, and more recently ADO.NET, LINQ-to-SQL, and Entity Framework. Many organizations struggle to standardize their data access stack and today they are faced with these primary choices: ADO.NET, LINQ-to-SQL, or Entity Framework.

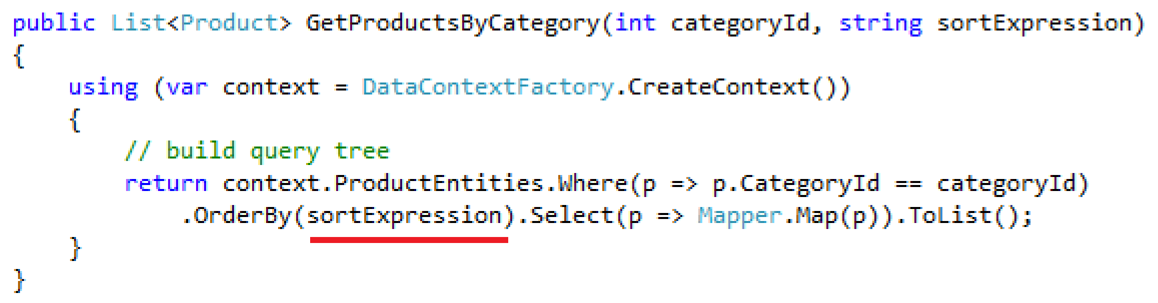
First there is ADO.NET which is a fast, effective, and proven technology. It also gives you full control over the actual SQL being issued to the database, meaning you can fully hand-optimize your queries. However, organizations that do not have strong SQL skills in-house are probably going to look at LINQ-to-SQL or Entity Framework. As we mentioned before, LINQ-to-SQL is the lightweight brother of the Entity Framework. LINQ-to-SQL's future is a bit uncertain because Microsoft has hinted that they will not continue development on this platform. Later they retracted, but it is clear that longer term the Entity Framework may be your best choice. However, we do like the 'agile, light-weight' nature of LINQ-to-SQL, so if you are starting to build a new website and need to be up and running as quickly as possible, then LINQ-to-SQL may be a good choice.

We cannot predict how the Entity Framework will evolve. But we can see that it is becoming more and more ingrained in Microsoft's long-term vision and technology stack. It would not be farfetched to see the Entity Framework become part of SQL Server, which essentially would turn SQL Server into an object-oriented data store. As a .NET developer/architect you could then focus on the object model and there would be no impedance mismatch (mismatch of object and relational model). Your access to the persistence layer (Entity Framework) would be through LINQ-to-Entities or Entity-SQL and you would basically never have to deal with SQL (although under the hood it is all SQL).

Dynamic LINQ

One of the challenges you run into when using any flavor of LINQ (LINQ-to-SQL, LINQ-to-Entities, etc), is that, at compile time, you may not know the exact query that will be requested at runtime. Let's say, you have a search page where users enter numerous criteria and multiple sort orders. This is one of those situations where you end up having to build lengthy case statements when constructing a late-bound LINQ expression.

To solve this, Microsoft has created a DynamicQuery feature that implements a subset of the LINQ extension methods (Where, OrderBy, etc) in a late-bound, string based manner. In *Patterns in Action 4.0* it is used at several places in both LINQ-to-SQL and Entity Framework. Open up class LinqProductDao to see an example. In method GetProductsByCategory we pass in a string named sortExpression which is directly used as an argument into the OrderBy dynamic extension method. This sortExpression could some something like: “ProductName Desc” (see how it is used - underlined below):



```
public List<Product> GetProductsByCategory(int categoryId, string sortExpression)
{
    using (var context = DataContextFactory.CreateContext())
    {
        // build query tree
        return context.ProductEntities.Where(p => p.CategoryId == categoryId)
            .OrderBy(sortExpression).Select(p => Mapper.Map(p)).ToList();
    }
}
```

You will find a file named DynamicQuery under the \Shared folder that supports all these dynamic string-based query features. It will be a nice addition to your personal LINQ toolset. In fact, it even ships with your copy of Visual Studio 2010. You can find it at:

For C#:

C:\Program Files\Microsoft Visual Studio 10.0\
Samples\1033\CSharpSamples.zip\LinqSamples\DynamicQuery

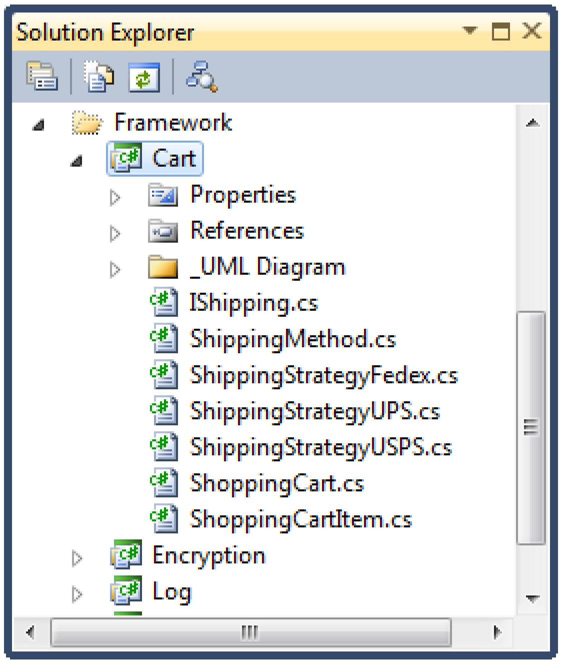
For VB:

C:\Program Files\Microsoft Visual Studio 10.0\
Samples\1033\VBSamples.zip\VB Samples\Language Samples\
LINQ Samples\DynamicQuery\DynamicQuery

Next we will review the 4 projects under the Framework folder.

Cart

Expand the Cart project until you see the view below:



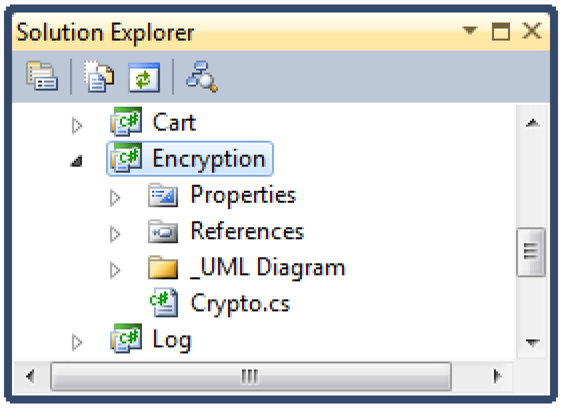
The Cart project is a class library with standard e-commerce shopping cart functionality. This cart is non-persistent, that is, it only exists for the duration of the session. So, when a user's session expires, the cart is destroyed as well. It would not be hard to make the shopping cart persistent by adding a Cart table to the database. To associate an unauthenticated user with a shopping cart you would use an anonymous identifier – most likely the shopping cart id. This identifier is then stored as a cookie on the client machine and in the Cart table. Then, when the user returns at some later time, you can re-instantiate the cart by linking the cookie value and the shopping cart.

The ShoppingCart and ShoppingCartItem classes work together to hold the cart items. ShoppingCart is the container class for the items and exposes the usual cart methods, such as, AddItem, RemoveItem, ReCalculate, ShippingStrategy, and others.

Other types in this project are involved with shipping and shipping calculations. The GoF Strategy pattern is used to facilitate a simple, but elegant, 'plug and play' model in which different shipping strategies can be swapped out for another. Supported shipping methods include Fedex, UPS, and USPS (United States Postal Services).

Encryption

Expand the Encryption project until you see the view below:

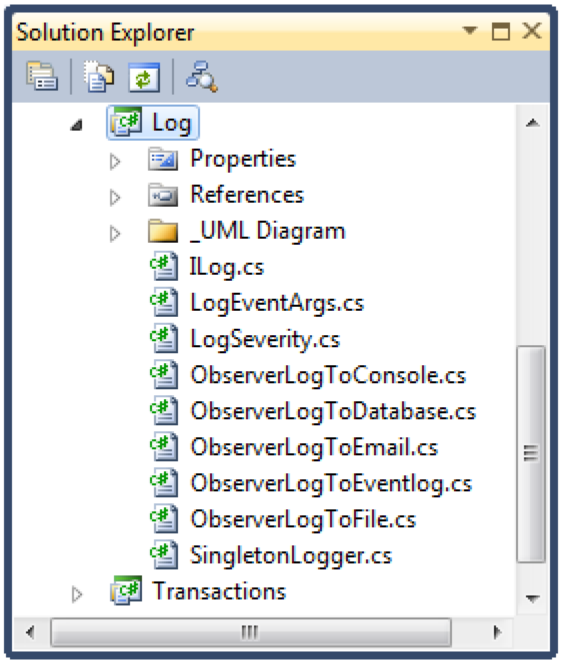


The Encryption class library has a static helper class that encrypts and decrypts strings using a hard-coded *key* and *iv* vector. It uses a *TripleDes* encryption algorithm provided by the .NET Framework. In your own projects, you should generate and use your own *key* and *iv* values and store these in a safe place (preferably outside the code).

In this release the Crypto class is not used. However, if you look at the ActionService class you will find two methods to encrypt and decrypt primary keys that are going to or coming from the service clients (the presentation tier). We decided not to implement it in *Patterns in Action 4.0* so as not to overcomplicate the application. Even so, primary key encryption is easy to add and would provide another level of security in that your clients are not able to see the true values of the database primary keys.

Log

Expand the Log project until you see the view below:



The Log project offers a logging facility that allows you to log application errors to any kind of storage or output device. The project has five sample classes that log to a database, email, event log, a file, and the console output window respectively. It would be easy to extend these to other output devices.

The Observer design pattern plays a key role in this library. It allows Observer classes to attach (subscribe) and detach (unsubscribe) to and from a central Logger (subject) and be notified about log events.

Each log event has a severity. During development / QA phases you may want to log messages that are tagged with Debug or Info severity. In production you're more likely to only log messages with a level of Warning or Error and higher.

Logging is not activated in Patterns in Action 4.0 because it requires that the application has write privileges to the output logging device. In particular when running an ASP.NET application this can be a bit tricky. But here is how you would use this facility in ASP.NET with the proper privileges.

First, notice in ASP.NET Web Forms application that in global.asax.cs (or vb) at Application_Start we initialize logging in the InitializeLogger() method. For demonstration purposes we attach 2 different loggers to listen to log events: they are: the debug console and email. In a real production system you would usually have only one output device, perhaps two, if you want to notify an administrator with emails in addition to logging it to a persistent data store.

Next, in the same file, we have a 'last resort' error handler named Application_Error. In it, we retrieve the last unhandled Exception and inform the logger that we have an error message that needs to be logged. The error will be logged by the listening devices (depending on the Severity level set in the web.config file). Usually, this is all you do and then let the <customErrors> setting in web.config determine to which page the user is redirected.

'Last resort' error handlers are a good place to log ASP.NET application errors. It limits the need for try/catch blocks throughout the application which is reasonable because meaningful exception handling within a page is often hard or impossible to do.

With or without a last resort error handler, the pattern you use within an application with exception logging is something like this:

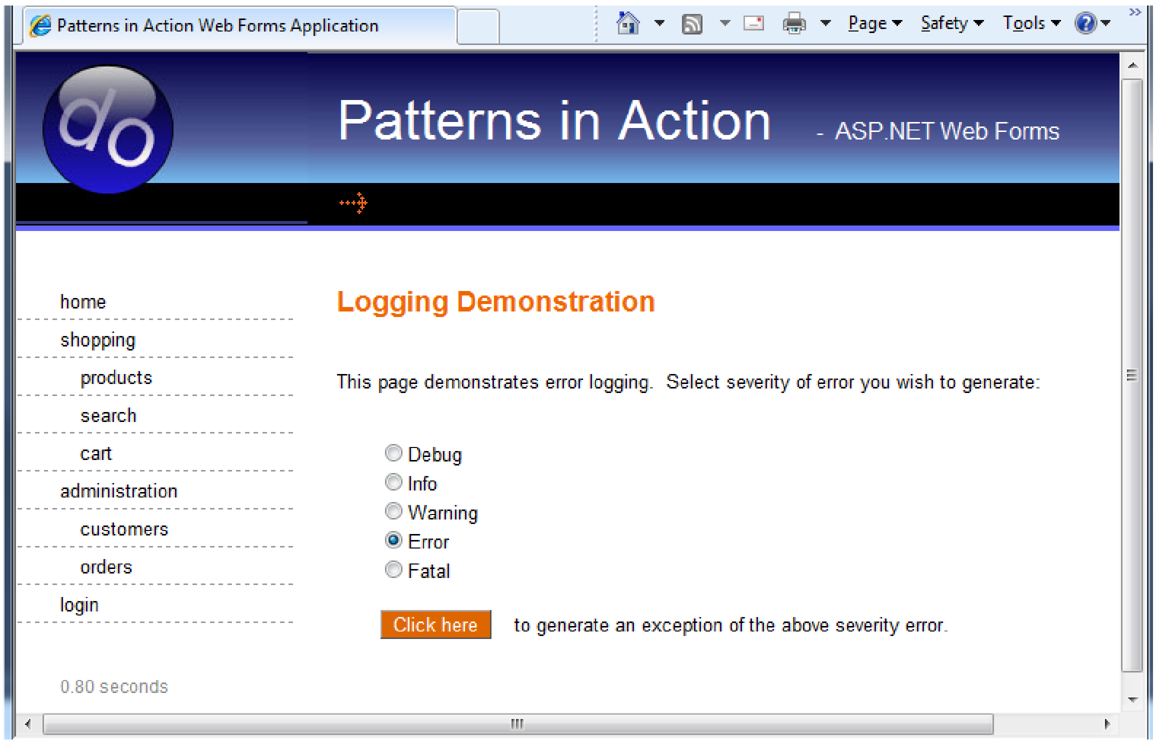
```
// C#

try
{
    int y = 0;
    int x = 10 / y;
}
catch (ApplicationException ex)
{
    SingletonLogger.Instance.Warning("Divide by zero in DoThis()");
    throw ex;
}
```

```
' VB

Try
    Dim y As Integer = 0
    Dim x As Integer = 10 / y
Catch ex As ApplicationException
    SingletonLogger.Instance.Warning("Divide by zero in DoThis()")
    Throw ex
End Try
```

In the catch block you log the exception as a Warning (or Error, or whatever level) after which you can respond to the exception or re-throw it and let the next try/catch handle it.

In the ASP.NET Web Application we have added a page named LoggingDemo.aspx that demonstrates how logging works. You access this page by directly typing the name on the command line, like so:

This will display the following demo page:



Logging Demonstration

This page demonstrates error logging. Select severity of error you wish to generate:

Debug
Info
Warning
[✓ Error]
Fatal

Click here to generate an exception of the above severity error.

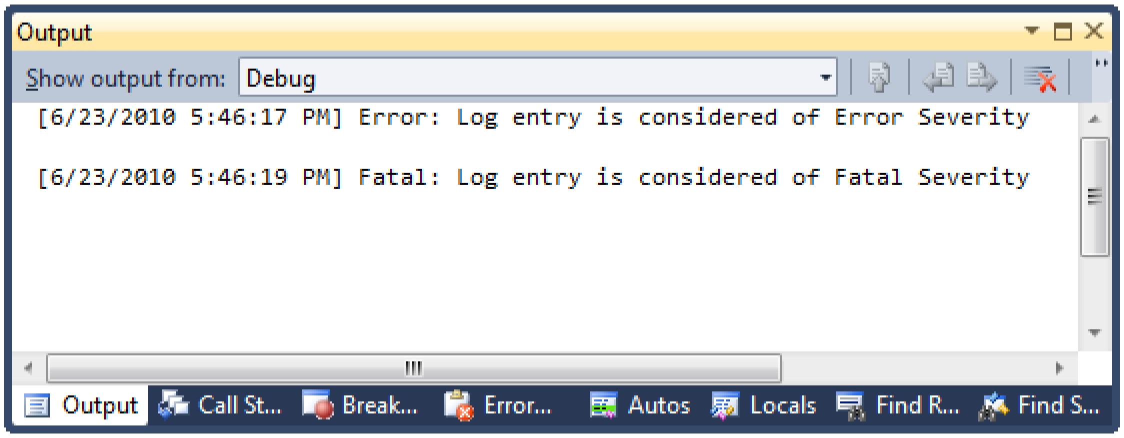
First select a radio button with the severity of the error you wish to generate. Then click the orange button. In the code behind of this page a log entry is generated of the selected severity level, that is, as a Debug entry, an Info entry, a Warning entry, an Error entry or a Fatal entry. This logging model assumes that throughout the application you call logging methods that log situations and exceptions with a certain severity code. So, if you think that a particular error only warrants Information logging you code this:

```
SingletonLogger.Instance.Info("Just to let you know that event XYZ occurred");
```

With the above code in place, the global configuration (which is set in web.config) determines <u>which log entries actually get logged</u>. Open the web.config file and see that by default we have selected 'Error' severity. This means that any lesser log entry (Debug, Info, and Warning) is not being logged, and only Error and Fatal entries are logged. So, the above Info method call is simply ignored. The assumption here is that initially in the development cycle you want to see everything and you may need many Debug and Info entries, whereas in production, when the code is stable you may only want to see Error and Fatal logging entries.

As mention above, in the global.asax file we've added two subscribers to the logger: one that logs to the console, and another that sends emails (the actual email sending part is commented out because you will need privileges to send email). However, you can run the demo page and you will see the appropriate log messages display in the Output window of Visual Studio.

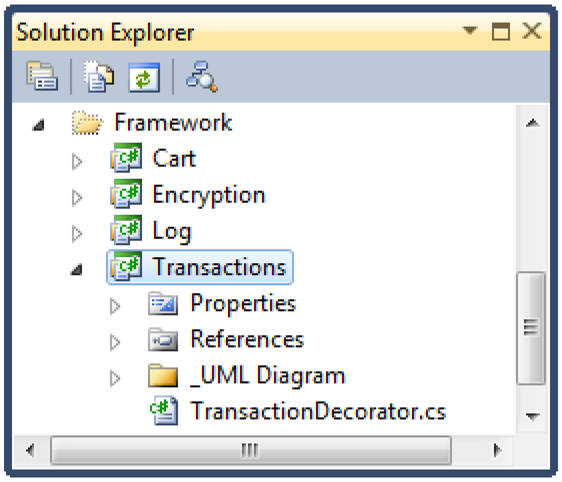
Let's run the demo page. We assume you still have the severity set to 'Error' in your web.config (the default). Then on the page select Debug severity and click the button; then Info and click the button; then Warning and click the button, all the way down to Fatal. The only messages that display are Error and Fatal, which is indeed the expected outcome. Be sure you are looking at the Output windows in Visual Studio. Below is a screenshot (ignore any other messages you may see).



```
[6/23/2010 5:46:17 PM] Error: Log entry is considered of Error Severity

[6/23/2010 5:46:19 PM] Fatal: Log entry is considered of Fatal Severity
```

Transactions

Expand the Transactions project until you see the view below:



The Transactions class library contains a class named TransactionDecorator which 'decorates' the built-in TransactionScope class using the Decorator design pattern. .NET supports a transaction framework that is available in the System.Transaction namespace. TransactionScope makes a code block transactional by implicitly enlisting data base connections in a distributed transaction.

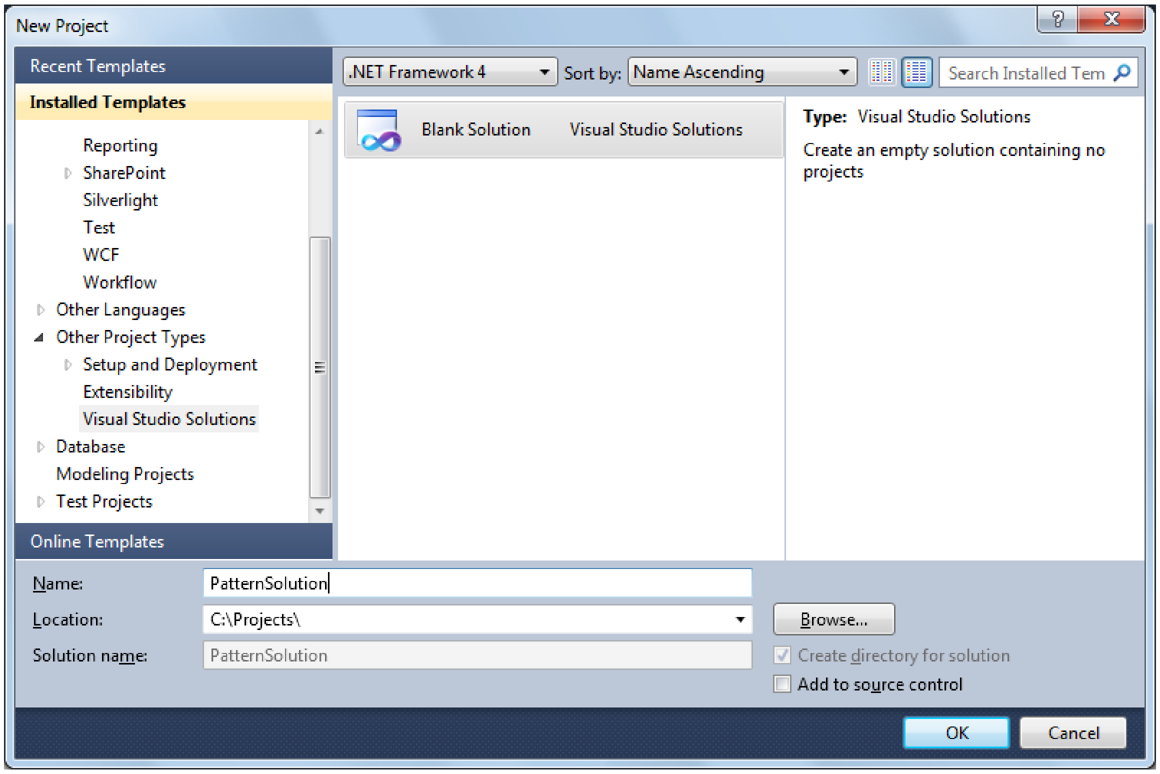
Microsoft recommends you use the *using* language block to ensure that Dispose is called on the TransactionScope object. In addition, the Complete() method must be called within the *using* block to commit the transaction. The exact same rules apply to the TransactionDecorator object.

The main purpose of the Decorator pattern is to 'embrace and extend' the functionality of the object being decorated. The TransactionDecorator embraces the TransactionScope but weeds out the MS Access TransactionScope object calls (MS Access is not supported by TransactionScope). The TransactionDecorator is not used in *Patterns in Action 4.0* because its potential for COM errors in certain environments due its dependency on MSDTC (Microsoft Distributed Transaction Coordinator).

Building your own Pattern-based .NET Solution

This section describes how to setup your own .NET 4.0 Solution using a structure similar to the one used in *Patterns in Action 4.0*. This step by step guide will get you started.

Step 1: Create a Blank Solution. Ensure your target is .NET Framework 4.0 (specified at the top).



Type: Visual Studio Solutions

Create an empty solution containing no projects

Name: PatternSolution

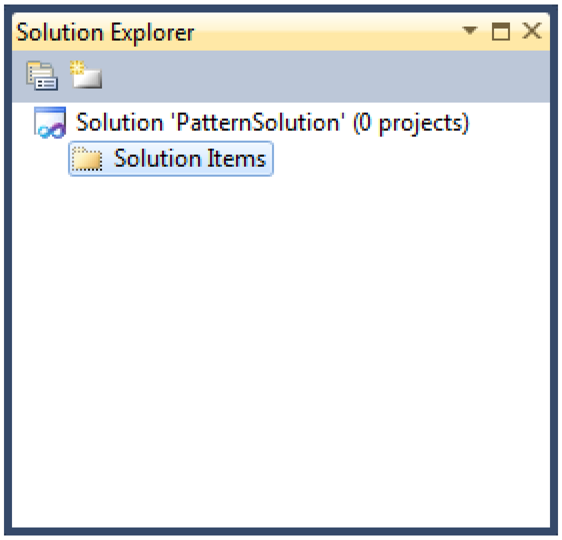
Location: C:\Projects\

Solution name: PatternSolution

[✓] Create directory for solution

[ ] Add to source control

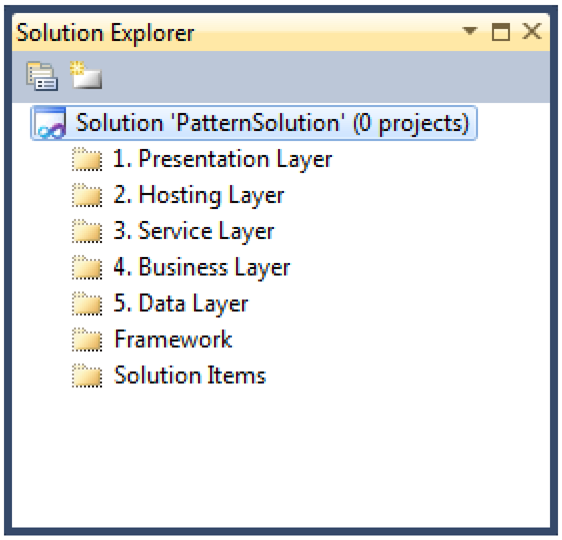
Step 2: Within this Solution immediately create a Folder named ‘Solution Items’. This will prevent any newly added project from 'taking over' the solution role. If you were to add a Web Site without this folder, it would take on the role of the Solution which is not what you want.



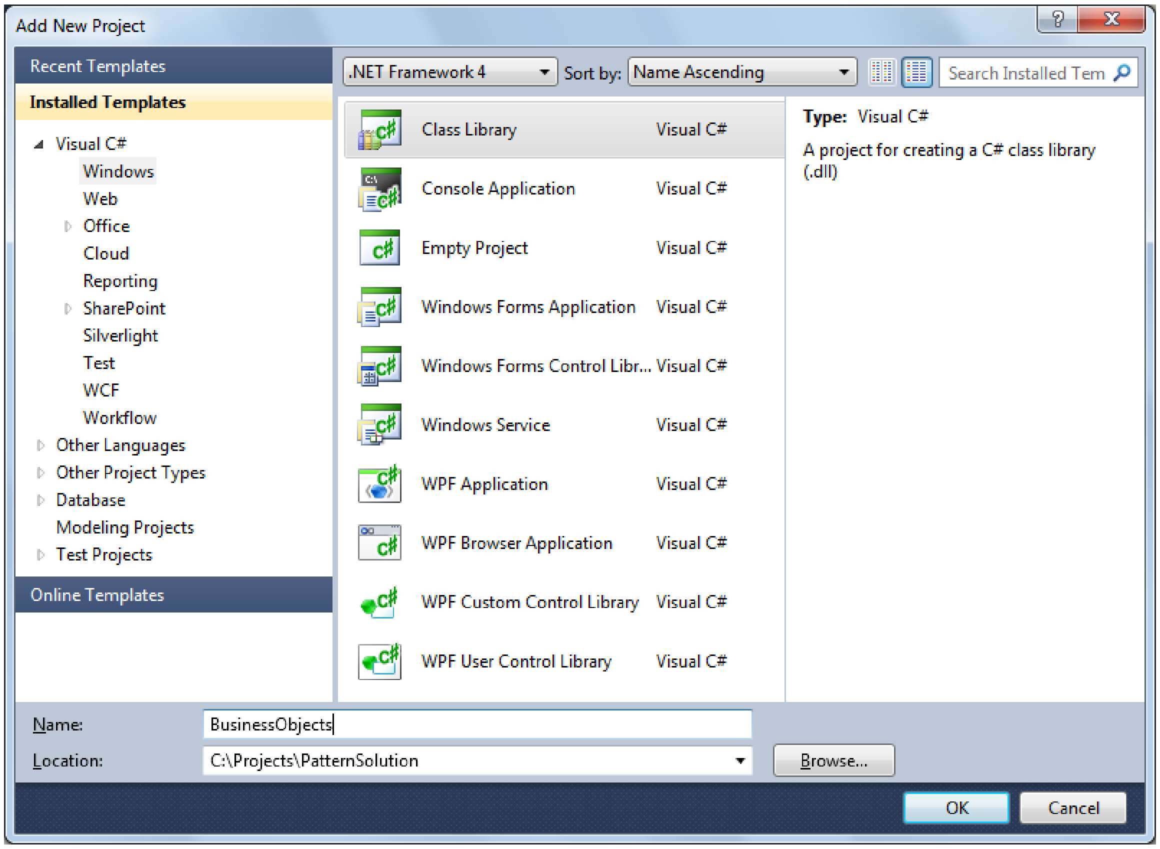
In this case, we are not adding a project just yet. First, we continue adding a fewl more folders in the next step.

Step 3: Create folders according to how you wish to arrange your projects and layers. In *Patterns in Action 4.0* the folders are *Presentation Layer, Hosting Layer, Service Layer, Business Layer*, *Data Layer* and *Framework* each with their own subfolders and projects. Prefix folder names with numbers so that they display in a logical top-to-bottom order, that is, Presentation Layer on the top, all the way down to the Data Layer at the bottom.

As mentioned before, if you are building a standalone application, such as ASP.NET, then you will probably want to reference the Service Layer assemblies directly from ASP.NET rather than using Service References to the Hosting Layer. In that case you would not have a Hosting Layer folder.



Step 4: Create class library projects according to your needs. In *Patterns in Action 4.0* we have 1 project under Business Layer, 1 project under Data Layer, 4 projects under Framework, and 2 projects under Services.



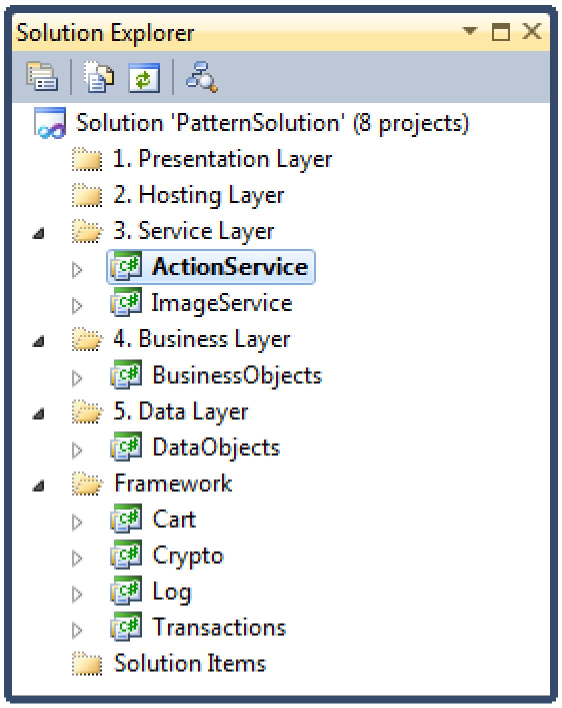
Type: Visual C#

A project for creating a C# class library (.dll)

Name: BusinessObjects

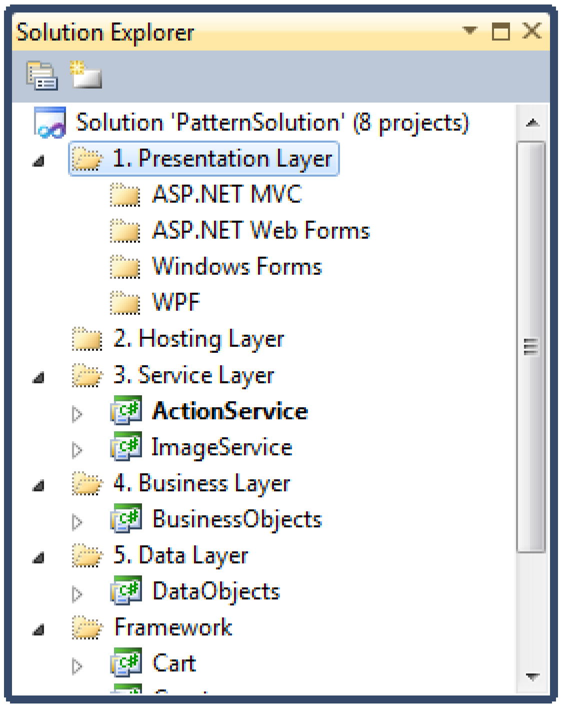
Location: C:\Projects\PatternSolution

The solution now looks like this:



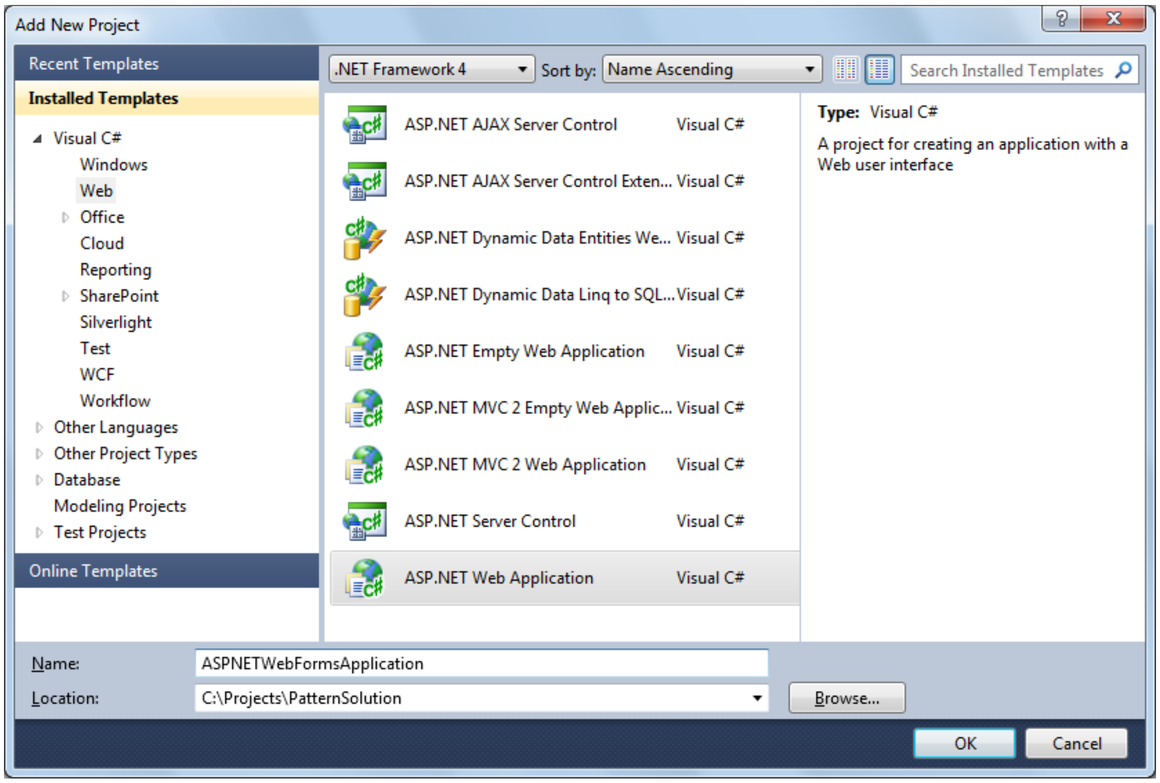
```
Solution 'PatternSolution' (8 projects)
    1. Presentation Layer
    2. Hosting Layer
    3. Service Layer
        ActionService
        ImageService
    4. Business Layer
        BusinessObjects
    5. Data Layer
        DataObjects
    Framework
        Cart
        Crypto
        Log
        Transactions
    Solution Items
```

Step 5: Next, add a folder under Presentation Layer for each of the clients you are planning to develop. *Patterns in Action 4.0* uses four folders named: *ASP.NET MVC, ASP.NET Web Forms, Windows Forms*, and *WPF*. In case you're building just one application (for example, ASP.NET Application), simply place your client project directly in the Presentation Layer folder and skip the subfolder.



Step 6: Then add projects of the appropriate type to the respective folders, for example, ASPNETMVCApplication and ASPNETApplication.Test in the MVC folder, etc. Each type has its own set of supporting class libraries, for example, Windows Forms has WindowsFormsModel, WindowsFormsPresenter, and WindowsFormsView, etc.

Note: at this stage we do not usually include spaces in the projects names. This will prevent getting underscores ('_') in the namespace names. In fact, Visual Studio's default projects names also do not contain spaces. In the last step we will show how to adjust the project names.



.NET Framework 4 Sort by: Name Ascending

ASP.NET AJAX Server Control	Visual C#
ASP.NET AJAX Server Control Exten...	Visual C#
ASP.NET Dynamic Data Entities We...	Visual C#
ASP.NET Dynamic Data Linq to SQL...	Visual C#
ASP.NET Empty Web Application	Visual C#
ASP.NET MVC 2 Empty Web Applic...	Visual C#
ASP.NET MVC 2 Web Application	Visual C#
ASP.NET Server Control	Visual C#
ASP.NET Web Application	Visual C#

Type: Visual C#

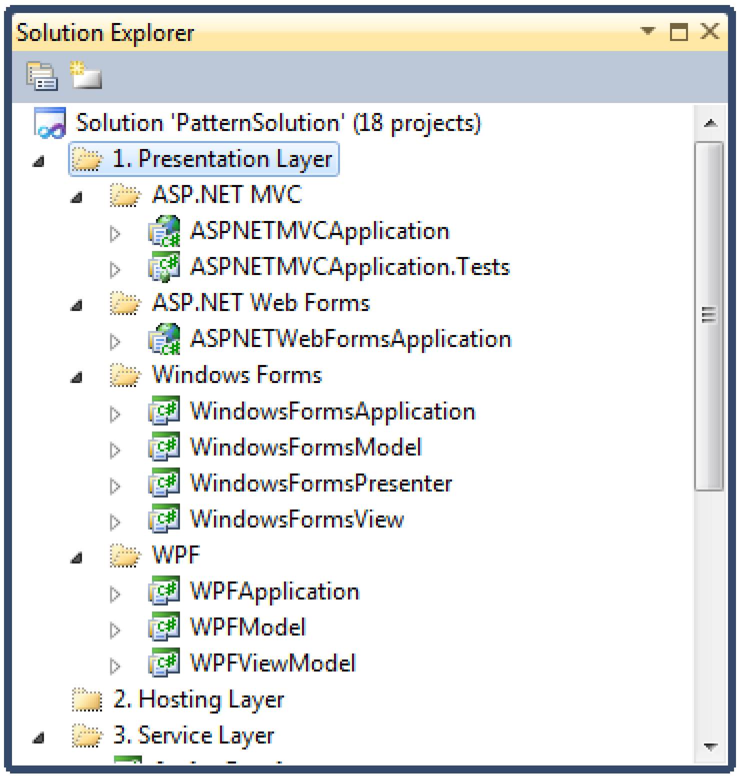
A project for creating an application with a Web user interface

Name: ASPNETWebFormsApplication

Location: C:\Projects\PatternSolution

<span style="color:red">Tip</span>: notice that in the dialog above you don't actually see in which folder the project will be placed -- basically what you see in the Solution Explorer is different from the way projects and folders are physically organized on your drive. As long as you right clicked on the proper Solution Explorer folder, the project will arrive at the desired location.

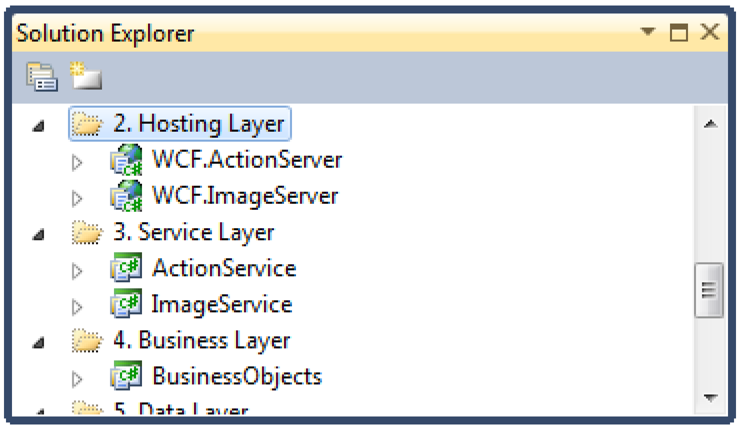
This is what the solution looks like:



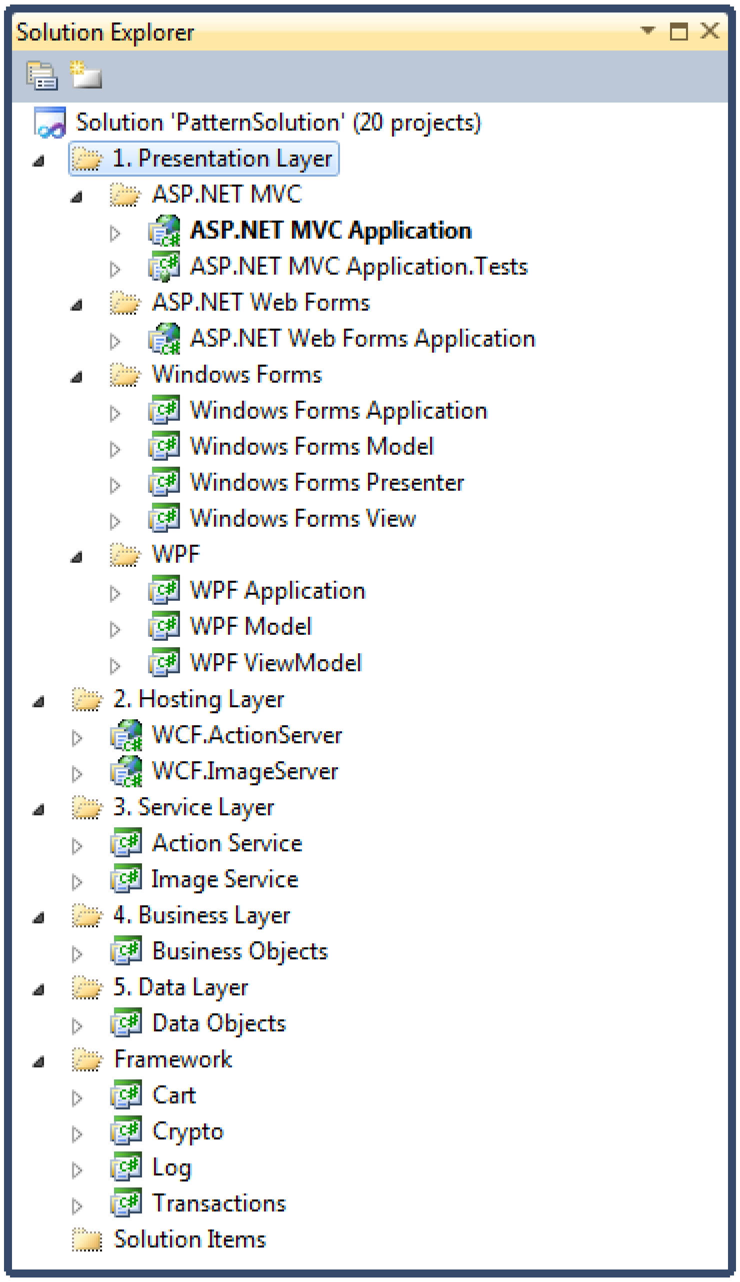
Solution Explorer

Solution 'PatternSolution' (18 projects)
- 1. Presentation Layer
  - ASP.NET MVC
    - ASPNETMVCApplication
    - ASPNETMVCApplication.Tests
  - ASP.NET Web Forms
    - ASPNETWebFormsApplication
  - Windows Forms
    - WindowsFormsApplication
    - WindowsFormsModel
    - WindowsFormsPresenter
    - WindowsFormsView
  - WPF
    - WPFApplication
    - WPFModel
    - WPFViewModel
- 2. Hosting Layer
- 3. Service Layer

Step 7: Next, create WCF Service Application projects according to your needs. Place it in the hosting folder. We added two services. Your solution will look like this:



Step 8: Finally, we will improve the legibility of project names by inserting spaces where appropriate. For example, we changed ‘ASPNETMVCApplication’ to ‘ASP.NET MVC Application’, and ‘DataObjects’ to ‘Data Objects’.   See below for a well-organized solution.



```
Solution 'PatternSolution' (20 projects)
  1. Presentation Layer
    ASP.NET MVC
      ASP.NET MVC Application
      ASP.NET MVC Application.Tests
    ASP.NET Web Forms
      ASP.NET Web Forms Application
    Windows Forms
      Windows Forms Application
      Windows Forms Model
      Windows Forms Presenter
      Windows Forms View
    WPF
      WPF Application
      WPF Model
      WPF ViewModel
  2. Hosting Layer
    WCF.ActionServer
    WCF.ImageServer
  3. Service Layer
    Action Service
    Image Service
  4. Business Layer
    Business Objects
  5. Data Layer
    Data Objects
  Framework
    Cart
    Crypto
    Log
    Transactions
  Solution Items
```

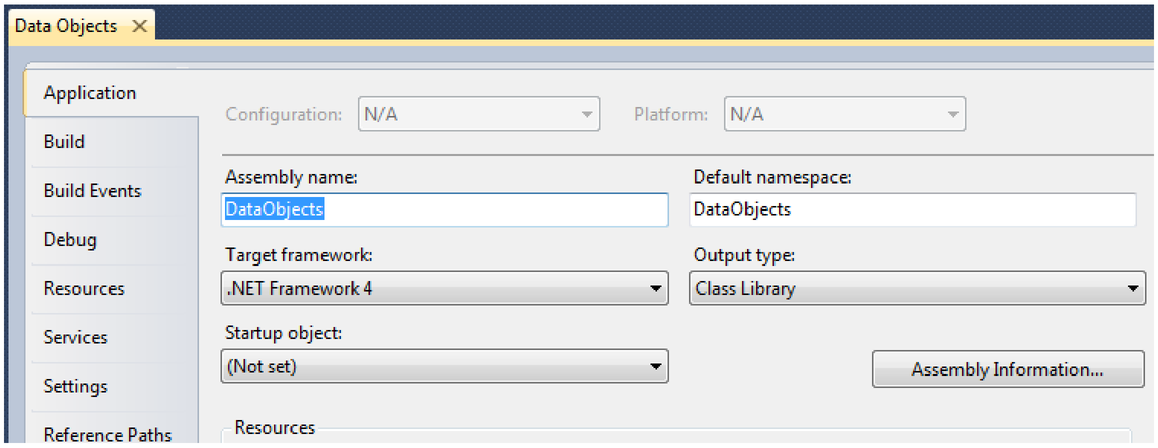
Note: when renaming a project, the project’s namespace remains unchanged. For example, when adding a new Customer class to the renamed project ‘Data Objects’, the namespace still reads: `namespace DataObjects` (one word). We like this feature because it allows you to make projects more human readable.

```
namespace DataObjects
{
    class Customer
    {
    }
}
```

If, at this stage, you made a mistake in naming your project items, don't worry, because you still have complete control, including:

1) Project name
2) Namespace name
3) Assembly name

Simply right click on the project, select Properties, and on the first tab you can configure your naming options as needed. See below.



Configuration: N/A
Platform: N/A
Assembly name: DataObjects
Default namespace: DataObjects
Target framework: .NET Framework 4
Output type: Class Library
Startup object: (Not set)

It is best to establish your naming conventions upfront and not change them in the middle of development.

Design Patterns and Best Practices

Patterns in Action 4.0 lets you explore how design patterns are used in a real-world e-commerce scenario. The design patterns and associated best practices fall into four broad categories:

1) Gang of Four (GoF) Design Patterns,
2) Fowler's Enterprise Design Patterns,
3) SOA and Messaging Design Patterns, and
4) Model-View Design Patterns.

Next, we'll review each of these categories.

Gang of Four Design Patterns

Design Patterns were first 'hatched' in the mid 90's when object-oriented languages began to take hold. In 1997, the Gang of Four (GoF) published their seminal work called "*Design Patterns, Elements of Reusable Object-Oriented Software*" in which they describe 23 different design patterns. These patterns are still highly relevant today, but time has proven that some patterns are more relevant to real-world application development than others.

There is a group of GoF patterns that are critical to the success of many enterprise level business applications. These include Factory, Proxy, Singleton, Façade, and Strategy. Many well-designed, mission-critical applications make extensive use of these patterns. Experienced .NET developers and architects use their names as part of their vocabulary. They may say things like: *this class is a stateless Singleton Proxy*, or *here we have a Singleton Factory of Data Provider Factories*. This may seem intimidating at first, but once you're familiar with the basics of these patterns, they become second nature. As a .NET architect, you are expected to be familiar with these terms.

Another group of patterns is more applicable to specialized, niche-type applications. The Flyweight pattern is an example of this: it is used primarily in word processors and graphical editors. Similarly, the Interpreter pattern is valuable for building scripting parsers, but it has limited value to business applications. Both Flyweight and Interpreter are highly specialized design patterns.

Several patterns have proven so immensely useful that they ended up in programming languages themselves. Examples are Iterator and Observer. The *foreach* (*For Each* in VB) language construct is an implementation of the Iterator pattern. In fact, LINQ is almost entirely designed around the Iterator pattern. Similarly, .NET events and delegates are an implementation of the Observer pattern. These examples show just how pervasive design patterns have become in everyday programming.

The majority of the GoF patterns falls into a category that is important but at a more granular and localized level (unlike the application level architecture patterns such as Façade and Factory). They are used in more specialized and focused circumstances. Examples include: State, Decorator, Builder, Prototype, and Template. The State pattern, for example, is used when you have clearly defined state transitions, such as a credit card application process that goes through a sequence of steps. The Decorator is used for extending the functionality of an existing class. The Template is used to provide a way to defer implementation details to a derived class while leaving the general algorithmic steps. Again, they are frequently used, but at a more detailed and focused level within the application.

Finally, there is a small group of patterns that are rarely used. These include the Visitor and Memento design patterns.

A note about the Factory pattern: the GoF patterns contain two Factory patterns; Abstract Factory and Factory Method. The Abstract Factory pattern is essentially a generalization of the Factory Method as it creates families of related classes that interact with each other in predictable ways. Over time the differences between the two have become blurry and developers usually just mention the Factory pattern, meaning a class that manufactures objects that share a common interface or base class. These manufactured objects may or may not interact with each other. In *Patterns in Action 4.0* we have also adopted this usage and refer to it simply as the Factory pattern.

Microsoft introduced the Provider design pattern in .NET 2.0. Although it is not a GoF pattern (it was not part of the original 23 patterns), it is included here because you'll find it used throughout the .NET Framework itself. The Provider design pattern is essentially a blending of three GoF design patterns. Functionally, it is very close to the Strategy design pattern, but it makes extensive use of the Factory and Singleton patterns.

The *Patterns in Action 4.0* reference application demonstrates where and how patterns are used in the real world. It includes only the most relevant and the most frequently used patterns. We could have crammed all 23 GoF patterns in the application, but that would have skewed reality by not reflecting the real-world usage of design patterns.

The table below summarizes the GoF patterns used in the application, their location, and their usage. These patterns are referenced also in the code comments. A good way to study these is to have the source code side-by-side with the 69 GoF Design Pattern Projects that are part of the *Design Pattern Framework 4.0*.

GoF Design Pattern	**Project**	**Class**	**Usage**
Proxy	BusinessObjects	ProxyList	Used as base class for proxy objects representing a list of items.
Façade	Façade	CustomerFacade ProductFacade	Used as the façade (or service interface) into the business layer. All communication to the business layer goes through the façade.
Proxy	Façade	ProxyForOrderDetails ProxyForOders	Used as proxies for list of orders and list of order details. A way to limit database access to what is absolutely necessary.
Factory	DataObjects	DaoFactories DaoFactory	Used in the manufacture of other classes. Each database (MS Access, SQL Server, Oracle) has its own Factory which creates database specific data access classes.
Proxy	DataObjects	DataAccess	Used as easy data access interface for the business layer.

Singleton	DataObjects	Db	Used for low level data access. Only one stateless Db object is required for the entire application.
Strategy	Cart	IShipping ShippingStrategyFedex ShippingStrategyUPS ShippingStrategyUSPS	Used to selectively choose a different shipping method that computes shipping costs.
Composite	Controls	MenuComposite MenuCompositeItem	Used to represent the hierarchical tree structure of the menu.
Observer	Log	ObserverLogToDatabase ObserverLogToEmail ObserverLogToEventLog ObserverLogToFile	Used to 'listen to' error events and log messages to a logging output device, such as email, event log, etc.
Decorator	Transactions	TransactionDecorator	Used to 'embrace and extend' the functionality of the built-in TransactionScope class.
Provider (Microsoft Pattern)	ViewState	ViewStateProviderCache ViewStateProviderGlobal ViewStateProviderSession	Used to build several providers that can hold ViewState on the server and therefore limit the size of the pages being sent to the browser.
Singleton	ViewState	GlobalViewStateSingleton	Used to hold all available view state providers.
Iterator	All Projects	foreach language construct (For Each in VB)	Iterator is built in the .NET programming languages.
Observer	All Projects	.NET event model	Observer is built in the .NET programming languages.

Enterprise Design Patterns

In 2003, Martin Fowler published a book titled: "*Patterns of Enterprise Application Architecture*". It was written for both Java and .NET developers, but there is a slant towards the Java side of things. This book provides an extensive catalog of patterns and best practices used in developing data driven enterprise-level applications. The word 'pattern' is used more loosely in this book. It is best to think about these patterns as best practice solutions to commonly occurring enterprise application problems. It proposes a layered architecture in which presentation, domain model, and data source make up the three principal layers. This layering exactly matches the 3-tier model employed in *Patterns in Action 4.0*.

At first, many of the Enterprise patterns in this book may seem trivial and irrelevant to .NET developers. The reason is that the .NET Framework has many of these Enterprise patterns built-in which shields .NET developers from having to write any code that implement these. In fact, .NET developers do not even have to know that there is a common pattern or best practice underlying the feature under consideration. There are many of these seemingly trivial patterns; examples include Client Session State (this is the .NET Session object), Record Set (this is the .NET DataTable), and the Page Controller (pages that derive from the Page class -- i.e. the code behind)

Even so, Fowler's book is useful in that it offers a clear catalog of patterns for developers of large and complex enterprise level applications. Important also is that it provides a consistent set of names for patterns and practices which makes discussing them easier. For example, when someone on your team starts talking about the Domain Model, a Data Mapper, or Lazy Loading, then you know immediately what they are talking about.

The Repository pattern is an Enterprise pattern that we added to *Patterns in Action 4.0*. Repository is defined as a group of classes where query construction takes place. Essentially, the Repository pattern allows you to create a clean and consistent query interface and in many cases it helps reduce code duplication. In *Patterns in Action 4.0* we use a common ancestor interface IRepository<T> which is then implemented by all Repository classes.

For testing purposes the Repository pattern is also very beneficial. Repositories provide access to the database and by *mocking* (mimicking) these Repository objects (using their interfaces) we can perform tests without the need to access a real (and slow) database. This mocking of repositories has been demonstrated in the MVC Test project in this package.

Below is a summary of Enterprise patterns in *Patterns in Action 4.0.*

Enterprise Design Patterns	**Project**	**Class**	**Usage**
Repository	ASP.NET Web Forms, ASP.NET MVC	IRepository,CatalogRepository, ProductRepository, and more..	Mediates between business objects (domain) and data access layers.
Domain Model	Business Objects	Catalog, Product, Customer, Order, Order Detail	Business Objects is essentially a different name for Domain model.
Identity Field	Business Objects	Category, Product, Customer, Order	The identity value is the only link between the database record and the business object.
Remote Façade	Action Service	ActionService	The API is course grained because it deals with business objects rather than attribute values.
Service Layer	Action Service, Image Service	ActionService ImagesService	A service layer that sits on top of the Domain model (business objects)
LazyLoad	ASP.NET Web Application	ControllerBase	LazyLoads the service client only when being asked for it. Once loaded its reference will be maintained.
Transaction Script	Action Service	ActionService	The Façade API is course grained and each method handles indivual presentation layer requests.
Transform View	Controls	MenuComposite	Menu items are processed and then transformed into HTML
LazyLoad	ViewState	ViewStateProviderService	ViewState Providers are loaded only when really necessary
Page	ASP.NET Web	AuthenticationController,	Several controller classes

Controller	application	CartController, CustomerController, ProductController, OrderController	exists that provide data to and from the web pages.
Data Transfer Object	Action Service	CustomerTransferObject OrderTransferObject OrderDetailTransferObject	Provides a way to transfer data between processes. These are objects that only hold data; they don't have behavior (properties or methods).
Data Mapper	Entity Framework	Entity Framework has a dedicated Mapper layer.	Provides a way to isolate the object model from the details of the data access in the database.

SOA and Messaging Design Patterns

The *Patterns in Action 4.0* reference application contains a set of design patterns that are applicable to the area of SOA (Service Oriented Architecture). We refer to these as SOA and Messaging Design Patterns.

Many articles have been written about the exact meaning of SOA, but at its core SOA is a way to integrate and aggregate applications from one or more autonomous service systems. The *Patterns in Action 4.0* application includes several SOA and messaging design patterns and best practices.

SOA is all about sending messages back and forth and sharing functional components across an organization, or across the world. SOA offers a way to build applications as a set of services that are accessible through a message based interface. Web Services using XML and SOAP protocols transported over HTTP frequently play a central role in this architecture. However, with WCF, Microsoft has unified several communication technologies under a single umbrella and provides a way to declaratively expose services using different protocols, channels, and message formats.

Patterns In Action 4.0 demonstrates how the 3-tier model with Façade and other design patterns provide an excellent foundation to building Service Oriented Architectures with WCF Services. Using a WCF, you have the ability to expose and publish the functionality of an application to the Internet (or elsewhere) in a platform independent manner. The consumers (or clients) of your Service can be any device or platform that understands the exposed WCF endpoint configuration.

When building SOAs you will run into GoF design patterns that are built-in the WCF Services (Proxy, Command, and Observer come to mind). However, these are not discussed in this section. The focus here is on some of the patterns and best practices that are designed to address SOA-specific challenges.

So, what are these SOA-specific challenges? Actually, there are many. An important issue is that services are subject to network latency and network failure. Furthermore, calling over remote connections is computationally expensive which results in slower response times. Web service API designers need to be aware of these and other issues.

Another important consideration when designing the API is that services are built around contracts (i.e. the service interface). Changing the interface may cause problems for existing clients. Next, a more in-depth look follows at these service contracts and related topics.

SOA best practice design principles

WCF Services provide a contract that defines its public interface. WSDL is used to inform the service consumer what is in the contract. The only way that service consumers interact with a service is through its public interface. Therefore, the design of the interface requires special attention. Below are some of the best practices (we prefer not to call them patterns at this point) that have evolved in the SOA space:

Keep the interface simple and small. Expose a minimum number of web methods that can handle different request scenarios. Favor ‘chunky’ interfaces over ‘chatty’ interfaces, that is, make fewer calls and pass more information with each call.

Keep your calls atomic and stateless. WCF methods should not depend on each other. Each method call should be autonomous and execute a complete unit-of-work. Calls must be stateless, meaning that a call does not leave a state behind that the next call depends on. For example, don’t do things like: call OpenResultSet and then make subsequent calls like GetNextRecord, GetLastRecord, etc. This would create a stateful contract in which a ResultSet is maintained between calls on the server.

Hide internal (private) data and processing details. Letting implementation details leak outside the interface leads to tight coupling. Tight coupling is undesirable because giving the client access to these details gives it the opportunity to control how things are done. This prevents the service from evolving over time and doing things differently.

Always consider versioning. Versioning, the process of changing the service as requirements change is a complex topic. However, you must plan for it because change is an inevitable part of any successful web service. Backward compatibility needs to be maintained (at least for some time) or else your service clients will stop functioning. Once a contract (or API) is published, it cannot be modified. Microsoft offers best practice advice on WCF data contract and service versioning. There are many issues to consider, and a discussion on these would be outside the scope of this document. Some versioning approaches that we have found on the web are listed next:

> *Amazon* uses a version number (release date) as part of it service URL. This results in the following URLs:
> www.companyname.com/service/01-12-10/ and
> www.companyname.com/service/03-08-10/ and
> www.companyname.com/service/ will always support the latest version.
>
> *Ebay* and *PayPal* include a version number and build number in their messages. Both service client and web service then are in complete agreement as to what version is being used. However, this does not solve the problem for when the API changes (method signature and/or argument type). Having messages in the form of a formatted XML string will prevent the API from changing though.
>
> It is possible to create a WCF service with a single method that accepts a single argument as an XML or comma separated string. This argument contains an encoded request with items such as action, version number, and data. An internal dispatcher (or router) then redirects the request to the appropriate programs once it reaches the web server. This model is sometimes referred to as a 'super-router'. It will always be backward compatible, despite changes in the XML schema. This may seem attractive but is generally not recommended because it renders the interface (or contract) meaningless. A contract should have clearly defined service semantics. Even so, it is an interesting idea.

Assume the worst. From a service consumer perspective, you need to take the approach that the service will be unreliable, slow, and may not even be available at times. The goal is to build reliable systems that continue to function even in the unpredictable and unreliable SOA space.

Think in terms of message end-points. There are two ways to look at WCF Services. One as objects (or components) deployed on the web, and the other as messaging end-points. This difference may seem unimportant, but it does influence how interfaces are designed. From an OO perspective the objects on the web view is weak because Web Services do not support inheritance or (true) polymorphism. Furthermore, an object view would suggest RPC type interfaces, which is not desirable because these are not very flexible. Interfaces designed around messages offer developers a more flexible model in which messages evolve over time with minimum impact on the clients. If a web service is viewed as a group of message end-points, the emphasis shifts from public methods to public messages.

Now that we have discussed some of principles and best practices, it is time to examine some SOA and Messaging patterns that are used in *Patterns in Action 4.0*.

Document-Centric Design Pattern

One of the challenges service designers face is how do define contracts that are flexible and are compliant with the SOA design principles discussed above. The answer to this is to think in terms of document-centric APIs. A document in this context is simply a unit of information, a complete package of data that represents something of value to the business.

Frequently these documents or messages are composed of smaller pieces of information. For example, *Patterns in Action 4.0* has a service method that responds with a document that contains a sorted array of Customers, but it also comes with other information relevant to the client who is requesting the document. They include a version number, an acknowledgement, a correlation Identifier, and an error message. Some of these data items are discussed later.

Document-centric services evolve easier because the actions and associated data are contained in a single, more flexible, package rather than an RPC method with hard-coded arguments.

Say, you have a web method with the following signature:

C#

```
bool SaveEmployee(string name)
```

VB

```
Function SaveEmployee(name as String) as Boolean
```

You have deployed your service and clients are happily using the service. After a while you realize that an additional argument is needed, for example an integer value for age. With the above method signature you're out of luck because the additional argument will violate the contract that you have with your service clients. Document-centric contracts are easier to evolve since all information exchange occurs within the document payload instead of a fixed and hard-to-change RPC method.

A service cannot assume or demand that clients use its methods in a certain manner or that they call methods in a pre-defined order. Each service method should be atomic and run as single transaction. This suggests that larger and more complete sets of data are exchanged. Of course, there will always be a need for long running transactions (hours or days) and for this there is another pattern: the reservation design pattern which is discussed later in this document.

Request-Response Design Pattern

Now that we agree on document-centric services, what does the method signature (or API) look like? We want to avoid interfaces in which methods have a fixed number of arguments. These 'RPC' type methods look like this:

```
boolean = Method (integer, string, double)
```

Instead, it is better to think in terms of a message based calling pattern where messages represent a complete unit of work and follow a request / response model. The model is simple: send a request message (with all the required information) and then respond with a response message (containing the entire result set and all its data). The method signature looks like this:

```
response = Method (request)
```

It is simple, consistent, easy to understand, and offers far more flexibility than standard method signatures. All web service methods in *Patterns in Action 4.0* follow this exact same request / response pattern.

Reservation Design Pattern

WCF service methods should be atomic, stateless, and independent from one another. However, there are scenarios where this is not possible. For example, what do you do if a method requires a long running business process that takes hours or days to complete?

Say, you're building a loan application system. The ApplyForLoanApproval method may require several steps, which take a couple of days. The application may have to go through a series of credit and background checks and perhaps include human intervention by a loan officer. What is needed is a loan application number, or more generically called a *reservation number*. This number will allow the client to request status or provide additional information if necessary.

In *Patterns in Action 4.0*, the response message payloads include a reservation number and a reservation expiration date. However, there are no long running transactions in *Patterns in Action 4.0* and therefore these data members are not used in our reference applications.

Idempotent Design Pattern

One of the challenges with SOA is that the service provider has virtually no control over the service client. A client could be repeating the same request multiple times and this can have adverse side effects. For example, running a financial transaction multiple times instead of just once can be bad (or good, depending who you're talking to). Another example: updating all employee salaries by 3% multiple times is probably not what you want.

The Idempotent Design Pattern is designed to address this problem. Idempotent is a term used in mathematics meaning: "Acting as if used only once, even if used multiple times". In SOA, it means that a request can be sent multiple times without adverse effect. The pattern requires that every request message be tagged with a unique request identifier (also called a unit-of-work). A good candidate for such an identifier is a GUID. The WCF Service needs to keep track of the request identifiers by maintaining a buffer of previously received request identifiers. Then, when the same request is sent multiple times it can respond accordingly. This approach ensures that a request be processed only once. Of course, the service has different options on how to respond to repeated requests; it could throw an exception, return an error, return a cached response, or it could simple run the same process again.

Mind you, that this is only a first line of defense and not a complete answer to a difficult to handle problem. Repeated requests can take different forms, such as, a 'service of denial' attack (whereby a client maliciously floods the server with an overwhelming number of requests), or a client going haywire in an incorrectly written loop, or a user on the client side who pushes the submit button too often and too fast.

To confirm receipt the service returns the *requestId* to client in a variable called *correlationId*. This is particularly important for asynchronous web request calls (which, by the way, are not included in the *Patterns in Action 4.0* reference application).

Message Router Design Pattern

As an alternative to using the name of the service method, the message payload may also contain instructions about which operations to perform (i.e. what class to invoke or what method to call). This approach of sending messages that include instructions on where to route the request is called a Message Router design pattern. Another name for this pattern is Command Message – this, of course, because it is closely related to GoF's Command pattern.

In *Patterns in Action 4.0* the Message Router design pattern is demonstrated by the SetCustomers method and its CustomerRequest message. This message contains a complete customer record together with an Action which can have any of these 3 values: *Create*, *Update*, or *Delete*. Internally, the service routes the request accordingly.

Private Identifier Design Pattern:

Instances of business objects usually have their own identity. This allows us to distinguish one object instance from another and makes it possible to locate a particular object among others. The need for a unique object identity is analogous to database records, which have unique primary key values. In most applications you will find that the business objects use the database primary key values as their identifiers.

The business layer in *Patterns in Action 4.0* has Customer, Order, Product, and Category business objects have identifiers with the same values as the surrogate keys in the associated database records (note: surrogate keys are the auto-number values in MS Access and identity values in SQL Server). These identifiers work well, because they are immutable (in the database, that is) and unique within their class. Once a primary key value is generated, it will never change.

Object identifiers are part of the data transfer objects that are transported in SOA messages. Since web services have no control over the clients, you want to protect the actual database primary key values because they may pose a security risk. For example, if the identifiers are simple integer identity values, then it does not take much for the client to start guessing valid identifiers.

The Private Identifier Design Patterns addresses this challenge by encrypting the identifiers in such a way that it guarantees that these encrypted values do not change over time. This non-changing characteristic is important because a client may choose to store these objects in a local database, make changes locally, and then perform a batch update to the server later. So, it is important that both client and service understand the encrypted identifiers and that they be immutable.

Note: to keep things simple, we decided not to include identifier encryption in Release 4.0 of *Patterns in Action*. However, the original methods are still available in project *Encryption* and *ActionService* class. The next two paragraphs apply to release 2.0 of *Patterns in Action* only.

Patterns in Action ***2.0*** uses the .NET built-in tripleDes algorithm to perform the encryption. The required *key* and *iv* vector are hard-coded in the code. In your own work, you will want to generate a different key and iv and store these at a secure place, ideally outside the code such as in the Windows registry.

Patterns in Action ***2.0*** demonstrates primary key encryption only. If foreign keys are present in your business objects, you must remember to encrypt these as well. Use the same encryption algorithm and keys as for the associated primary keys or else primary and foreign key mismatches will occur. It should be noted that foreign key values are changeable by the client, but primary key values are not. But this is similar to what you find in database operations.

Below is a summary listing of SOA patterns in *Patterns in Action 4.0.*

SOA & Messaging Design Patterns	**Project**	**Class**	**Usage**
Document Centric	Action Service	All messages	Each service method represent a business process as a complete unit of work. This requires a document-centric viewpoint. No stateful type interfaces.
Request-Response	Action Service	IActionService ActionService	All methods in the the service interface follows the same simple, yet flexible signature model: response = method (request)
Reservation	Action Service	MessageBase	A way to support long running transactions without explicit statefulness on the server.
Idempotent	Action Service	MessageBase	Tag each request with a request number and return it as a correlation identifiers in the response messages.
Message Router	Action Service	PersistType	Embed the operation to be executed by the receiver in the actual message. Also called Command design pattern.
Private Identifier	Action Service	MessageBase	Hide internal identifiers (primary key and foreign key values) from external clients. Note: not implemented in the current release of Patterns in Action.

Model-View Design Patterns

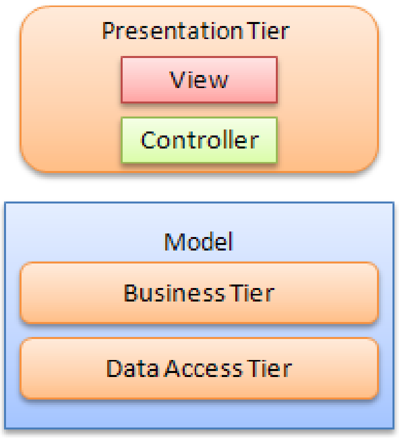
The Model-View-Controller (MVC) design pattern is one of the earliest documented patterns (in 1979). MVC remains popular today and is widely used in modern-day application architectures. In fact, Microsoft has recently added a brand new platform called ASP.NET MVC that allows developers to develop web sites around the MVC pattern.

Although MVC is widely used, it is also showing its age. In its original form MVC is seen as too rigid which inhibits its use with modern UI platforms. Consequently, several MVC-derived patterns have evolved, two of which are included in our *Patterns in Action 4.0* reference application: they are, Model-View-Presenter (MVP) and Model-View-ViewModel (MVVM). We refer to this family of related patterns as *Model-View* (MV) patterns. The three MV patterns used in *Patterns in Action 4.0* are MVC, MVP, and MVVM, one for each of the UI platforms. Next we'll explore each of these patterns.

MVC (Model-View-Controller) Design Pattern

Throughout this document, we have seen that well-designed .NET applications are usually built on 3-tier pattern architectures with Presentation, Business, and Data Access tiers. When exploring the ASP.NET MVC application we focused on the Presentation tier. As a result the relationship between MVC and the 3-tier architecture may not be so obvious. Let's review this situation.

In reality, the Presentation tier holds just two of the three MVC component types: the *Controllers* and *Views*. All tiers below the Presentation layer are collectively referred to as the *Model* -- it should be mentioned that Model is also often referred to as the Domain Model, that is, the collection of business objects in your application, but that is not the original definition. Anyhow, this may all be just semantics, but what is important is that MVC is not confused with 3-tier architecture; these are quite different concepts. Their topological relationship is depicted in the figure below.

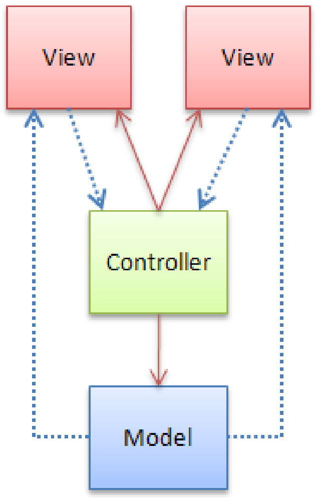


Relationship between MVC and 3-tier architecture

The traditional MVC pattern is best explained with an application like Microsoft Excel. Say, you have a comma-separated file (CSV) with daily temperature observations at the North Pole for the last 365 days. You open the file in Excel and the data (dates and temperature values) display nicely in your spreadsheet. Then you decide to create a chart and show the temperature changes over time in a line chart. You select all 365 observations and place a chart on the spreadsheet. It looks good and you're pleased with the results.

What is at work here is the MVC pattern. The data file with the temperature observations is the Model (the M in MVC). The Grid (spreadsheet) and the Chart are two separate instances of a View (the V in MVC). They are two different ways of looking at the same data (the Model). If the underlying data in the Model changes, we expect that both the Grid and the Chart will reflect this change. So, for example, if you were to change the data file and bump all temperatures up by 10 degrees for the each day in July, you would expect to see this in both the Grid and the Chart. The synchronization between the views and the model is the role of the Controller (the C in MVC). The controller coordinates changes between the Model and one or more Views.

Similarly, if a user changes the values in the Grid, you would expect the Chart to reflect this immediately (as well as the Model data). Again, it is the Controller that is notified of the change, which in turn changes the model, which then notifies the Views to update themselves with the new model values. Below is a diagram of the relationships between Model, View and Controller parts.



Model-View-Controller relationships

The solid red lines depict a direct association. The Controller maintains references to the Model and both Views. The dotted blue lines represent indirect associations (in fact, this is the Observer design pattern) in which the Views notify the Controller of any changes and the Model notifies the Views when its data has changed.

Again, using Excel as an example, the flow is usually as follows. The user makes a change in the Grid (View). This triggers an event of which the Controller is notified. The Controller gets the changed data item and applies the same change to the Model. The Model then triggers an event of which all Views are notified. The Views get the data from the Model and change their displays accordingly.

In MVC, the role of the Controller is rather limited; all it does is monitor View changes and coordinates these changes with the Model. Further down this document, you will see derivatives of the MVC pattern in which the Controller plays a larger role and is given more responsibilities.

The ASP.NET MVC Web Application in *Patterns in Action 4.0* demonstrates a modern-day implementation of the MVC pattern. Let's review the major players in this application.

First off, several controller classes manage the communication between the Views (aspx pages) and the Model (service layer and below). Four controller classes are in use: AdminController, AuthController, ShopController and HomeController.

Controllers pass data to the View via ViewData, which is dictionary that holds all the data necessary to fully render the View. ViewData contains Model objects (also called ViewModel objects) that are easy-to-read data objects that exist to support a particular view. For example, in the Shopping area you find a model (viewmodel) called CartModel. It is used to send shopping cart data to the view. It has a property called Total which is the total price for all items in the Shopping cart. You might guess that its data type is numeric, but it is not; instead it is a string formatted, $-sign and all, ready for immediate display. All the View does is insert it at the proper place on the page.

Please note that these Model objects are *not* the M in MVC, but simply helper classes that facilitate data transfer to the View. Quite often they are called ViewModel, but that name is already in use with MVVM. The MV space is really running out of unique names.

Microsoft has given its new ASP.NET platform the moniker MVC. However they have been adding refinements that make it less pure in the eyes of some pattern purists who argue that it breaks with the MVC pattern. In fact, they have a point. Let's look at an example: There are two extension methods on HtmlHelper called Html.RenderAction and Html.Action. They allow the View to call an action method on the Controller which will then render the data returned from the action method. So, we have a View that is calling the Controller, which turns the MVC model upside-down -- the Controller should be sending the View its results, but the View should not be asking for model data. That being said, we're more pragmatic and think that the RenderAction feature has its place as it allows you to build re-usable action methods that can be re-used on multiple pages. This purist vs. pragmatic discussion will probably continue for as long as MVC is around.

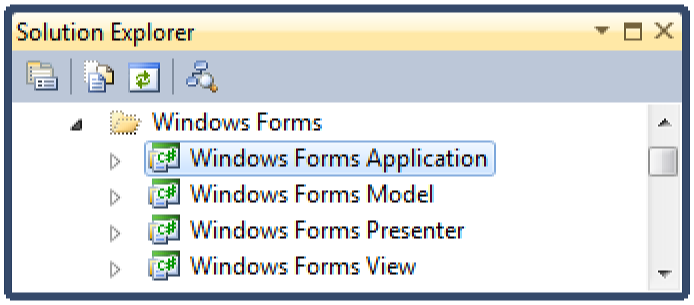
MVP (Model-View-Presenter) Design Pattern

The MVP pattern is an extension of MVC. In MVP the Controller is assigned more responsibility by giving it access to the View and its user gestures. User gestures are events created by the keyboard and mouse, such as, clicking a button, entering text, dragging an icon, etc. When a Controller can respond to gestures by directly changing elements on the View it is called a Presenter. Here is an example: a user selects a product category from a dropdown list and the Presenter responds to the SelectedIndexChanged event and updates a list of products (on the View) that are available for that category.

What are the advantages of MVP over MVC? One huge advantage is *testability*. UI systems are notoriously difficult to test using automatic testing tools. Migrating some of the logic from the UI (i.e. the View) to the Controller (i.e. Presenter) allows this logic to be included in unit- or system-tests using modern-day testing tools. Another reason developers use MVP is that the UI is usually not the right place for business logic and is better maintained at a central location. Developing an application with MVP forces the developer to think in terms of reusability which improves the modeling and coding process. It can be very difficult to find candidates for refactoring when dealing with many forms or pages with validation and other business logic all over the place. MVP helps prevent this problem from the onset.

Model-View patterns are technology independent, but certain UI frameworks lend themselves better than others to certain patterns. This is certainly the case for Windows Forms which is particularly well suited for MVP. The Windows Forms application in *Patterns in Action 4.0* uses MVP and demonstrates how you can apply this pattern.

The Windows Forms application consists of 4 projects (see image below). Their names clearly explain what part of the MVP they represent.



The Windows Forms Model project has local Business Model objects located in folder \Business Objects, they are, CustomerModel, OrderModel, and OrderDetailModel. Model objects are essentially the same as business objects in the Business Layer.

The Windows Forms Model project is the only project in this application with a Service Reference to the WCF services. IModel is the interface to the MVP Model. It is implemented by a class named Model which consumes the WCF service API.

The WCF API returns DTOs (Data Transfer Objects) and a class named Mapper (located in folder \DataTransferObjectMapper) handles the transformation from DTOs to BOs (Business Model Objects) and vice versa.

The View project is fairly easy to understand. It has an IView interface (a marker or placeholder interface with no members) from which all other View interfaces derive. For each form in the application there are one or more derived IViews. For example, FormCustomer implements ICustomerView and FormLogin implements ILoginView, but FormMain implements two interfaces: ICustomersView and IOrdersView. These interfaces contain the fields that are displayed on their respective forms. For instance, ILoginView has two data members that are read only: UserName and Password. This exactly matches the Login screen where we read the username and password provided by the user.

The Windows Forms Presenter project contains MVP Presenter classes, each of which derives from the Presenter base class. The Presenter base class maintains references to both a View and the Model. The Model is the same for every Presenter and is therefore declared as static (Shared in VB). The View is set in the Presenter's constructor. For each View there is usually a Presenter. CustomerPresenter is a fairly complete example of what an MVP Presenter does. It gets information from the Model and assigns the values to the View (Form). It then takes the values and depending on the type of action requested, it saves or deletes the currently viewed record.

MVP aims to facilitate test-driven development. When developing unit tests with MVP you build and run the tests against the methods and event handlers of the different Presenter classes. First you build a 'Mock' class for every IView interface (Mock objects act as mediators for the real objects). Our CustomerPresenter and ICustomerView are fairly simple and do not include event handlers that respond to user gestures. However, if it did, the unit test would include simulated user-gestures (events) and the test coverage of methods and event handlers of the Presenter would be very good.

MVVM (Model-View-ViewModel) Design Pattern

First of all, here we discuss MVVM in the context of the WPF application. To see MVVM with Silverlight please go to the *Silverlight Patterns 4.0* application (which is a separate solution) and associated documentation.

The Model View ViewModel (MVVM) pattern is a more recent addition to the family of MV patterns. It was first documented by Microsoft in 2005 where it evolved when they began building rich UI applications based on WPF (Windows Presentation Foundation). Expression Blend, for example, makes extensive use of the MVVM pattern. Several built-in WPF features, including *databinding* and its *commanding* architecture, make it highly suitable for MVVM.

The WPF Application in *Patterns in Action 4.0* is implemented using the MVVM pattern. Working with this pattern requires that you are familiar with WPF. Even then, it will take some time to fully understand the inner workings and how the pieces are put together. But, once you have built a couple of WPF windows using MVVM you will begin to see how it helps streamline the design of your WPF UIs.

WPF is known to be 'notoriously flexible'; for every feature you implement there are several alternative ways of accomplishing the same results. Using the MVVM pattern will assist you in following a structured path and a proven method for designing WPF applications. We have tried to keep the WPF application in *Patterns in Action 4.0* simple, yet sufficiently complete, with menus, windows, and data access, to give you a feel how this pattern is used in a WPF application.

What are the primary objects and classes involved in MVVM? As its name implies there are three main players: the Model, the View and the ViewModel. We'll look at each of them starting with the View. The View is represented by the XAML files with their code-behinds. With MVVM, the code-behind is usually small or non-existent because most of the logic ends up in the ViewModel.

Next, we examine the ViewModel. This component represents the 'Model of the View', meaning that it exposes the relevant data to the View, as well as its behavior, usually via commands. The ViewModel responds to user gestures (user input) and is very much aware of the status of the UI. You may recall that the Presenter in the MVP pattern assumes a larger role than the Controller in MVC. ViewModel goes one step further because it is totally aware of what is happening in the UI and responds accordingly. For example, in our WPF application, the ViewModel knows which customer is currently selected, it knows when all required fields for a new customer have been provided, and it knows which menu items should be enabled or not.

The third component in MVVM is the Model. The Model consists of one or more model objects (similar to business objects) that contain the data that is to be exposed to the UI (the View). These model objects implement interfaces that facilitate direct databinding to the View (the interface used is INotifyPropertyChanged). To get data from the database the model objects access a Provider class. In *Patterns in Action 4.0*, the Provider consumes the services offered by the service layer. However, in your own WPF applications you could consider bypassing the Hosting layer altogether and reference the Service assemblies directly. This will speed up performance of the application. Remember, this is the same consideration we discussed earlier about the deployment options for your ASP.NET applications.

Next, we explore the MVVM components and their projects in more detail. First of all, notice that there is no separate WPF View project. There are no separate Views as they are represented by standard WPF XAML windows. These windows reside in the WPF Application project.

The WPF ViewModel project has just two classes. The abstract CommandModel is a thin wrapper around the built-in RoutedUICommand. The CustomerViewModel class contains custom commands, based on the abstract CommandModel, that perform the basic add, edit, and delete operations. These operations are handled by 3 nested classes AddCommand, EditCommand, and DeleteCommand.

CustomerViewModel contains an ObservableCollection of customer model objects and an 'Index' (into the customer collection) representing the currently selected customer on the UI. A series of properties are exposed that determine whether the UI is ready to perform certain actions. They are CanAdd, CanEdit, CanDelete, and CanViewOrders. Notice that a reference to a data access provider (IProvider) is passed into the constructor. The provider interface is used to load customer data from the backend service. IProvider is also passed to any newly constructed customer model objects.

The WPF Model project has several model objects (also called Business Model Objects) that contain the data to be displayed on the View via WPF databinding. They are: CustomerModel, OrderModel, and OrderDetailModel. Of these, the CustomerModel is the most interesting. It contains a reference to IProvider which in turn calls into the Service Layer to add, update and delete customer data to the database. It also lazy loads (another pattern) Order data if necessary. Note: the WPF Model project is the only project within the WPF application with a Service Reference to the Service Layer. The Service Layer returns DTOs (Data Transfer Objects, another pattern). A Mapper class maps model objects to DTOs and vice versa. This class is located under a folder named \DataTransferObjectMapper.

All Model objects derive from abstract class BaseModel. BaseModel is important for two reasons: 1) it implements the INotifyPropertyChanged interface which prepares Model objects for databinding (to the View), and 2) it provides functionality that ensures that methods and properties are called on the UI thread (this is a WPF requirement). It keeps a reference to the Dispatcher object from when it was created in the constructor, and then checks that all subsequent calls are on the same thread as on which the object was created.

Next, we'll explore how the different MVVM components interact and work together. Let's start at the bottom. IProvider in project WPF Model is a simple interface that defines basic operations that the WPF application needs, such as, Login, Logout, GetCustomers, GetCustomers, and AddCustomer. The interface is implemented by the Provider class which communicates with the WCF services via request and response messages. The Provider uses the Mapper to map data transfer objects to model objects and vice versa.

The CustomerViewModel and the CustomerModel both have references to a Provider instance. The CustomerViewModel uses it to load all customers in the LoadCustomers method. Customers are loaded into an ObservableCollection of Customer model objects. This collection is public, which is important because it must be accessible for databinding. It is through databinding that the data gets transferred to the ListBox on the main WPF window. The following XAML snippet shows where databinding takes place:

```
<ListBox Name="CustomerListBox"
        ItemsSource="{Binding Customers}"
        SelectedIndex="{Binding Index, Mode=OneWayToSource}" >
```

It shows that the ListBox is databound to the Customer collection. But how does it know where the Customers collection is (remember that Customers is a public property on the CustomerViewModel)? The answer is that this is done by assigning the CustomerViewModel to the DataContext of the main window. Look at the constructor of the WindowMain and you'll find the relevant lines of code:

In C#

```
/// <summary>
/// Constructor
/// </summary>
public WindowMain()
{
   InitializeComponent();

   // Create viewmodel and set data context.
   ViewModel = new CustomerViewModel(new Provider());
   DataContext = ViewModel;
}
```

And in VB:

```
''' <summary>
''' Constructor
''' </summary>
Public Sub New()
    InitializeComponent()

    ' Create viewmodel and set data context.
    ViewModel = New CustomerViewModel(New Provider())
    DataContext = ViewModel
End Sub
```

Here, a CustomerViewModel is created and given a new Provider into the constructor. After that, the new CustomerViewModel is assigned to the DataContext property of the Window. Now the listbox knows how to find the Customers property using the parent's DataContext.

Additionally, in the XAML snippet above, notice that the SelectedIndex is databound to the Index property in the CustomerViewModel with a binding mode of 'OneWayToSource'. This is how the ViewModel is kept up-to-date with the currently selected customer on the UI.

What we have seen so far, is how Model data coming from the database is ultimately rendered onto the View. Next, we'll look at 1) how menu items are enabled / disabled by the ViewModel and 2) how changes made to a customer are persisted to the database.

Perhaps you have noticed that the ViewModel is kind of 'close' to the View. This is certainly true. In fact, the ViewModel is the DataContext of the View (i.e. the window). This closeness facilitates databinding. It would be nice if the menus on the main window would be databound to the ViewModel as well. The ViewModel does have the necessary information of when to enable/disable the different menu items. However, when responding to menu clicks the ViewModel would be responsible for launching

Login and / or Customer Edit dialog windows which would be incorrect. Remember, the ViewModel knows about the UI but it should not get into the business of opening UI specific windows or related activities. This would invalidate and negate the improved testability of these MV patterns.

Instead, in *Patterns in Action 4.0* we implemented an extra step by using a class named ActionCommands which holds RoutedUICommand for every menu item. In the XAML file these commands are bound to the window's CommandBindings. Executed and CanExecute map to command handlers that are located in the WindowMain code behind.

```
<Window.CommandBindings>
    <CommandBinding
      Command="commands:ActionCommands.LoginCommand"
      Executed="LoginCommand_Executed"
      CanExecute="LoginCommand_CanExecute" />
    <CommandBinding
      Command="commands:ActionCommands.LogoutCommand"
      Executed="LogoutCommand_Executed"
      CanExecute="LogoutCommand_CanExecute" />
    <CommandBinding
      Command="commands:ActionCommands.ExitCommand"
      Executed="ExitCommand_Executed" />
    <CommandBinding
      Command="commands:ActionCommands.AddCommand"
      Executed="AddCommand_Executed"
      CanExecute="AddCommand_CanExecute"/>
    <CommandBinding
      Command="commands:ActionCommands.EditCommand"
      Executed="EditCommand_Executed"
      CanExecute="EditCommand_CanExecute" />
    <CommandBinding
      Command="commands:ActionCommands.DeleteCommand"
      Executed="DeleteCommand_Executed"
      CanExecute="DeleteCommand_CanExecute" />
    <CommandBinding
      Command="commands:ActionCommands.ViewOrdersCommand"
      Executed="ViewOrdersCommand_Executed"
      CanExecute="ViewOrdersCommand_CanExecute" />

    <CommandBinding
      Command="commands:ActionCommands.HowDoICommand"
      Executed="HowDoICommand_Executed" />
    <CommandBinding
      Command="commands:ActionCommands.IndexCommand"
      Executed="IndexCommand_Executed" />
    <CommandBinding
      Command="commands:ActionCommands.AboutCommand"
      Executed="AboutCommand_Executed" />
```

```
   </Window.CommandBindings>
```

We recognize that some of the CanExecute and Executed parameters could have been databound directly to the ViewModel, but for consistency reasons we opted not to do so. Instead, these handlers query the CustomerViewModel to determine the proper action or response. An example of this is the ViewOrdersCommand_CanExecute which has just a single line of code.

```
e.CanExecute = ViewModel.CanViewOrders;
```

Next, let's examine how customers are added and changed. Customer information is edited in a separate window called WindowCustomer. To explore how MVVM works in this window it is important that you understand what goes on in its code-behind. In Window_Loaded the window's DataContext is assigned either a new CustomerModel or the currently selected CustomerModel (the assignment depends on whether this is a new or existing customer). The CustomerViewModel is made available via the Application object. Notice that the Save button is databound to the current CommandModel (which really is a RoutedUICommand). Both add and edit operations take place in this windows, but not the delete operation.

Two behaviors need further explanation. They occur while adding or editing a customer. When adding a new Customer, you'll notice that the Save button is initially inactive. The CanExecute in ViewModel's AddCommand validates the values as they are being entered. Only after all values are entered and the user has tabbed out of the last field will the button be activated (by the way, this behavior can be changed with UpdateSourceTrigger).

When editing an existing customer the Save button is enabled immediately. This makes sense, because all values are available and ready to be saved to the database. Now, start editing. Change the name and tab to the next field. Notice that the underlying customer box on the main windows is also changing. That is, as soon as a change is made to the model, the associated views are updated as well. In fact we have two views bound to one model. This can potentially cause a problem for when a user decides to cancel the edit. We solve this by keeping a copy of the original values in the CustomerWindow. So, when the edit is cancelled, the original values can be recovered (a candidate for the Memento pattern). Of course, alternative approaches are possible, but it is something to keep in mind when implementing MVVM.

WPF Commands (an original GoF pattern) play an important role in MVVM. Commands encapsulate a request as an object. For example, if your application supports Cut/Copy/Paste, then your UI probably has at least 3 different ways to support this: 1) menu items under the Edit menu, 2) hitting Ctrl-C, Ctrl-X, and Ctrl-V anywhere in the application, and 3) context menus that are invoked by right clicking the mouse. All these user events call the same Command object, so that the functionality is located at a single place.

What are the advantages of using of MVVM? Most importantly, it makes a very clean separation of the visual style from the behavior. This separation makes the application highly testable while the visual style can be changed without affecting the functionality. WPF has the concept of 'lookless' controls and the separation of visual style and its behavior is already fundamental to WPF. The MVVM promotes this separation by placing all behavior in a separate component instead of in code-behind. The visuals (sometimes referred to as the 'glass') and the behavior are loosely coupled by the use of databinding and commanding mechanisms.

Frequently, the ViewModel works as a filter between the data coming from the Model and the actual data that is displayed on the UI. This filter can involve data manipulation or transformation, or a true filter in the sense that only a subset of all records need to be displayed. In *Patterns in Action 4.0* this aspect of the ViewModel is not demonstrated.

As an extension to the Model you can include a timer which checks for database or service updates at regular intervals. Let's say you have a stock quote system that needs frequent updates of the latest ticker values. Using a timer you can query the quote service provider and update the view with new ticker values. It is important that this happens asynchronously on a different thread from the UI to avoid blocking. Since model objects are databound to the View, the updated values will be displayed immediately after they are retrieved from the quoting service.

Below is a summary of Model-View Design Patterns in *Patterns in Action 4.0.*

Model-View Design Patterns	**Project**	**Classes / Projects**
Model View Controller (MVC)	ASP.NET MVC Application	All Model, View, and Controller classes under the different Areas
Model View Presenter (MVP)	Windows Forms Application	Numerous classes spread over 4 projects.
Model View ViewModel (MVVM)	WPF Application	Numerous classes spread over 3 projects.

Summary

Patterns in Action 4.0 is a reference application that demonstrates when, where, and how design patterns are used in a modern, 3-tier, enterprise level, e-commerce environment. We are hopeful that after studying this reference application you are convinced that design patterns form an integral part in modern-day application architecture. Design patterns help you architect and design simple, elegant, extensible, and easily maintainable applications that users are demanding

If you have questions on using design patterns, the architecture of *Patterns in Action 4.0*, or have suggestion for future enhancements and improvements please do not hesitate to contact us via email at info@dofactory.com, or from our 'contact us' page on our website at www.dofactory.com/contact/contact.aspx. We look forward to hearing from you.

Good luck with your future design pattern and architecture endeavors.