

Patterns in Action 4.5

A pattern reference application



Companion document to Design Pattern Framework™ 4.5

by

Data & Object Factory, LLC
www.dofactory.com

Index

Index	2
Introduction	4
Goals and Objectives.....	4
What is Patterns in Action 4.5	6
What is new in Release 4.5	8
About this document	10
Setup and Configuration.....	11
Solution setup.....	11
Database Setup.....	14
Using SQL Server	14
Web.config setup	15
App.config setup	17
Application Functionality.....	17
Web app (MVC and Web Forms).....	18
Windows Forms Application.....	20
WPF Application	23
Application Architecture.....	28
Layered Architecture.....	28
Analyzing the layers.....	30
Solution and Projects	32
The 8 Projects	33
MVC Application	34
MVC Test Application	42
Web Forms Application.....	48
Windows Forms Application.....	53
WPF Application	55
Action Service.....	57
BusinessObjects	57
Object Persistence	59
DataObjects.....	60

ADO.NET as the data access platform	61
Linq-to-Sql as the data access platform	64
Entity Framework as the data access platform	67
DAO Review	72
Error Logging.....	72
Design Patterns and Practices	77
Gang of Four Design Patterns	77
Enterprise Design Patterns	81
SOA and Messaging Design Patterns.....	83
Model-View Design Patterns.....	83
MVC (Model-View-Controller) Design Pattern	84
MVP (Model-View-Presenter) Design Pattern	88
MVVM (Model-View-ViewModel) Design Pattern.....	90
MV Patterns Summary	97
Summary.....	98

Introduction

Patterns in Action 4.5 is a reference application designed to demonstrate when, where, and how design patterns and practices are used in building multi-layer .NET apps.

Goals and Objectives

The following list summarizes the goals and objectives for the *Patterns in Action 4.5* reference application:

Educational – the purpose of *Patterns in Action 4.5* is to educate you on the importance of design patterns, and when, where, and how to use them in a multi-layered application. The ASP.NET MVC platform 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 4.5*, combined with the new .NET 4.5 features offer a great opportunity for enhanced productivity. Design patterns help you build apps effectively and efficiently.

Extensibility – extensibility is implicit in applications that 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. If you have any experience building and deploying applications, you know that once your application has been released, 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, robust, 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 '*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). 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 three previous points: extensibility, simplicity and elegance. Code that is extensible, simple, and elegant is much easier to support and maintain.

Modular –applications that are designed around autonomous functional modules are easier to understand and maintain. With modules 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 modules, 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 employee data. Without knowing the exact requirements, you already know, ahead of time, that there will be an employee module. 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 (repository or service).

The best applications (granted, ‘best’ is subjective) are built by architects who think in modules and then apply the design patterns to make these ‘slices of functionality’ a reality.

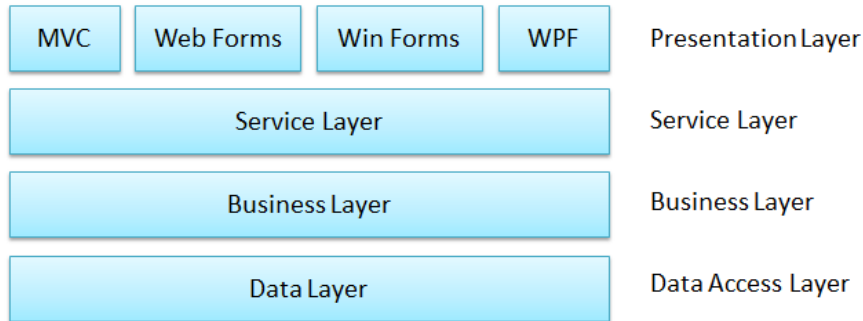
Not coincidentally, ASP.NET MVC supports a feature called *Areas*. They are designed to help developers better organize their code functional areas. These Areas are very much in line with the concept of building modular code and we strongly suggest you use Areas in your own MVC projects.

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 4.5

Patterns in Action 4.5 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 (not fully implemented in this version. See *Spark 4.5* for a complete shopping cart). A separate administrative area allows administrators to view and maintain (add, edit, delete) user records as well as analyze customer orders and order details.

From an architectural perspective, *Patterns in Action 4.5* is many applications in one. A core element in this solution is that it has a central, well-defined service layer (Façade pattern) that exposes common e-commerce application tasks as a service. This service is consumed by 4 separate client applications built on 4 different UI technologies: ASP.NET MVC, ASP.NET Web Forms, Windows Forms, and WPF. Each client technology consumes the *exact same public service interface*. Here is an overview:

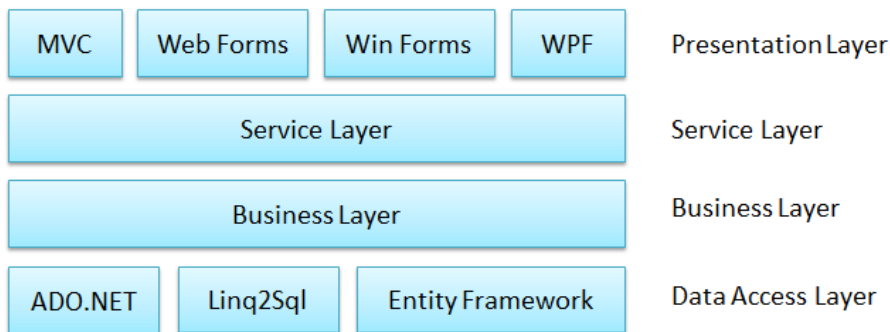


Along the top are the 4 clients: 2 web clients, MVC and Web Forms, and 2 windows clients, Win Forms and WPF. This is called the Presentation Layer.

The Service layer provides a central access point to all e-commerce services offered by the application. The 4 UI Technologies all access the same services.

The Business Layer contains the business objects with their own properties and methods. Each business object has the ability to handle validation business rules.

The Data Layer in *Patterns in Action 4.5* supports 3 different data access technologies: ADO.NET, Linq2Sql, and Entity Framework. Below is a figure which includes these technologies.



As you can see, the system can vary at the top (the UI technology) and vary at the bottom (the data access provider). You can select any combination and run it. For example you could run MVC with Entity Framework, WPF with ADO.NET, or Web Forms with Linq2Sql. They all work fine.

What this demonstrates is that with design patterns you can build great flexibility into your architecture. The *Patterns in Action 4.5* application demonstrates architectural flexibility, but this is most likely overkill in your own work. Usually teams settle on a single data access technology (such as, ADO.NET, NHibernate, or Entity Framework) and leave it at that. This is the right thing to do because there is almost never a need to swap out data access technology and/or database (contrary to some who argue for this flexibility, but in reality it almost never happens).

Swapping out the UI technology is a different story. Suppose you are building an intranet web app using MVC for your company. You complete the app, deploy it, and it becomes an instant hit. Users just love it. What will happen next is that they will start asking for a tablet or phone version to gain access while on the road. Back in the office, other users are asking for a high-end dashboard system for which WPF seems the appropriate tool. So, here is a situation where you need to support multiple UI technologies, ideally all using the same backend system, that is, the Service, Business, and Data Layers combined.

Supporting multiple UI technologies is entirely possible with the above layered architecture. You can experience it first hand in *Patterns in Action 4.5* with 4 different UI clients. For each client, the supporting libraries (i.e. layers) are referenced and directly linked in, but when supporting remote devices (tablets and phones) you will need to expose a web service. Most likely, you will want to consider building a REST interface using Microsoft's new Web API which comes with Visual Studio 2012. Using WCF is another option.

What is new in Release 4.5

Release 4.5 contains numerous changes, including many simplifications. This release embraces the *Convention over Configuration* principle or CoC for short. CoC is a best practice technique that relies on patterns, standards, and conventions to make programming easier. For example, a convention used in this app is that business objects have the same name as their associated database tables. Similarly, property

names are the same as their corresponding columns names in the tables. This one-to-one mapping is the Active Record design pattern which is one of Fowler's Enterprise Patterns.

CoC simplifies the development process by reducing the number of decisions a developer needs to make. Conventions also reduce the need for configuration as there is little left to configure. For example, several ORMs require that each table column is associated and mapped to an object property by declaring the relationship in an XML file. However, if you establish a standard that all class and property names are the same as their associated table and column names (as we did above), then there is no need for this. Everyone on the team knows and works according to the one-to-one mapping rule. Only when there is an exception, will you need to consider how to handle this.

In our quest for simplicity and conventions, we decided to remove WCF in this release. WCF is a powerful and useful tool but anyone who has spent some time with it will readily admit that is complex and can be frustrating to setup and configure.

Of course, WCF, SOAP, and other protocols have their place and are still widely used, but REST is where the action is today. A reference REST implementation is available in *Spark 4.5* which is available as a separate solution in this release.

Microsoft has also recognized this trend toward keeping things simple. They added Web API as a simpler alternative to WCF when building REST interfaces. Furthermore, they replaced the Membership systems with a new system called SimpleMembership. As its name implies it is simpler than the older system. SimpleMembership is light-weight and easy-to-configure. The ADONETDB.mdf database file, which was included in our prior release, is now a thing of the past. In this release we use SimpleMembership rather than the old Membership system.

With the addition of Nuget to Visual Studio 2012 Microsoft has made it easy to include and consume packages in .NET projects. Packages are freely available tools and libraries, from Microsoft or other 3rd parties. Pretty much all popular open source projects that are relevant to .NET developers are available today. In *Patterns in Action 4.5* we include the following packages (many others are included by default):

- 1) Automapper
- 2) Moq

Talking about CoC and building simpler systems, included in this release of the Design Pattern Framework is a new rapid application development (RAD) framework called *Spark 4.5*. With Spark we have pushed the CoC, YAGNI (You Aint Gonna Need It), and 'keeping it simple' paradigm even further: Spark is a light-weight, pattern-based application framework that allows you to quickly build apps that are simple, robust and very fast. It is available as a separate Visual Studio 2012 solution (called Art Shop) with its own PDF documentation. Upon completing the *Patterns in Action 4.5* tour your next step will be the *Spark 4.5* framework.

In summary, CoC and building simpler systems is the central theme of this release.

About this document

It is best to read this document from the beginning to the end. Each section builds on the previous one, so follow along in a linear fashion. There are five 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 and app.config files.

Application Functionality: This section will step you through the functionality of the application in which users shop for products and administrators manage customer's records and their orders.

Application Architecture: This section provides an overview of the different layers used to construct the application: Presentation, Service, Business, and Data. You will learn how the different layers communicate (i.e. who references who) as well as the best places to add any 'non-functional' items, such as, authentication, authorization, data validation, and transaction management.

Solution and Projects: This section looks at the .NET Solution and its 8 projects. The projects are structured according to the solution's layered architecture and it is important that you have a good grasp of the organization.

Design Patterns and Best Practices: This section lists and catalogues the numerous design patterns that are used in *Patterns in Action 4.5*. They are organized in 4 groups:

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

Setup and Configuration

Here we discuss setup and configuration of *Patterns in Action 4.5*. The solution consists of 8 projects of 6 different types: ASP.NET MVC, ASP.NET Web Forms, Windows Forms, WPF, class libraries, and a test project. Please note that the free Visual Studio 2012 Express edition does not support this variety of project types which is why you need Visual Studio 2012 Professional or better to run the application.

Solution setup

If you followed the recommended path for *Patterns in Action 4.5* solution then the Patterns in Action solution will be here (although any location will do).

C# Edition: *C:\Users\%username%\Documents\DoFactory\DPF\4.5\CS\Patterns in Action*

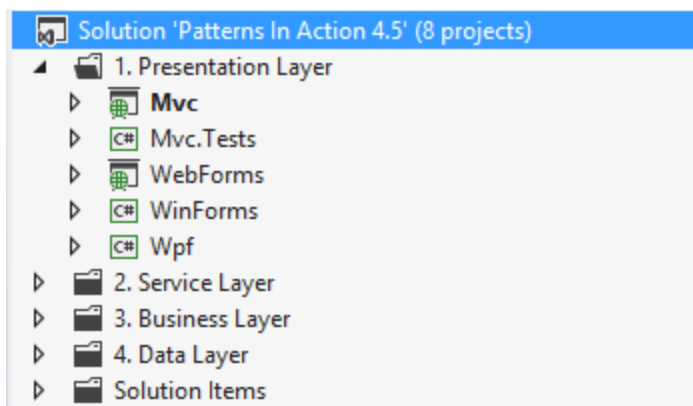
VB Edition: *C:\Users\%username%\Documents\DoFactory\DPF\4.5\VB\Patterns in Action*

Inside this path, your folder structure looks like this:

Name	Date modified	Type	Size
ActionService	5/17/2013 3:14 PM	File folder	
BusinessObjects	5/13/2013 8:06 PM	File folder	
DataObjects	5/19/2013 1:09 PM	File folder	
Mvc	5/18/2013 7:33 PM	File folder	
Mvc.Tests	5/12/2013 11:07 AM	File folder	
packages	5/12/2013 11:04 AM	File folder	
WebForms	5/19/2013 11:11 AM	File folder	
WinFormsApp	5/19/2013 2:03 PM	File folder	
WpfApp	5/19/2013 2:08 PM	File folder	
Action.sql	5/9/2013 1:52 PM	Microsoft SQL Ser...	488 KB
Patterns In Action 4.5.sln	5/17/2013 2:13 PM	Microsoft Visual S...	7 KB
Patterns In Action 4.5.v11.suo	5/20/2013 12:16 PM	Visual Studio Solu...	665 KB

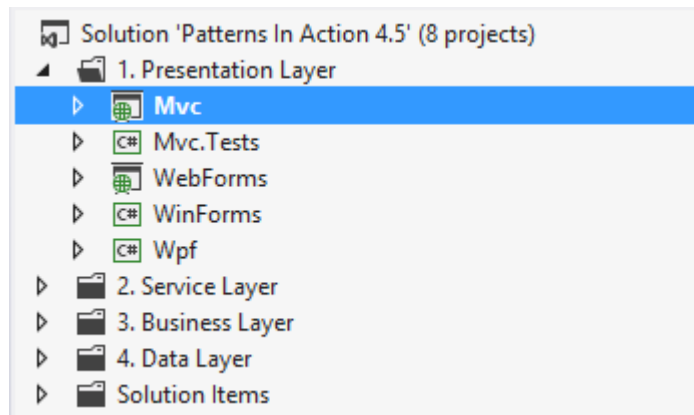
Double click or right click on the solution file named “**Patterns in Action 4.5.sln**”. Visual Studio 2012 launches and opens the solution. You can also load the solution from within Visual Studio using the File->Open->Project menus.

Once loaded in Visual Studio 2012, you can immediately run the application. The default UI client is MVC. To change this you can select any of the projects under the Presentation Layer as the Startup Project. There are 4 types: MVC, Web Forms, Windows Forms, and WPF:



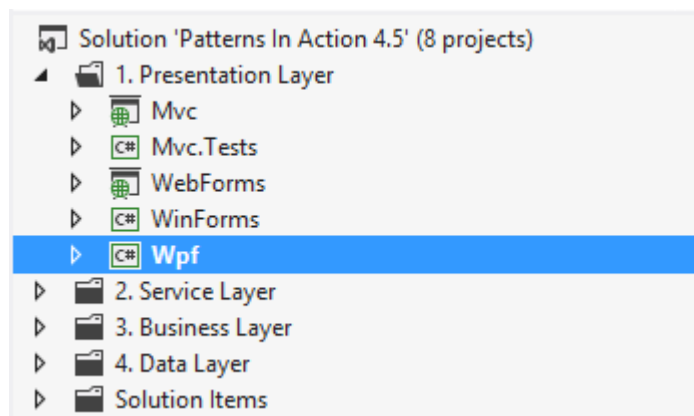
Here are two examples of starting a project:

- 1) To run the MVC Application set the **MVC** project as the Startup Project (it will display in bold). See below.



By the way, the **MVC.Tests** project contains tests for the MVC application. Testing and running these is discussed later in this document.

- 2) To run the WPF Application set the **WPF** project as the Startup Project (it will display in bold). See below.



Database Setup

With the release of Visual Studio 2012 Microsoft has discontinued SQL Express and replaced it with LocalDb. In addition, they added SimpleMembership as an alternative for the traditional Membership services that were offered. In keeping with these changes we are now using both LocalDb and SimpleMembership. The membership change removed the need for a separate ASPNETDB.MDF database; much simpler.

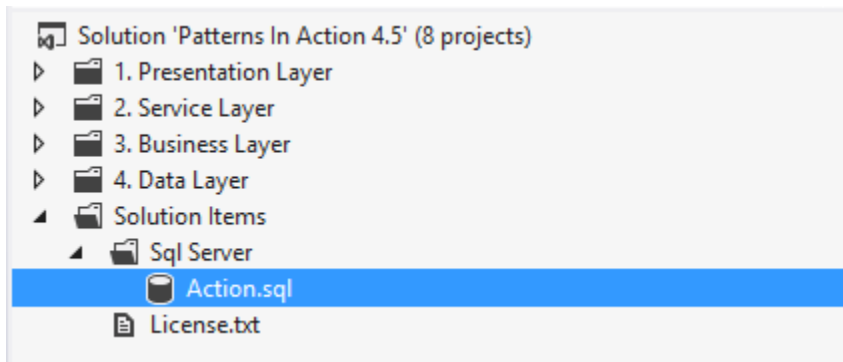
In this release of *Patterns in Action* we focus on LocalDb and Sql Server. MS Access and Oracle support are not included. Focusing on a single technology keeps the noise level to a minimum and the app simple and tidy. Now, it would be easy to add these back in if you wanted to. If interested, please check the previous *Patterns in Action* version 4.0 for code and details.

Out of the box, *Patterns in Action* 4.5 supports LocalDb. Each UI client project has its own copy. Their names are Action1.mdf, Action2.mdf, etc. (or ActionVb1.mdf, ActionVb2, md, etc. for VB users). You can find them in the respective *App-Data* folders in each UI project. Each database contains 5 application tables, as well as 4 extra tables that are required for SimpleMembership. Please note that SimpleMembership automatically adds these 4 tables if they don't already exist.

You can also run the 4 UI clients against a single SQL Server database, rather than against different copies of LocalDb. This is discussed next.

Using SQL Server

First, create an empty database named **Action** (or any other name). Second, run the script named *Action.sql* against this new database. This script will create the data model (i.e. 5 tables) as well as enter their data. The file *Action.sql* is located in the .NET solution under a folder named *Solution Items\Sql Server*. See image below.



As a final step you will need to adjust the connection string in web.config file or app.config and you're ready to go. Setting the connection string is discussed next.

Web.config setup

Web.config is the configuration file for web sites and web services. It is used to configure database connections and other custom application options. In *Patterns in Action 4.5* the configuration options are kept to a minimum. For the purposes of this app the only items you need to know about are 1) the connection strings and 2) the provider for the data access layer.

They are listed below:

```
<connectionStrings>
  <add name="Action"
        connectionString="Data Source=..."
        providerName="System.Data.SqlClient" />
  <add name="ActionEntities"
        connectionString="..."
        providerName="System.Data.EntityClient" />
</connectionStrings>

<appSettings>
  <!-- options are: Ado.Net, Linq2Sql, or EntityFramework -->
  <add key="DataProvider" value="Linq2Sql" />
</appSettings>
```

The database connection string is named *Action*. The Entity Framework connectionstring is named *ActionEntities*. Note that the connection strings are slightly different for each UI client because the physical database names are different: Action1.mdf, Action2.mdf and so on (in VB projects they are named ActionVb1.mdf, ActionVb2.mdf, etc.). If you use

SQL Server you need to adjust both Action and ActionEntities which are then the same for each UI project.

In the appSettings section you select the DataProvider used in the Data Access Layer. Valid options are *ADO.NET*, *Linq2Sql*, or *EntityFramework*.

The web.config file for the **Web Forms** application requires some additional details. First it contains a configSection in which the ViewState Service is configured (a description on this is available later in this document). ViewState is a feature that only exists in Web Forms. Second, authorization is configured in web.config with a section named location. It specifies the users that have access to the application's /admin folder, like so:

```
<location path="admin">
  <system.web>
    <authorization>
      <!-- allow admins -->
      <allow roles="Admin" />
      <!-- deny all other users -->
      <deny users="*" />
    </authorization>
  </system.web>
</location>
```

This will only allow authenticated users with the role of Admin to the /admin url and denies access to anyone else.

As mentioned before, *Patterns in Action 4.5* uses the new SimpleMembership system. The default Membership provider is SimpleMembership and in general no web.config settings are required. The exception is again the **Web Forms** project which requires that the following sections be included in web.config. Note: don't add these, as they are already included.

```
<membership defaultProvider="SimpleMembershipProvider">
  <providers>
    <add name="SimpleMembershipProvider"
        type="WebMatrix.WebData.SimpleMembershipProvider,
            WebMatrix.WebData" />
  </providers>
</membership>
```



```

    </providers>
</membership>

<roleManager enabled="true" defaultProvider="SimpleRoleProvider">
  <providers>
    <add name="SimpleRoleProvider"
        type="WebMatrix.WebData.SimpleRoleProvider,
            WebMatrix.WebData" />
  </providers>
</roleManager>

```

App.config setup

Instead of *web.config*, windows applications (Windows Forms and WPF) have an *app.config* configuration file. They are smaller than *web.config* files and only contain specifications for the connection strings and the data access provider. Other than the physical database name (Action3.mdf and Action4.mdf or ActionVb3.mdf and ActionVb4.mdf) they are the same for both Windows projects.

Application Functionality

This section presents an overview of the functionality offered by *Patterns in Action 4.5*. We'll review what the application does and how users and the administrator navigate through the system. The application itself is kept simple so as to maximize learning and minimize unnecessary distractions.

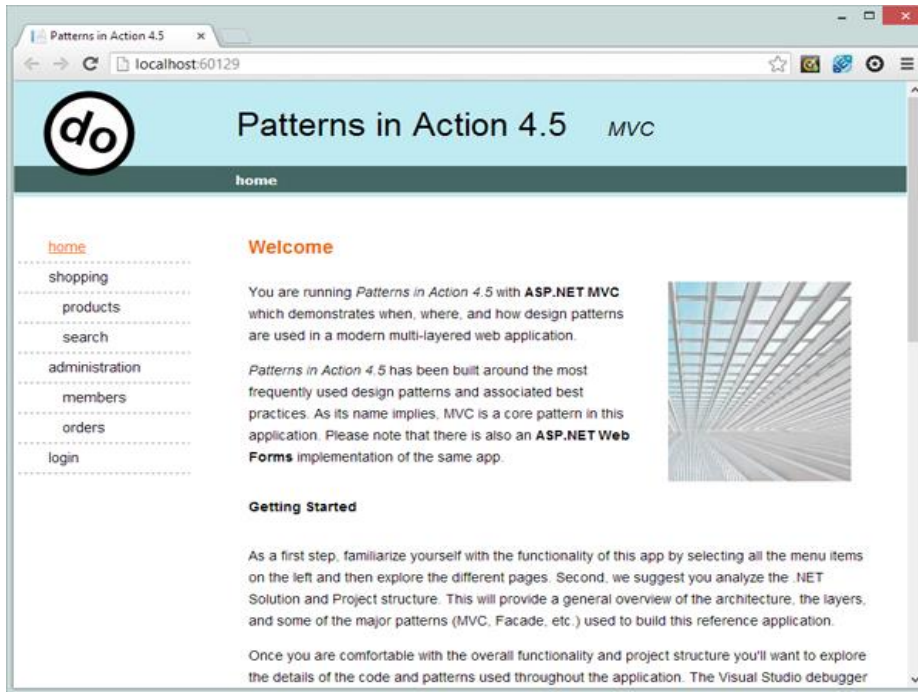
You can *experience Patterns in Action 4.5* in one of 4 ways:

- 1) as an MVC web app
- 2) as an Web Forms web app
- 3) as a Windows app
- 4) as a WPF app

Functionality and look-and-feel of the first two, MVC and Web Forms, are essentially the same. We collapsed these two in our next discussion. So we will discuss 3 *experiences*, that is, Web App, Windows App, and WPF.

Web app (MVC and Web Forms)

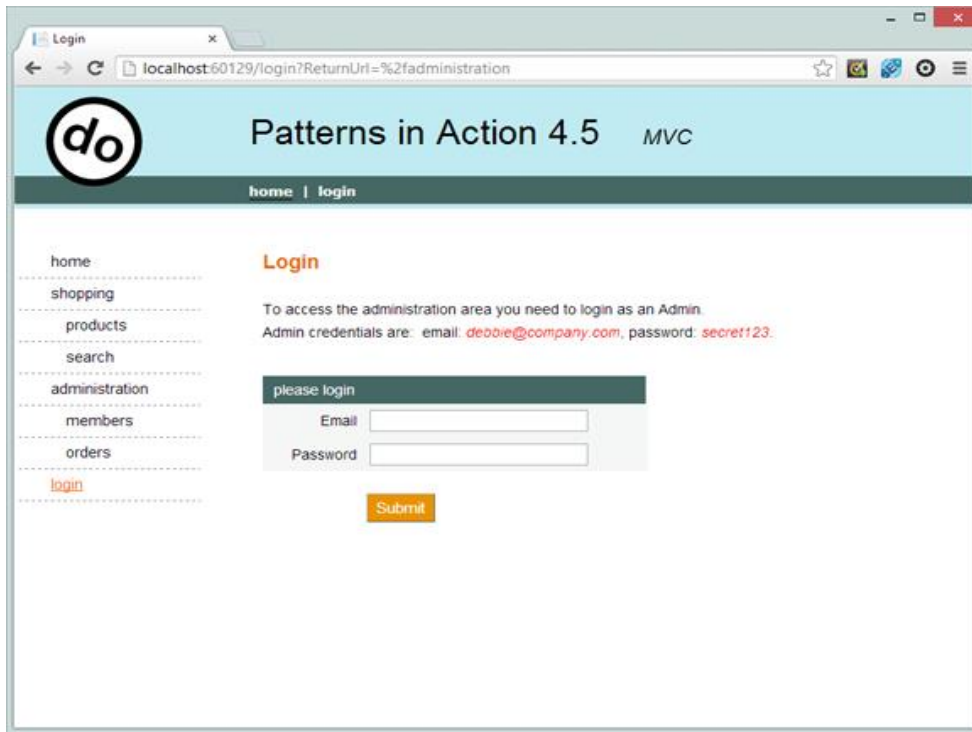
To run the web app select one of the web projects as your Startup Project and select Run (hit F5). You will see the following opening screen. Please read the welcome message: it contains information on how to make the most of the application.



You are currently in the role of an **unauthenticated** user who is shopping for electronic products. Start with 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 there is a button to add a product to your shopping cart. Note that there is no functioning shopping cart in this app. For a fully functioning shopping cart we refer to the new *Spark 4.5* application, which is part of the *Design Pattern Framework 4.5* package.

Next, login as an administrator whose task it is to manage the customers (called members) and review their orders and order details. This functionality is available from

the menu items listed under administration. Before entering the administrative area you must be authenticated and any of the admin menus will redirect to the login page. You can also select login yourself with the login menu item. Enter the suggested credentials. They are: email *debbie@company.com* and password *secret123*.



Once logged in, open the members page which allows three basic customer maintenance operations: Add, Edit, and Delete. Experiment with these options. A business rule in the application states that members with orders cannot be deleted. All existing customers have orders, so to be able to delete we suggest that you first Add a new member to the list. After that, find the new member in the list. Select Edit and change some fields of the new member and save your changes. Finally, from the member list, Delete that member from the database.

Orders can be viewed on the Orders page. It contains a member 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). View all orders for a customer and order details (line items) for one of the orders.

All order-related pages follow the master-detail paradigm, that is, the list of customers is the master and the customer 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.

Selecting logout will log you out from the application.

A few more items that are worth pointing out. The app displays 'bread crumbs' just below the header. Essentially, it displays the current position in the site map. Web Forms uses the built-in SiteMapPath control (using the Web.sitemap file), but in MVC this is a custom 'Crumb' control.

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

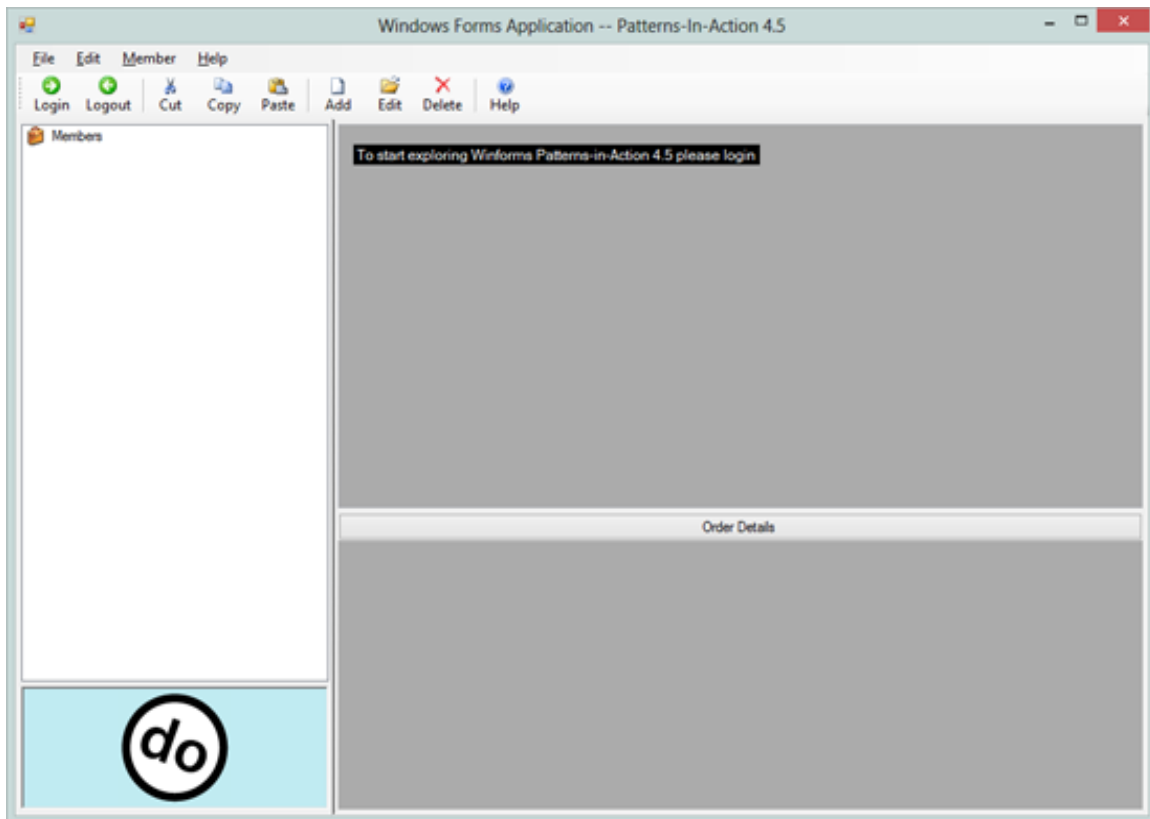
These pages are relatively light on JavaScript and jQuery. Today, more and more web applications are shifting functionality to the client to create highly interactive, beautiful app. All this requires a heavy dose of JavaScript and jQuery. For a great example of an app that relies much more on JavaScript and jQuery we refer to the *Spark 4.5* reference application.

With the increased use of JavaScript and jQuery so is the need and interest in Design Patterns and Best Practices for these languages. JavaScript, it turns out, could not have reached the level of success and popularity it has without the use of patterns and practices. The use of patterns is very widespread in JavaScript. For additional information on pattern for JavaScript and jQuery we refer to our *JavaScript and jQuery Patterns* package available from our website at www.dofactory.com.

Windows Forms Application

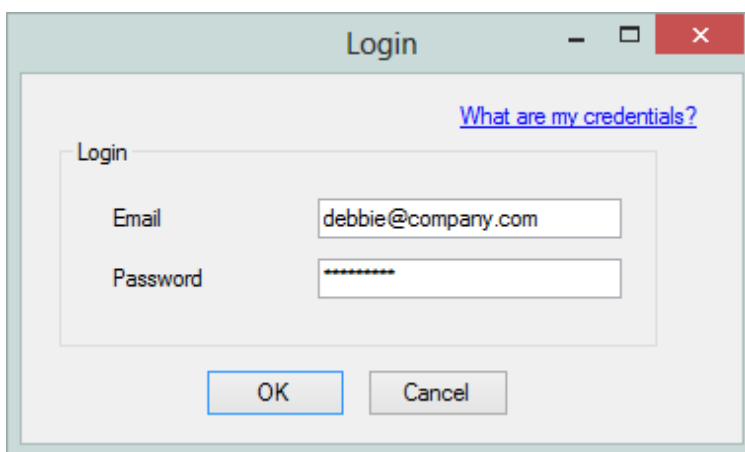
To start the Windows Forms Application set Windows Forms Application as the Startup Project and select Run (hit F5). As mentioned earlier, this app is also a client to the three shared layers: Service, Business, and Data.

Following launch, you will see the main form with three empty panes.



Windows application (not logged in)

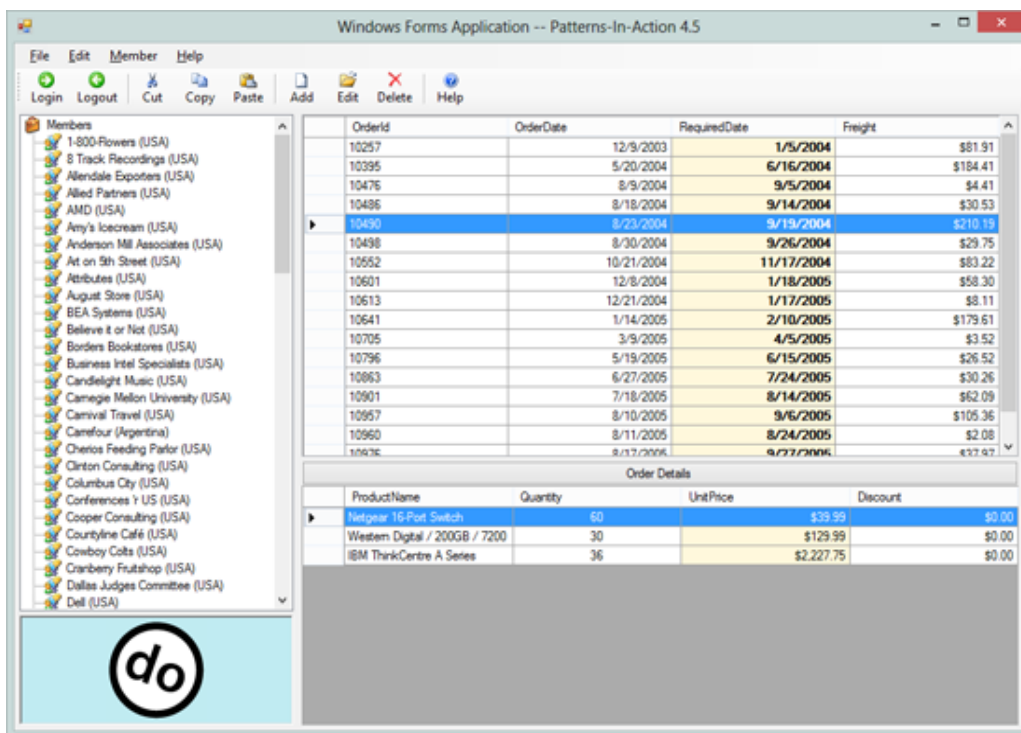
Once loaded, select the File->Login menu item or click the Login toolbar button. This opens the login dialog in which login credentials are entered.



Login form

The text boxes in the login dialog are pre-populated with valid values and selecting OK will automatically log you in. The credentials are: email *debbie@company.com* and password: *secret123*. If you forget your credentials click the [“What are my credentials”](#) link label.

Once logged in, a list of customers will display in the tree view on the left. Click on one and their orders and order detail data are retrieved and displayed. Highlight an order and order details will display instantly at the bottom pane.



Windows forms app with customers, order, and order details

Orders and order details are lazy loaded (LazyLoad pattern), meaning that they are only retrieve when necessary. Once these have been retrieved from the database, they are cached on the client. Over time the performance of the application improves as more and more orders are being cached.

The member (customer) maintenance functionality is the same as it is in the Web Application: you can Add, Edit, and Delete member records. To reiterate: not only are

they functionally the same, they also share the same Service, Business, and Data layers. The details of which are explained later in this document.

Experiment with the member maintenance options from the Member menu or by clicking the Add, Edit, Delete toolbar icons. The functionality is fairly straightforward and self-explanatory. Again, a business rule states that members with orders cannot be deleted, but since all customers have orders you need to create a new member to be able to explore this functionality. Note that new members are added to the bottom of the customer tree. Once added, edit it if you want and then finally delete it from the list of members.

As a last step select Logout and exit from the application.

As an aside, when running in Debug mode, both WinForms and WPF create a copy of the LocalDb database to their \bin folders. This is just something to be aware of: the application will function as expected, but when going to the \App_Data\Action3.mdf or \App_Data\Action4.mdf databases (or ActionVb versions) you may be puzzled as to why your changes do not show up.

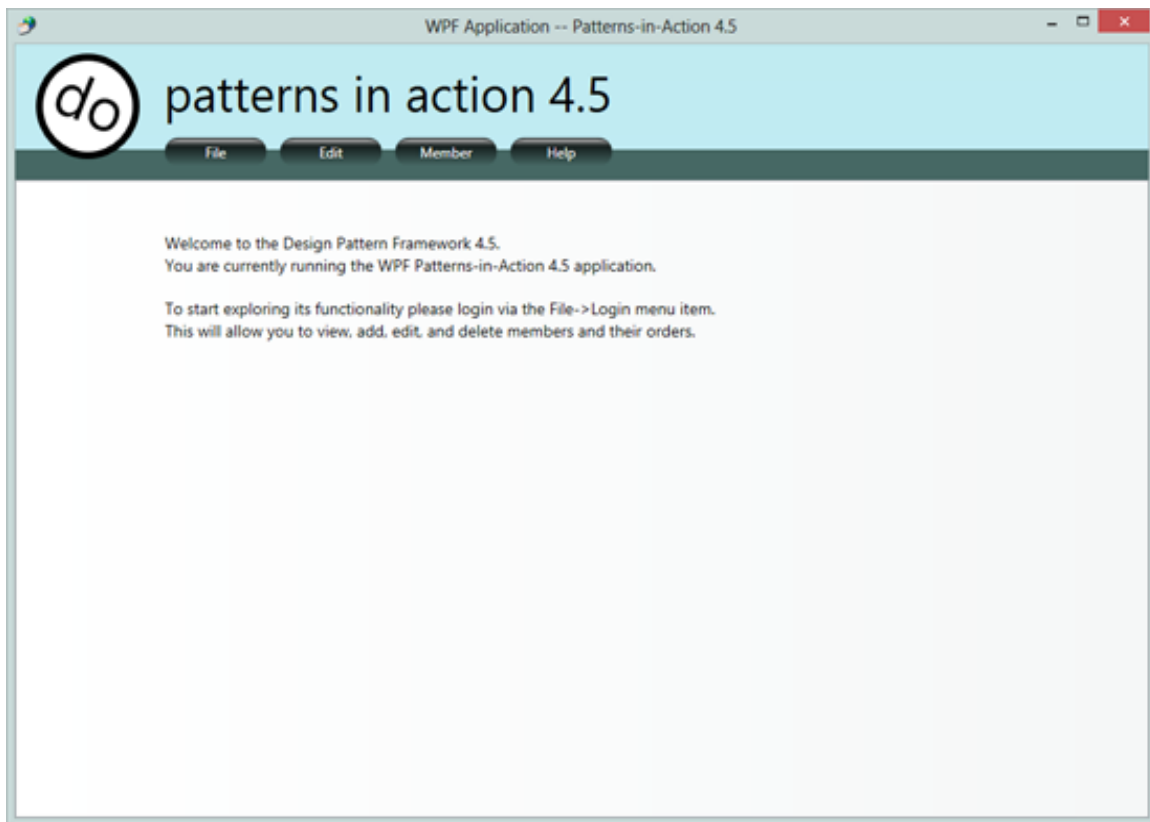
These apps will continue working on their \bin databases until a change is made to the original database in the \App_Data folder. At that time Visual Studio will create a new copy of that database and override the one in the \bin folder.

Of course, none of this will be an issue if you use a SQL Server database.

A final note about the Win Forms app: login is stubbed out because SimpleMembership is for web only.

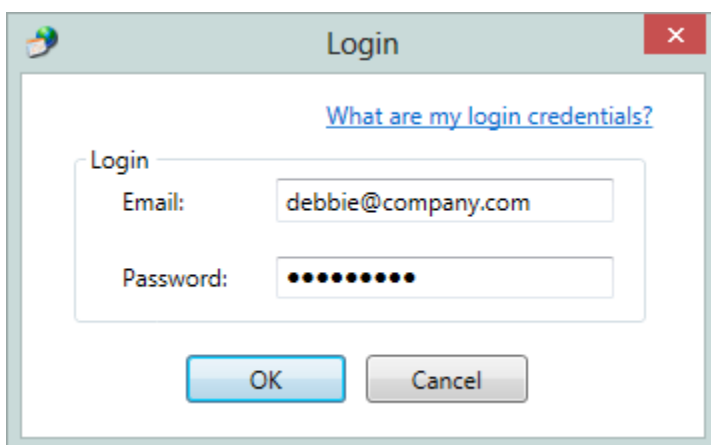
WPF Application

To start the WPF Application set the WPF Application Project as the Startup Project and select Run (hit F5). Like the other apps before, this application also is a client to the 3 shared layers. The implementation details of this app are discussed later in this document.



WPF Application

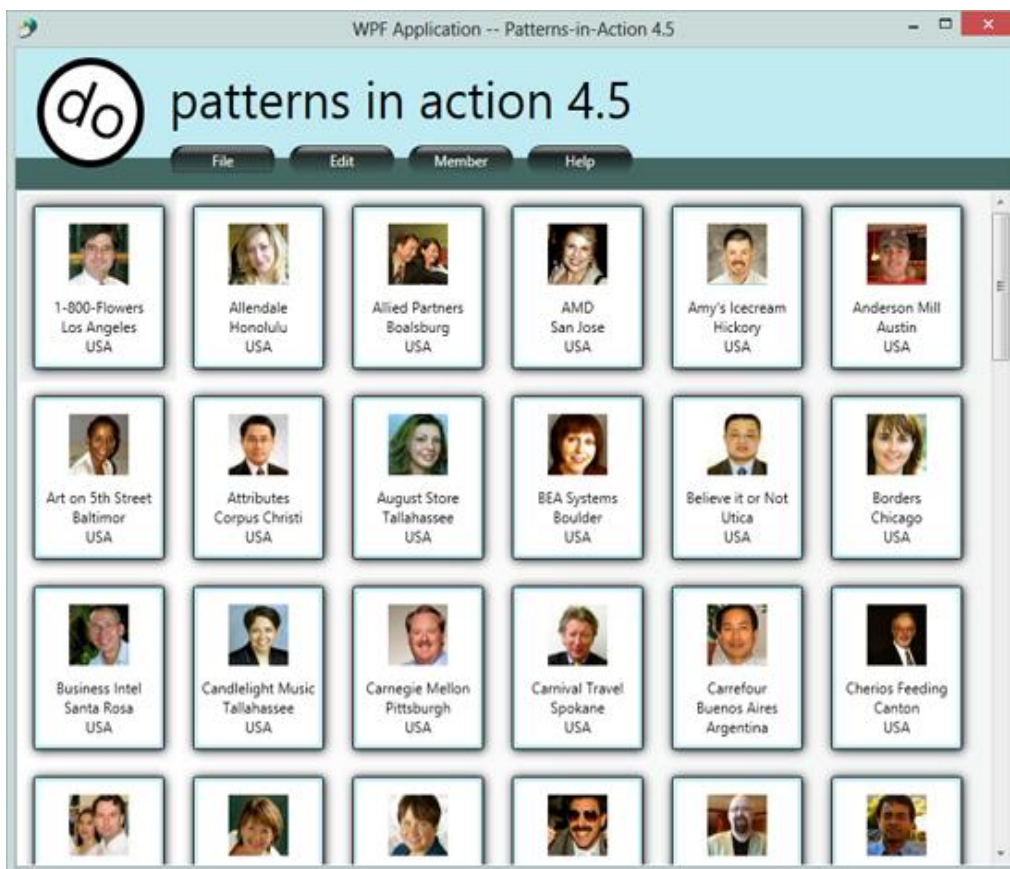
Once loaded you see the main form with four top level menus. Notice the menu button glow when the mouse moves over them. Select File->Login menu item or enter Ctrl-I. This opens the login dialog in which you enter login credentials.



WPF Login dialog

The text boxes in the 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: email *debbie@company.com* and password *secret123*.

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



WPF Application with customers retrieved.

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

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



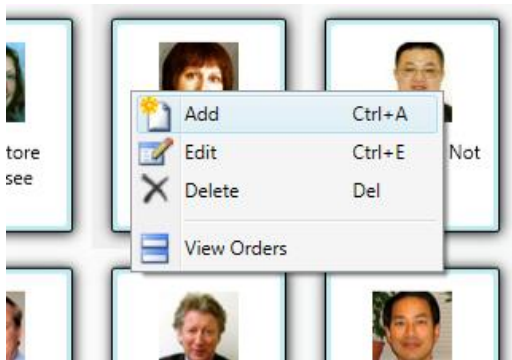
WPF Application, customer details, orders, and order details.

In this window select an order and the related order details will display instantly at the order detail list (both orders and associated order details have been retrieved for the selected member). Selecting the Close button will close this window.

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

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

below. By the way, the Command pattern plays an important role in supporting both top menus and context menus.



Right clicking on customer box opens Context menu

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

As a last step select Logout under the File menu button, which logs you out and clears the main page. Then select File->Exit.

A final note about the WPF app: login is stubbed out because SimpleMembership is for web only.

Application Architecture

In this section the application architecture of the *Patterns in Action 4.5* reference application is presented and the design decisions that went into building this solution.

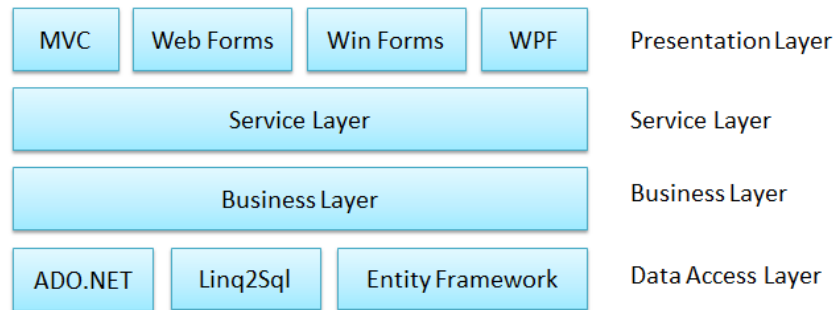
In general, you should have a good understanding of the functional (and non-functional) requirements before you can make decisions on the design and architecture of a new application. Functional requirements include the business processes that the application will perform. An example may be: the user 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.

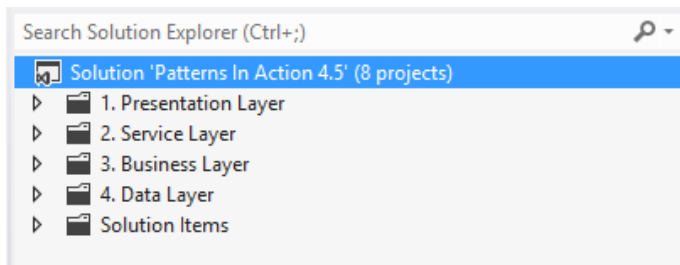
Layered Architecture

The *Patterns in Action 4.5* solution is built around a 4-layer architecture. The next figure has a cross-sectional view of the application layering.



First of all, you may be more familiar with a 3-layered system: Presentation, Business, and Data. We have depicted the Service Layer as a separate layer, but this is a matter of preference; some developers simply consider it part of the Business Layer.

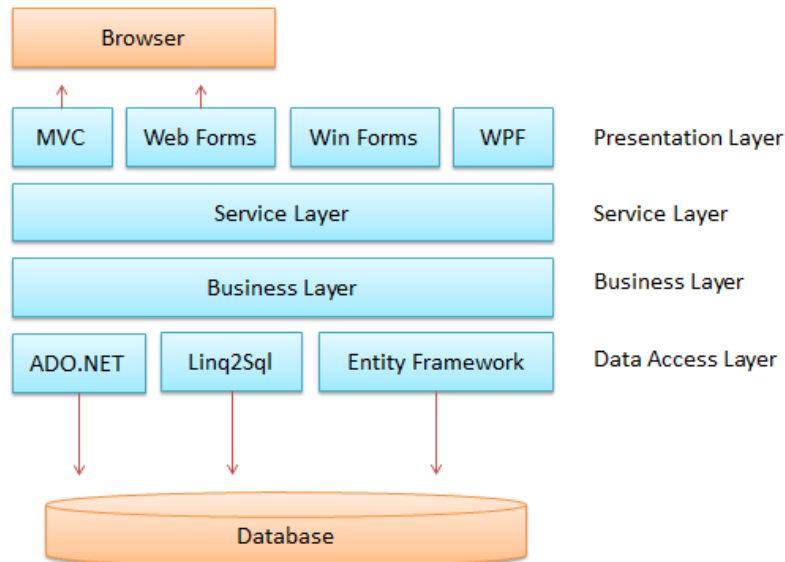
When viewing the Solution Explorer in Visual Studio you see the layers represented by numbered sub-folders (see image below). The four layers in the cross section above exactly matches the 4 solution folders in the application.



Solution Explorer with Layers

There are some architects who refer to the architecture as having 4, 5, 6, and sometimes even 7 layers. In reality they are talking about the same basic model, except that they choose to separate sub-systems that are required to run the application and define these as a separate layer.

Some front-end developers consider the client tier (browser) to be the 1st tier and some DBAs consider the database to be the last tier. Actually, 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). Here is their view of the world:



In this document we will stick with common terminology and continue referring to our architecture as 4-layer, but realize that more than 4 layers or subsystems are involved.

Analyzing the layers

Let's examine the cross-section starting at the bottom and then going up. At the very bottom we have the database. Our database is simple; it has just 5 application tables with the necessary indexes and 4 SimpleMembership related tables. That is all.

Next we find the data access layer who's main responsibility is to provide access to the database and map database tables to business objects and vice versa. The data layers 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 different data providers: one set for ADO.NET, another for Linq-to-Sql, and a set for the Entity Framework. The DAO pattern makes it easy to switch to a different data access technology just by changing a web.config configuration setting.

The Business layer 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 built-in business rule engine made available through a base class from which each business object derives.

Next is the Service layer. This layer implements a single point of entry (the Façade pattern) through which all communication with the layers below must occur. The Façade is the entry point into the backend and exposes a simple API. In *Patterns in Action 4.5* the Service API is the same as the DAO API (they expose the same methods), but there is no rule that says that they have to be the same.

In fact, in *Spark 4.5* you will learn about a different, and more flexible API that is modeled in terms of repositories (the Repository Pattern is one of the Enterprise patterns). *Spark 4.5* also has a service layer but this is just a thin veneer over the repositories. This service layer focuses on more complex business rules and it also manages the necessary database transactions. Don't worry if you don't quite understand this, as we are getting ahead of ourselves. Continue exploring this application and only when you are ready move on to *Spark 4.5*.

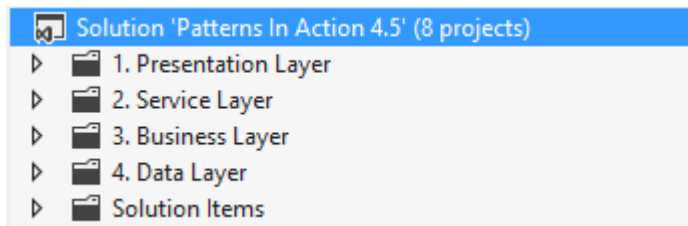
Note: The previous version of *Patterns in Action* had a service layer that was built around SOA (service oriented architecture) using WCF. It was more complex with concepts like Request and Response Messages, DataContracts, DataMembers, and ServiceContract. If you need to build a Service Oriented API (SOA), please check out the previous version (it is available for download). In this release WCF is not used.

The Presentation layer supports four different UI technologies: ASP.NET MVC, ASP.NET Web Forms, Windows Forms, and WPF. They all consume the exact same Service layer and other layers underneath it.

The browser, sometimes referred to as the Client Layer, is relevant only to Web Applications and represents the browser which renders HTML and can include JavaScript/jQuery, Flash, and/or Silverlight.

Solution and Projects

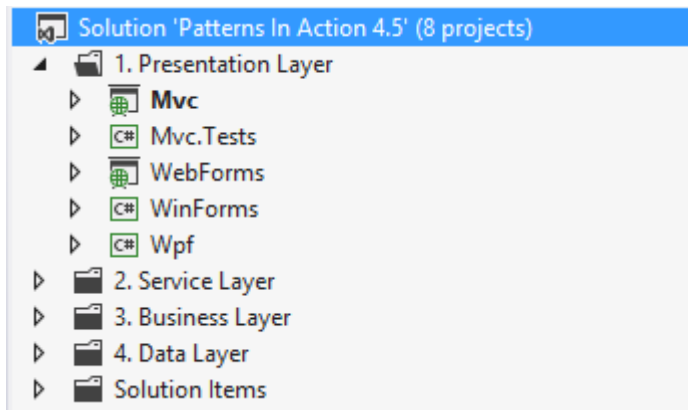
This section explores the *Patterns in Action 4.5* Solution and its Projects as well as their relationship with the aforementioned layers. Open the application in Visual Studio 2012. Collapse all projects and all folders. Don't open the folders just yet. Your Solution Explorer will look like this:



.NET solutions and projects

The Solution Explorer shows 4 numbered layers. They are numbered so they display in a logical top-to-bottom order.

Open the Presentation Layer folder.



You see 4 UI project types: an MVC application (with Tests), a Web Forms application, a Windows Forms application, and a Windows WPF application. Again, all 4 consume the same services in the lower levels: that is, layers 2, 3, and 4.

The Service Layer represents the single point of entry for all applications into the backend services, meaning the Business layer and the Data layer. The Data Layer is the only place with access to the database. The Presentation layer never accesses the

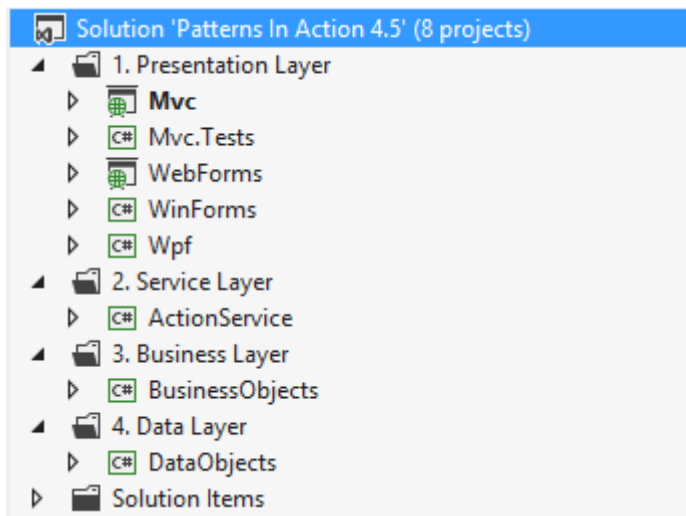
Data Layer, but it does keep a reference to the Business Layer to make it aware of the Business Objects.

The folder named \Solution Items contains a couple project artifacts, including a license statement and a file named action.sql which contains the scripts to create and populate the database used in this solution.

Version 4.5 of *Patterns in Action* is simpler than prior versions with fewer projects. Some projects have merged with others and WCF and the Hosting Layer are not present in this release. As mentioned, if you wish to study up on WCF the older *Patterns in Action* version 4.0 includes an extensive WCF section. Download it from our website for free; links are available on your download page. Note that version 4.0 requires VS 2010.

The 8 Projects

Next we will review each of the projects in *Patterns in Action 4.5*. To view the projects, expand the folders. A total of 8 projects will be exposed.



The 8 projects in *Patterns in Action 4.5*

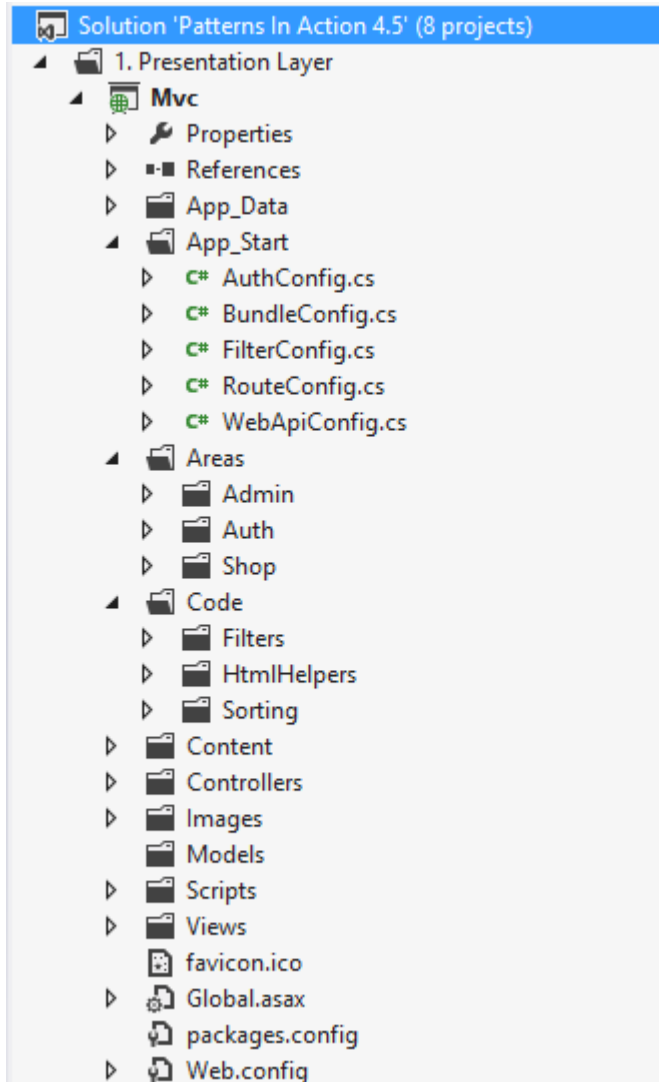
Here is a summary of the list of projects.

Project	Summary
MVC	The MVC web application (MVC pattern)
MVC.Tests	Unit tests for the MVC application
Web Forms	The Web Forms web application
Windows Forms	The Windows Forms application (MVP pattern)
WPF	The WPF application (MVVM pattern)
Action Service	The Application service (service layer)
Business Objects	Business objects with Business rules (business layer)
Data Objects	Data Access objects (data access layer)

Next, we will examine each project in more detail:

MVC Application

Expand the MVC folder, select the 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:



The folders already reveal the notion of the MVC pattern with Models, Views, and Controllers folders and classes. Within the Areas folders you will see this repeated. Microsoft built this UI platform based on the popular MVC Design Pattern. Details of the actual MVC pattern are explained in the Patterns and Practices section later in this document.

Let's further examine this project. The MVC application is an e-commerce app with its own LocalDb database under the \App-Data folder named Action1.mdf. The \App_Start folder contains common startup files that are used to setup items such as the OAuth authentication, routing, and bundling of css and JavaScript assets. In prior version these

activities were mostly performed in the Global.asax file but they have been factored out into their own start folder.

Areas

In Visual Studio each new MVC project will automatically have \Models, \Views, and \Controllers folders. The idea is that 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, this works only for small apps. When your MVC project starts getting more than, say, a couple dozen pages you will discover that managing a large number of files in a single folder becomes unwieldy. To solve this, Microsoft introduced *Areas* which are an effective way to organize and partition the application in functional areas (also called Modules). Each area has its own set of \Models, \Views, and \Controllers folders. For any non-trivial MVC application you should make use of Areas.

Pattern in Action 4.5 has 3 Areas. Admin, Auth, and Shop: they reflect the Administration, Authentication and Shopping functional areas or modules. 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 Orders action method renders a list of customer orders and the Products action method renders the Products 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. Note: in *Spark 4.5* we also demonstrate the use of the Http PUT and DELETE verbs.

Outside the \Areas folder there is just one other Controller class and one View. They are located at the root level \Views and \Controllers folders and their only task is to render the home page. In \Controllers you find HomeController with an action method Index and the associated View resides in \Views\Home. Alternatively, we could have created a separate Home Area and always have all models, views, and controllers in Areas. This is strictly a matter of preference.

The `_Layout.cshtml` (or `.vbhtml`) master page is located in the `\View\Shared` folder. It provides the standard look-and-feel for the entire application and includes a header, breadcrumbs, and menu for every page. The `_Layout` file contains the custom Html helper called `Html.BreadCrumbs`. Breadcrumbs are discussed next.

Breadcrumbs

Open `_Layout.cshtml` (`.vbhtml`) and locate the `Html.BreadCrumb` helper:

Breadcrumb helper:

```
@* Subheader *@

<div id="subheader">
    <div id="subheader-left"></div>
    <div id="subheader-right">
        <span>@Html.BreadCrumbs()</span>
    </div>
</div>
<div style="height: 4px; background: #c0ebf2;"></div>
```

The underlying code for this helper can be found under the `\Code\HtmlHelpers` folder with the `BreadCrumb` and `BreadCrumbHelper` classes. Open any Action method and you will see that the actual breadcrumbs are specified in each Action method.

Custom breadcrumbs

```
[HttpGet]
public ActionResult Administration()
{
    ViewBag.Crums = new List<BreadCrumb>();
    ViewBag.Crums.Add(new BreadCrumb { Title = "home", Url = "/" });
    ViewBag.Crums.Add(new BreadCrumb { Title = "administration" });

    ViewBag.Menu = "administration";
    return View();
}
```

This requires some coding but it also gives you total control. Later you will see that the breadcrumbs in the Web Forms application follow a very different model. There we use

the SiteMapPath control which renders the breadcrumbs automatically using information retrieved from a *web.sitemap* configuration file.

Menus

Previous versions of Patterns in Action included an Html.Menu helper but this has been replaced with a simple ViewBag variable. In each action method the current menu is assigned to ViewBag.Menu. It is used in the View to highlight the selected menu; very simple indeed.

Menu Assignment

```
[HttpGet]
public ActionResult Administration()
{
    ViewBag.Crumbs = new List<Breadcrumb>();
    ViewBag.Crumbs.Add(new Breadcrumb { Title = "home", Url = "/" });
    ViewBag.Crumbs.Add(new Breadcrumb { Title = "administration" });

    ViewBag.Menu = "administration";
    return View();
}
```

This will highlight the *administration* menu is highlighted.

Sorting

The application allows the user to sort collections: they are Products, Members, and Orders. This is done by clicking on the table headers with the help of an Html helper called Html.Sorter. On the Orders view, for example, you can see the custom Html.Sorter helpers in action.

Sort helpers

```
<tr class="table-header">
    <td class="tac">@Html.Sorter(Model.Members, "Id", "MemberId", "asc") </td>
    <td class="tal">@Html.Sorter(Model.Members, "Member Name", "CompanyName", '
    <td class="tac">@Html.Sorter(Model.Members, "Country", "Country", "asc") </
    <td class="tac">@Html.Sorter(Model.Members, "# Orders", "NumOrders", "asc")
    <td class="tac">@Html.Sorter(Model.Members, "Last Order", "LastOrderDate",
    <td class="tac">View</td>
</tr>
```

Three items under the \Code\Sorting folder provide the basis for 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

```
[HttpGet]
public ActionResult Orders(string sort = "memberId", string order = "desc")
{
    ViewBag.Crumbs = new List<Breadcrumb>();
    ViewBag.Crumbs.Add(new Breadcrumb { Title = "home", Url = "/" });
    ViewBag.Crumbs.Add(new Breadcrumb { Title = "administration", Url = "/adm
    ViewBag.Crumbs.Add(new Breadcrumb { Title = "orders" });

    ViewBag.Menu = "orders";

    var model = new OrdersModel();
    var members = service.GetMembersWithOrderStatistics(sort + " " + order);
    var memberModels = Mapper.Map<List<Member>, List<MemberModel>>(members);
    model.Members = new SortedList<MemberModel>(memberModels, sort, order);

    return View(model);
}
```

First, the action method receives sort and order argument values. If none are specified then their default values are assigned. Next, members are retrieved in the given sort order. The members are mapped to MemberModels using the AutoMapper third party tool (available under NuGet). Then a typed SortedList item is populated with the sorted list and the current sort and order are passed in.

This information is passed to the view where it is then used to display the list and configure the different sort controls. The Sorter extension method in the Html Sorters is interesting as it demonstrates how to issue postbacks using jQuery. Two hidden controls on the page, named Sort and Order, maintain the current selections.

Note: *Spark 4.5* expands on this with a more comprehensive solution that includes searching, sorting, filtering and pagination.

Validation

For validation, MVC makes use of Data Annotations. To see an example, open up the MemberModel class located under \Areas\Admin\Models folder.

Data Annotation

```
public class MemberModel
{
    [DisplayName("Id")]
    public int MemberId { get; set; }

    [DisplayName("Email")]
    [Required(ErrorMessage = "Email is required.")]
    [StringLength(100, ErrorMessage = "Email can be at most 100 chara")]
    [EmailAddress(ErrorMessage = "Invalid Email")]
    public string Email { get; set; }

    [DisplayName("Company Name")]
    [Required(ErrorMessage = "Company Name is required.")]
    [StringLength(30, ErrorMessage = "Company Name can be at most 30")]
    public string CompanyName { get; set; }

    [Required(ErrorMessage = "City is required.")]
    [StringLength(30, ErrorMessage = "City can be at most 30 characte")]
    public string City { get; set; }

    [Required(ErrorMessage = "Country is required.")]
    [StringLength(30, ErrorMessage = "Country can be at most 30 chara")]
    public string Country { 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. [EmailAddress] validates emails. When editing a member 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 4.5*:

Member Details

[< back to member list](#)

Id	0
Email	<input type="text" value="bad email"/>
Company Name	<input type="text"/>
City	<input type="text"/>
Country	<input type="text"/>



Invalid Email
 Company Name is required.
 City is required.
 Country is required.

The actual validation occurs in the action method by calling the `ModelState.IsValid` validation property. It returns false when something is not valid. In MVC, data annotations make field level validation very easy. There are quite a few built-in data annotation attributes, and we have shown just a small sample and how to use them.

Assets

The `\Content` folder contains the application css file named `app.css`. The `\Scripts` folder contains the standard JavaScript files. The `\Images` folder contains application images including product and member images.

jQuery is included at the bottom of the `_Layout` file, just before the `</body>` tag. Placing the file there will create a better experience for the user because the page rendering will take place earlier. It does not have to wait until the JavaScript is parsed. It is said, that this puts 'the pixels earlier on the screen'. Of course, the total load time will be the same but to the user the page appears faster.

The jQuery script tag is followed by a `@RenderSection` statement which renders the view specific JavaScript sections at that location. It is important that this is placed *after* the jQuery script or else your JavaScript code will not be able to use jQuery.

```

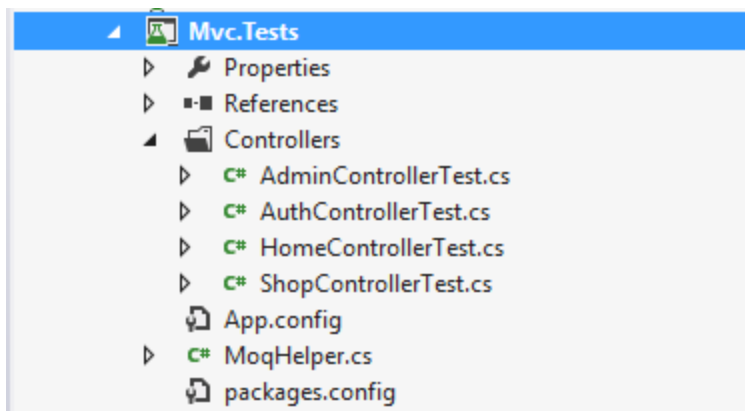
86
87     </div>
88     <script src="/scripts/jquery-1.7.1.min.js" type="text/javascript" ></script>
89     @RenderSection("scripts", required: false)
90 </body>
91 </html>

```

MVC Test Application

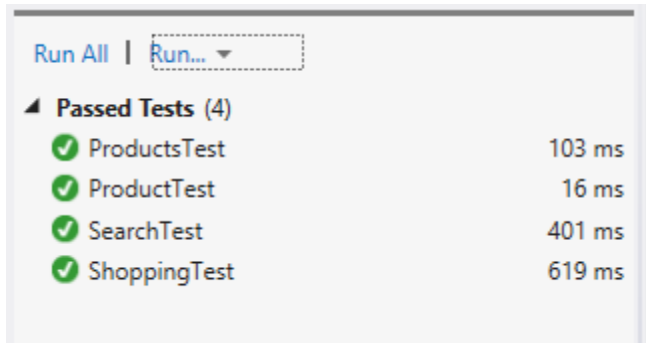
Next we will see how to unit test an MVC application. The main idea of unit testing is to test basic units of functionality. Ideally you can run these tests quickly to verify that a change you made does not accidentally break something that used to work before.

This project is designed to perform unit testing on the MVC app. It is a small project. Here is a screen shot.



Here we use Microsoft's built in unit testing platform called *mstest* and a 3rd party tool called *Moq*. The testing platform *mstest* is built-in Visual Studio and *Moq* is available from NuGet. If you select the menu Test->Run->All Tests you should see that all tests

pass. Note: running for the first time may take a few moments before you see the results.



MVC testing focuses on 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 Views are not part of these tests, but what about the Model? Testing a Model is tricky because it involves data base access which 1) is relatively slow and 2) it often is non-repeatable. What does it mean to be non-repeatable? Here is an example. Say your application allows users to register themselves and create an account. A user should create only a single account and this is enforced by a unique index on the email column which prevents insertion of duplicate records. The business reasons for this restriction are clear, but it makes running repeatable units tests hard; once a user has registered there is no way to run registering this user again. 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 mocks) the functionality of a real object in a predictable and controlled way. To simulate the Model we will create mock objects for the Service object (the service layer).

Mocking is a technique in which classes are stubbed out and mocked by a mock class. They behave just like the real objects and mock objects have the same interface as the objects they simulate. This is one of the reasons why the Service class is implemented with an interface IService. The controller class being tested is unaware that it executes mock objects rather than the real Service object.

Dependency Injection

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

```
public class ShopController : Controller
{
    IService service { get; set; }

    // static constructor. establishes AutoMapper object maps

    static ShopController()
    {
        Mapper.CreateMap<Product, ProductModel>()
            .ForMember(dest => dest.UnitPrice, opt => opt.MapFrom(src => string
        Mapper.CreateMap<ProductModel, Product>());
    }

    // default constructor

    public ShopController() : this(new Service()) { }

    // overloaded 'injectable' constructor
    // ** Constructor Dependency Injection (DI).

    public ShopController(IService service) { this.service = service; }
```

Ignore the static constructor in the above code, but all other code is relevant. The controller has a private property named service of type IService. This property is initialized in one of the controller’s constructors; either the default one or the overloaded constructor. When running the MVC application normally the default constructor is called which creates the standard Service object. However, in the Test project the

overloaded constructor is explicitly called with a mocked Service object. This assigns the mocked Service to the local service property. This is an example of *constructor injection*. Generally, dependency injection is a fairly complex topic, but the above example is as simple as it gets.

The relevant code where a mocked Service object is passed into the constructor can be found in the Test project in the ShopControllerTest class:

```
ShopController CreateShopController()
{
    // Note: this is where DI (Dependency Injection) takes place.
    // the service layer is injected (via the constructor) into the controller.

    return new ShopController(mockService.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 I call you with these arguments then this is how you respond". Essentially, Setup is the process where you 'teach' the mock objects to respond in a certain way when called with a particular set of arguments.

Here is an example:

```
// setup for getting a list of products

var products = new List<Product> { new Product { ProductId = 1, ProductName = "test-product" } };
mockService.Setup(s => s.GetProductsByCategory(It.IsAny<int>()), "unitprice asc").Returns(products);
```

It states that when you call GetProductsByCategory on the mocked Service with any categoryId and a sortExpression of "unitprice asc" then you return the list of products defined in the line before (in this case just 1 product). This is a very simple example of Moq. Please be aware that Moq offers a lot of flexibility by its use of lambda methods

When designing and developing tests you may run into IOC (Inversion of Control) containers which are a popular technique to control which classes are instantiated at runtime (typically you configure this with XML files). In this testing environment we could

have used an IOC container to control the Service object (the real or mocked one) 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 because they tend to be rather complex and require XML configuration files (remember our Convention over Configuration paradigm). Our tests are simple and 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 is not used in our example. Finally, each test method is tagged with the [TestMethod] attribute.

Unit test methods typically have a narrow focus and each should test one thing or one behavior only. Testing methods are normally arranged according to a 3 step pattern, called AAA (Arrange-Act-Assert). Arrange is where you create and initialize the items involved in the test. Act is where you execute the test. Finally, 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));

    // check product name
    Assert.AreEqual((result.ViewData.Model as ProductModel).ProductName, "test-product");
}
```

The Controller classes being tested are located under the \Controllers folder. Please note that of the controller classes in this folder only the ShopControllerTest is implemented. The purpose of this project is to demonstrate 1) MVC unit testing, 2) dependency injection, and 3) mocking; not to get full code coverage.

To reach adequate test coverage would involve a significant effort and include several dozens of unit test cases. Just as an aside: automatic testing may give you a sense that you are covering your bases, 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 discussed earlier that some controller action methods only respond to Http GET requests or Http POST requests. The [HttpGet] and [HttpPost] attributes are used for this purpose. Simulating POST and GET requests in a testing environment requires some extra code. We used a class named MoqHelper to create fake HttpContext or fake ControllerContext objects (by the way, ‘fake’ is an official testing term). MoqHelper was published by Microsoft’s Hanselman and it is available here:

<http://www.hanselman.com/blog/ASPNETMVCSessionAtMix08TDDAndMvcMockHelpers.aspx>

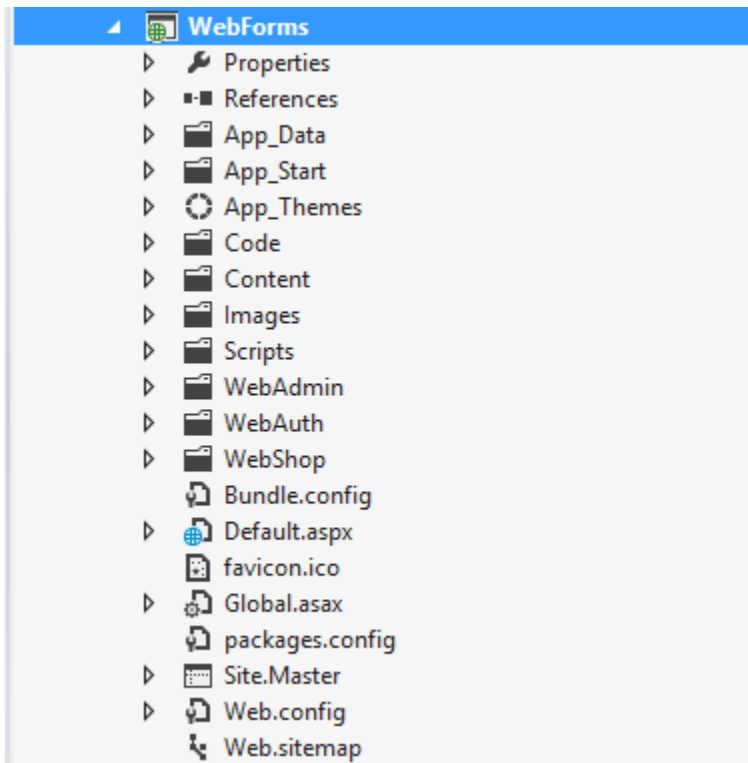
It is used in the SearchTest method to test a postback (using the POST verb) in which a search is performed. You can find the MoqHelper class is under the \Moq folder.

As the MVC test application demonstrates it is important that you construct your controller classes and service 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. This means that if your organization is committed to testing they will need to allocate the necessary resources (skills, money, time, etc.) as well.

In this example we have offered a quick peek at how to perform MVC unit testing in Visual Studio. Next we discuss the Web Forms Application.

Web Forms Application

Expand the Web Forms Application project until you see the view below. Set this app to be the default Startup Project (it turns bold).



From the outside the MVC and Web Forms application have a lot in common; their functionality and look-and-feel are the same and they consume the same Service layer. However, under the hood things are very different.

WebForms has no concept of controllers, models or views. Instead, it is built around a system of pages (.aspx) each with their own code behind.

The Default page, Global.asax, and Site.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 are in the \WebAdmin folder; this is where the admin manages the web site.

Authentication pages are named Login.aspx and Logout.aspx. They reside in the \WebAuth folder. Access to the web pages in the \Admin folder is restricted to authenticated administrators and requires a login. Security is also handled by SimpleMembership. The Web Forms app maintains its own copy of the LocalDb database: it is named Action2.mdf (ActionVb2.mdf in VB) and is located in the \App_Data folder.

The \Code folder contains several utility classes. One is PageBase.cs which is the base class to all pages in this app. It contains functionality that is useful to all pages. Another class is UrlMaker, which helps in building consistent Urls that are used throughout the application for routing purposes. The class named PriceRange.cs supports the search page with a price range dropdown.

Under the \Code folder are three subfolders: \Controls, \Logging, and \ViewState. 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. ViewState manages the ViewState on the server side (it demonstrates the Provider design pattern discussed in more detail at the end of this section). Finally the \Logging folder has a logging facility that allows you to log errors to any device (email, database, file, etc.). It demonstrates the Singleton pattern.

The \App_Themes folder contains themes that control the overall appearance and look and feel of an application. *Patterns in Action 4.5* has a new theme, named 'Theme45'. Theme45.css is a style sheet which assists in formatting of the HTML elements. Theme45.skin contains skin definitions to control the appearance of several of the built-in ASP.NET controls.

The \Images folder has both customer and product images.

A lesser known feature in Web Forms is the SiteMapPath control. A file named *Web.sitemap* defines the hierarchical structure of the web site and it forms the data

source for the SiteMapPath control which displays the breadcrumbs along the top of each page.

When using Master pages, .NET developers usually require 1) access to the Master page from the content pages, and 2) access to the content page from the Master page. *Patterns in Action 4.5* offers both; the code behind page of the Master page demonstrates how this bi-directional access is handled.

Web Forms includes a couple of tools that are important for SEO (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 (these meta tag values are 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 directly 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=3340d99af you can now have this: www.company.com/users/john-travolta. The routing system is great, but be mindful that 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. To explore routing check out global.asax (where the routes are registered). Also see UrlMaker where routes are built. UrlMaker prevents the use of *magic strings* throughout the application).

The ViewState in ASP.NET is a feature that makes it easy to keep track of page state. The downside is that it adds 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. In fact, the increase in 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 (or Dictionary) data structure. All three providers are implemented in *Patterns in Action 4.5* using Microsoft's Provider design pattern.

ViewState and the Provider design pattern

As far as patterns are concerned, the built-in .NET Provider design pattern is rather comprehensive and large. It offers an infrastructure in which you need to implement several different elements. 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.

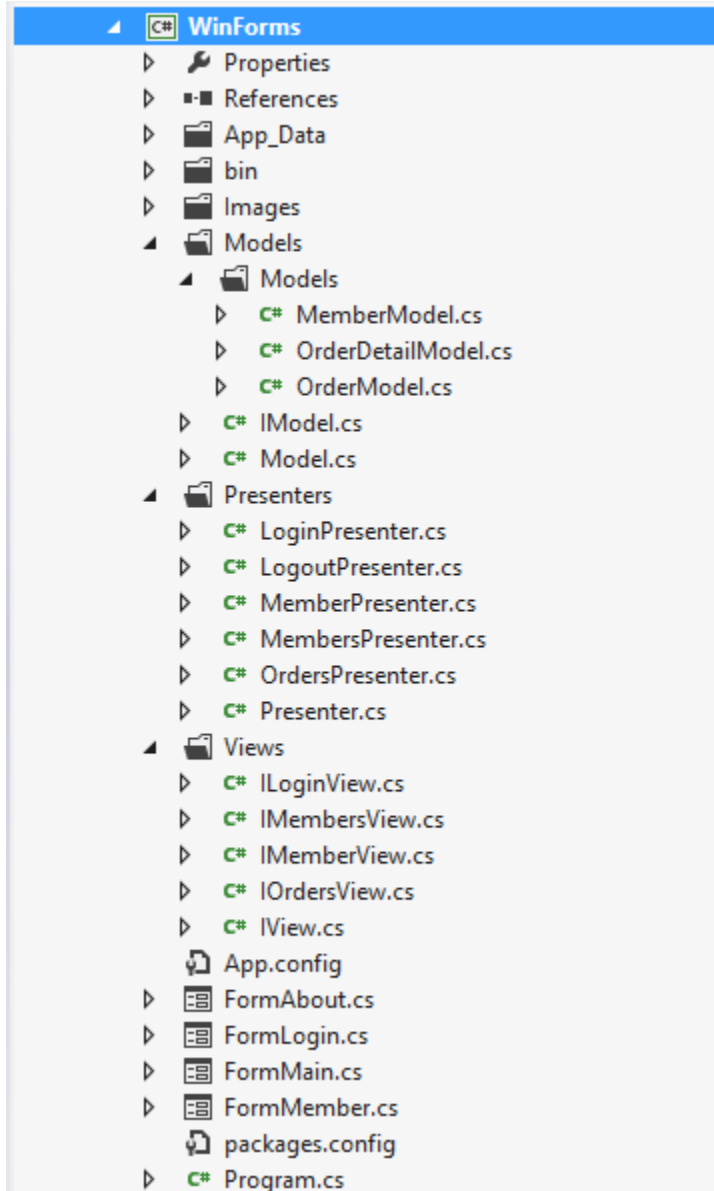
Folder \ProviderBase contains the classes that perform the ‘plumbing’ for the viewstate provider. Abstract class ViewStateProviderBase declares abstract methods that need to be implemented by the ViewStateProvider instances; they are SavePageState and LoadPageState. ViewStateProviderCollection is a collection 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 can 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 4.5* is written for educational purposes only and may or may not work under all scenarios and configurations.

Windows Forms Application

Select and expand the Windows Forms folder. Open up the Models, Presenters, and Views folders. Your explorer will look like the image below:



The Windows Forms Project

The Windows Forms Application is a traditional Windows application. It is an older technology but still widely used. Like the two web projects discussed before, this application also communicates exclusively with Service layer and has a LocalDb database called Action3.mdf (or ActionVb3.mdf).

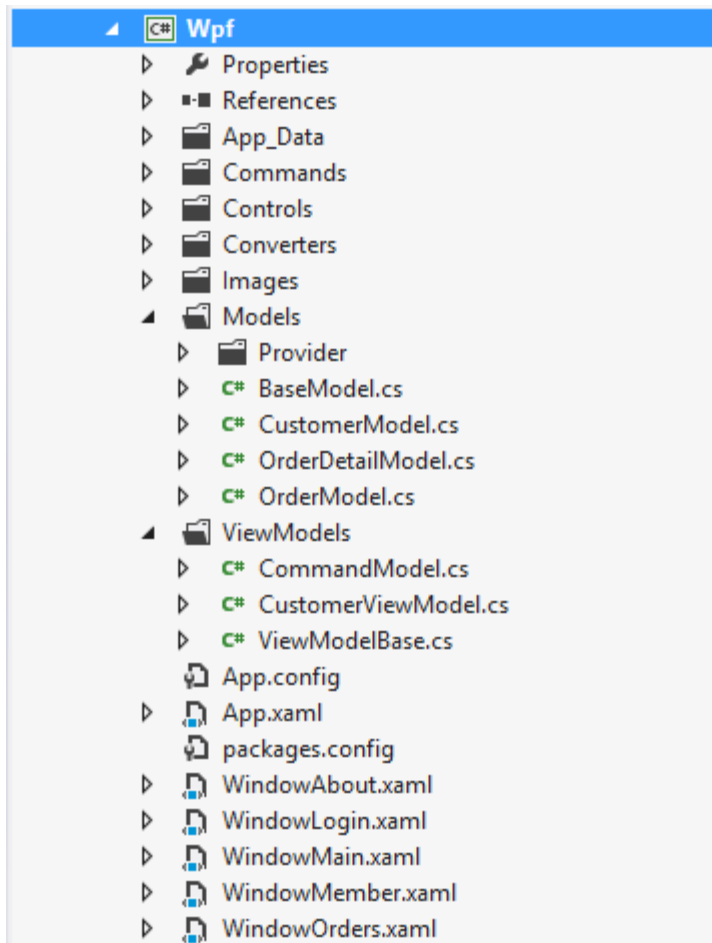
The Windows Forms application is built around a MVP (Model View Presenter) design pattern. Three folders Models, Presenters, and Views represent each of the parts. The forms in the Application derive from the View interfaces defined in the Views folder. Details of the MVP pattern are explained in the Patterns and Practices section later in this document.

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 approach 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 Automapper utility maps data transfer objects to the local business objects and vice versa. The three client-side business objects are located in the \Models\Models folder.

WPF Application

Select and expand the WPF project until you see the view below:



The WPF Application

The application is built around the Model ViewModel View (MVVM) design pattern. Model and ViewModel classes reside in similarly named folders. The View is represented by the Forms in the WPF Application, that is, the Forms are the Views. Details on the MVVM pattern are explained in the Patterns and Practices section later in this document.

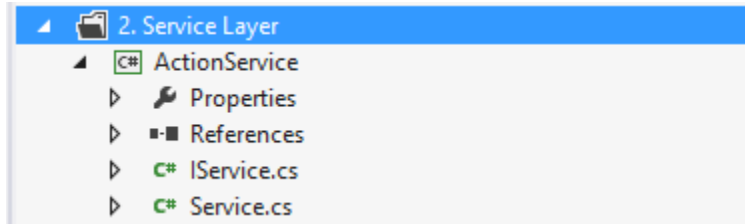
The application has five windows (the Window*.xaml files). Several WPF specific classes exist in the \Commands and \Converters folders. The \Controls folder contains a control that facilitates the glowing menus.

The WPF Model has three business model objects in the \Models 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 Automapper tool maps DTOs (Data Transfer Objects – which is a pattern) to Business Model Objects and vice versa. The WPF client receives DTOs from the WCF service.

The \ViewModels folder has only three classes which represent 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.

Action Service

Expand the ActionService project.



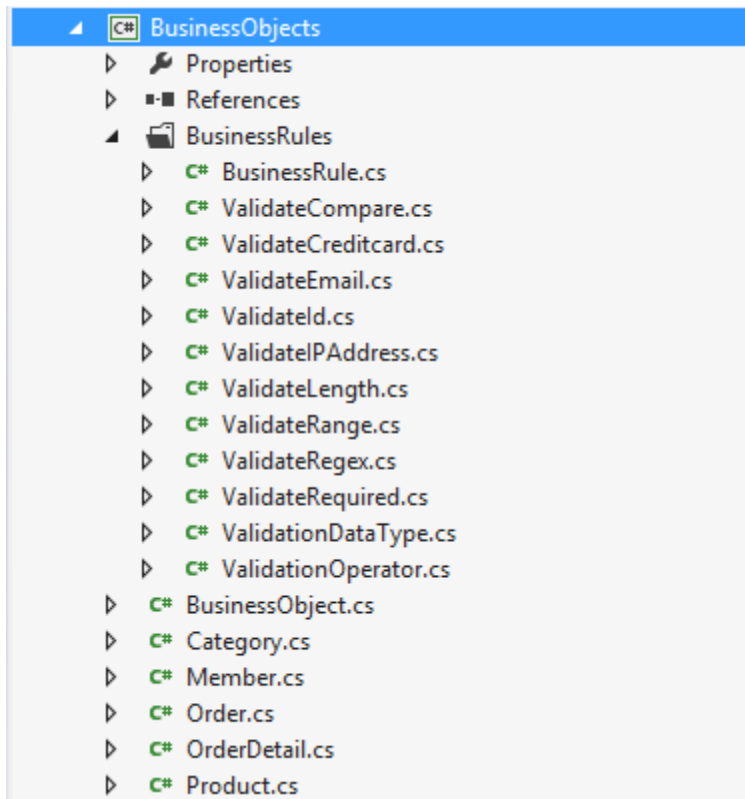
This project is small with just two files: `IService` and `Service`. It is the façade (Façade Design Pattern) to all underlying data and business layers. All UI clients interact with the API on this façade.

The `IService` file offers a single interface. Many applications break these up in separate Repositories, for example one for Customers, one for Products, one for Orders, etc. However, our API is small so we decided to keep them in a single interface. In the `IService` interface file you will see the members clustered to reflect Repositories. By the way, a far more robust Service and Repository system is demonstrated in our *Spark 4.5* application framework.

The `Service` class is the implementation of the `IService` interface. It contains references to all five Data Access Object (DAO) interfaces. This is where the `DataProvider` application setting in `Web.config` and `App.config` is read to determine which Data Access technology is used: ADO.NET, Linq-to-Sql, or the Entity Framework. The code that deals with DAO selection relies heavily on the Factory design pattern: it's a factory that creates the correct Data Access Objects. Most Service method implementations are very simple as they directly map to the corresponding DAO methods; as mentioned before, there is not rule that states that they have to be the same.

BusinessObjects

Expand the BusinessObjects Project until you see the view below:



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 4.5* are: Category, Product, Member, Order, and Order Detail.

A small business rules engine is built in *Patterns in Action 4.5*. It is implemented by 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, such as ValidateCompare, ValidateLength, ValidateRange, etc.

Let's look at the Member class and see how the business rules work. The member's validation 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 constructor. Once the object is populated with data, the application can call Validate on the Member object. If Validate returns false, the

validation has failed. You can check the Errors collection on the object to get list of error messages. The application logic can then decide how to respond to these errors.

Depending on the requirements, business rules may be encoded in these classes. In *Patterns in Action 4.5* the rules are relatively straightforward validation-type business rules. In your own projects you may have more complex business rules involving multiple business object types. Here's an example: if the user has placed more than 3 orders with a total value over \$500 in the last month, and the order includes a Dell computer and they have been a customer for more than 3 years, then offer a 10% discount.

In most ASP.NET web applications these complex rules are typically implemented in the code behind (Web Forms) or action methods (MVC). The *Spark 4.5* app shows an alternative place; it has a file named Service.cs (a thin layer over the Repository layer) and it easily handles these types of complex business rules, as well as associated transactions (i.e. Unit of Work pattern).

There are many different ways to implement and enforce validation and business rules. For simple property based validation a good alternative is the use of DataAnnotations which were introduced by Microsoft when they released their Dynamic Data controls. The MVC platform has further enhanced these making it easy to use DataAnnotations for validation and rendering purposes. It also works on the client, that is, these rules can be validated on the browser before data is posted back to the server. So, when using MVC exclusively, DataAnnotations are a good choice for validation.

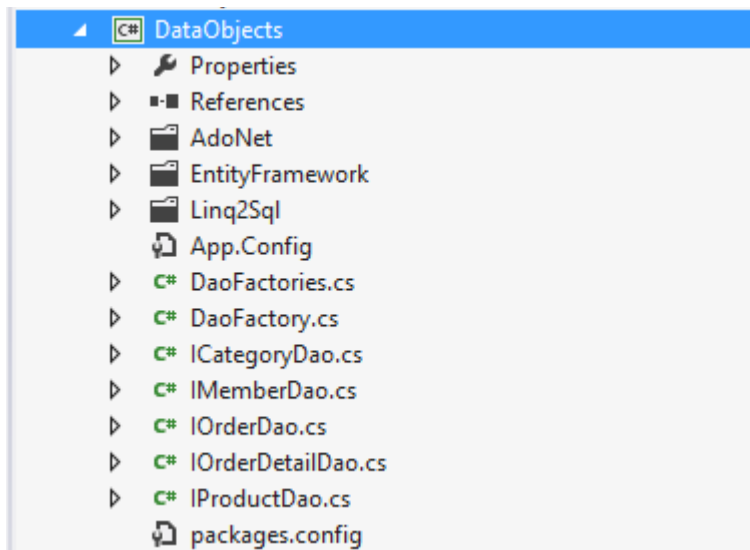
Object Persistence

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 by dedicated Data Access Objects (Daos) in the Data Layer. Daos accept and return values from and to business objects. The Data Layer accepts business objects, then gets and sets their values via object properties. This requires intimate knowledge

about the object model, as well as the data model, and therefore the Data Layer is the intermediary between the two models (object model and data model).

DataObjects

Expand the DataObjects project. This is what you will see:



The data layer with Data Objects project

The Data Layer resides in a class library project named DataObjects. It offers three different data access technologies: ADO.NET, Linq-to-Sql, or the Entity Framework. The implementation files for each technology are in clearly named folders: \AdoNet, \Linq2Sql, and \EntityFramework. In web.config you specify which provider to use (the details were presented earlier in this document).

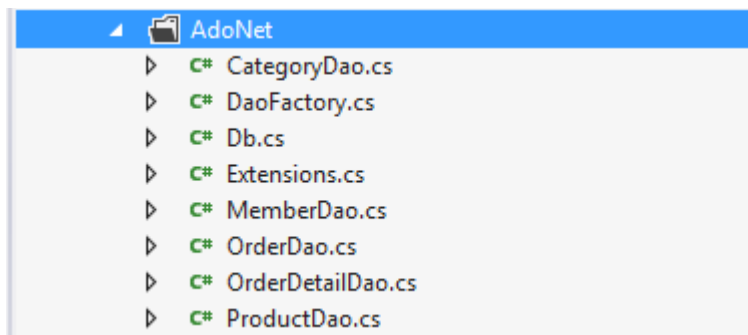
Five Dao (Data Access Object) interfaces, ICategoryDao, IProductDoa, IMemberDao, IOrderDao, and IOrderDetailsDao define the interface between Business Layer and Data Layer. The Business Layer does not know anything about data access and these interfaces facilitate the persistence of the business objects. Each business class has its own Dao. So, business object persistence is entirely handled by the Data Layer which maps the object model to the relational data model and vice versa.

Two other classes are significant: IDaoFactory and DaoFactories. The IDaoFactory interface contains methods that return IDoa* objects. The DoaFactories is an interesting pattern (or meta pattern): it is a factory of factories. Given a provider, it will return the correct DaoFactory: AdoNetDaoFactory, Linq2SqlDoaFactory, or EntityFrameworkDaoFactory. These factories, in turn, return business object specific Dao objects (for example ProductDao or OrderDao). These last ones are provider specific and reside in the data access technology folders: \AdoNet, \Linq2Sql, and \EntityFramework.

Let's look at the details of the three data access technologies.

ADO.NET as the data access platform

ADO.NET data access classes are located in the \AdoNet folder.



It contains the 5 Dao implementations and an ADO.NET specific DaoFactory.

The Db.cs class is the real workhorse of the ADO.NET Data Layer. It handles all ADO.NET data access calls and shields the rest of the system from low level database concerns (connections, commands, data readers, etc.).

Db.cs uses the very fast DbDataReader class (often referred to as a 'fire hose' data stream) to get the data from the database. Furthermore it uses DbParameters which helps prevent SQL Injection attacks. The sql parameters are passed as a linear object[]

array. This is a choice we made, because by using an object array, the client does not have to know anything about DbParameters. However, you could easily change this to an array of DbParameters being passed into the Db methods. Generic 'make' delegate methods are passed into the Read method which makes creating and populating business objects easy and flexible.

There are a number of extension methods in the data access layer that you need to be aware of. They help in getting data in and out the database. First there is the AddParameters method in the Db.cs file, and secondly, there is a group of extensions methods that help getting data in and out of the DbDataReader.

The AddParameters extension method 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 you could bypass the AddParameters extension method.

A second set of extension methods is available in Extensions.cs. 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 ProductDao and find the Make lambda expression. In it, you will see the AsId(), AsString(), and AsDouble() extension methods in action. They are used in all Dao objects.


```
private static Func<IDataReader, Product> Make = reader =>
    new Product
    {
        ProductId = reader["ProductId"].AsId(),
        ProductName = reader["ProductName"].AsString(),
        Weight = reader["Weight"].AsString(),
        UnitPrice = reader["UnitPrice"].AsDouble(),
        UnitsInStock = reader["UnitsInStock"].AsInt()
    };
```



Now scroll down a bit in ProductDao 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. Each Dao (ProductDao, MemberDao, etc.) has exactly

one Make and one Take method. The Take is executed in the Dao itself, whereas the Make is passed as a delegate to the Db class and executes there when reading the data reader. Here is the code for the Take method:

```
private object[] Take(Product product)
{
    return new object[]
    {
        "@ProductId", product.ProductId,
        "@ProductName", product.ProductName,
        "@Weight", product.Weight,
        "@UnitPrice", product.UnitPrice,
        "@UnitsInStock", product.UnitsInStock
    };
}
```

A question that comes up regularly is how to use Db.cs with stored procedures. The answer is simple: just 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 (the red arrows show the changes):

```
public static List<T> Read<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;
        }
    }
}
```

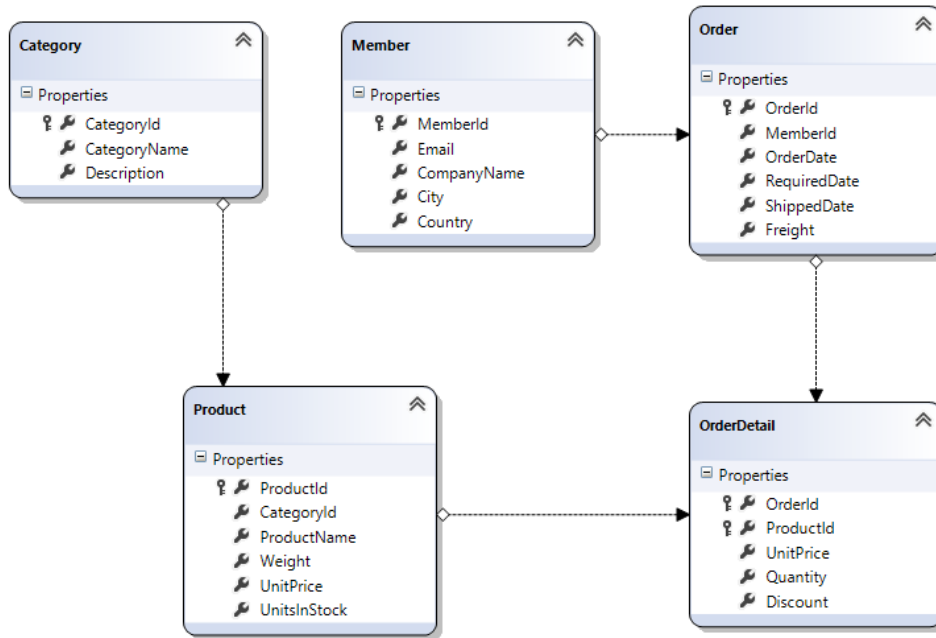
Linq-to-Sql as the data access platform

Some time back, Microsoft announced that it will not further develop or enhance Linq2Sql in favor of Entity Framework. This is unfortunate because, compared to the Entity Framework, Linq-to-Sql is a simple, light-weight, and fast data access technology. Linq-to-Sql is still used by numerous applications that were built over the last several years.

Linq-to-Sql is an object relational mapping library 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. One limitation is that Linq-to-Sql is only available for SQL Server.

Classes that support Linq-to-Sql data access are found under folder \Linq2Sql. Just like ADO.NET, the Linq-to-Sql data access classes implement the five Dao interfaces: ICategoryDao, IProductDao, IMemberDao, IOrderDao, and IOrderDetailDao. Likewise, Linq-to-Sql also implements the IDaoFactory interface. You find all implementations in the \Linq2Sql 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 next figure.



The Object Relational Modeler with *Patterns in Action 4.5* 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 business object needs to be updated in the database, the DataContext with the original values of the record does not exist anymore and therefore Linq-to-Sql cannot perform the 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.

In previous versions of *Patterns in Action* we included on every table in our database a column named Version which was of type 'timestamp'. In this release the timestamp was removed, but below is the original discussion (for those interested).

Every entity has also a property called Version. 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 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 in the page. 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 Web Forms) or in hidden variables on the page (MVC).

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 4.5 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 AutoMapper tool is used to map entities to business objects and vice versa. In *Patterns in Action 4.5* 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.

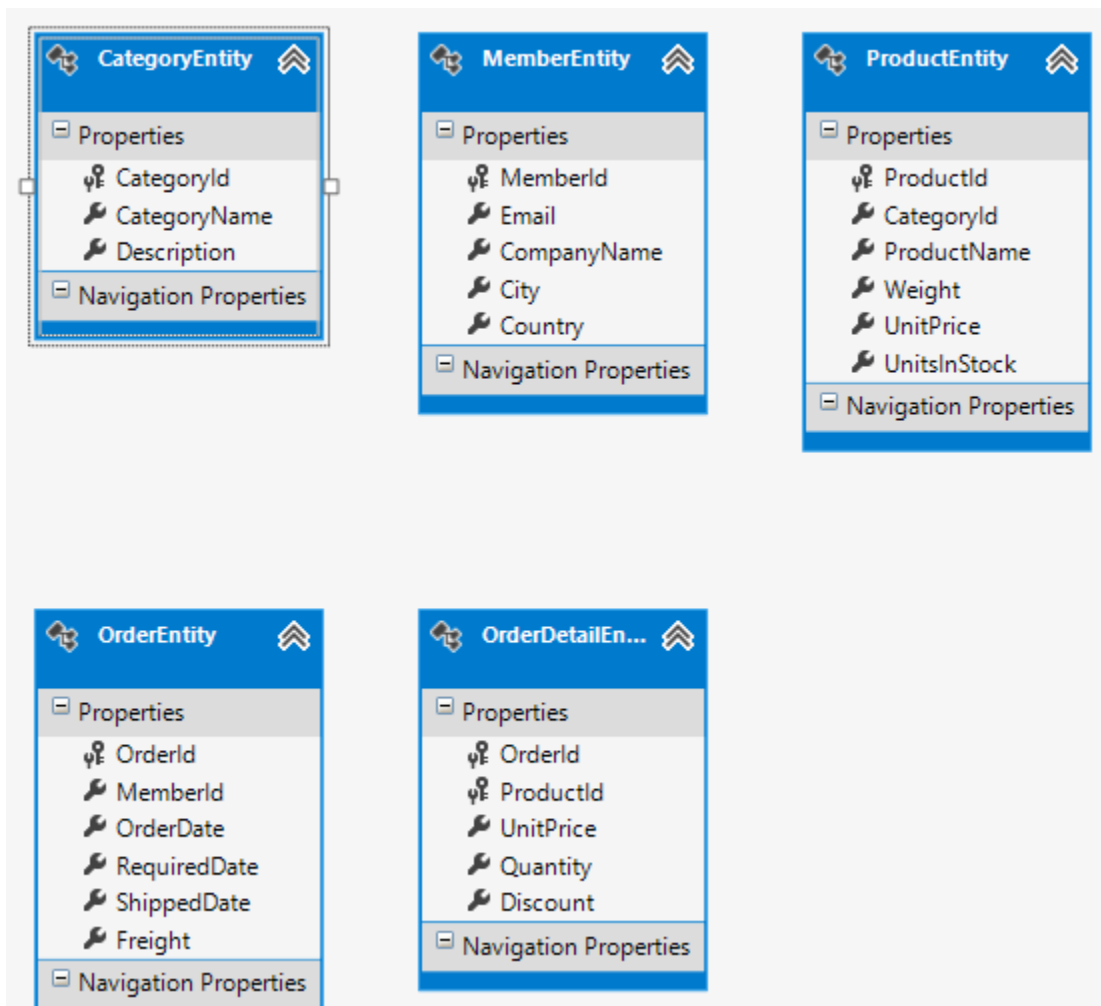
Entity Framework as the data access platform

Among other things, the Entity Framework was designed to address the object-relational 'impedance mismatch', that is, the mismatch between object model and relational data models. Microsoft is putting enormous resources into development and refinement of the Entity Framework. As a data provider the Entity Framework is by far the largest and most complex system of the three we use.

In this project, the Entity Framework data access files are located under folder named \EntityFramework. Just like the other data access platforms, Entity Framework also implements the five Dao interfaces: ICategoryDao, IProductDao, IMemberDao, IOrderDao, and IOrderDetailDao as well as the IDaoFactory interface.

The file named Action.edmx represents the object-relational model and mappings for the Entity Framework. It contains the entities that are created by selecting the appropriate tables from a list of database tables when creating the .edmx file. This modeling tool allows you to easily and quickly create entities (similar to business objects). We appended the word Entity to entity names to avoid conflicts with the Business objects and Linq2Sql entities which are named similarly

See figure below.

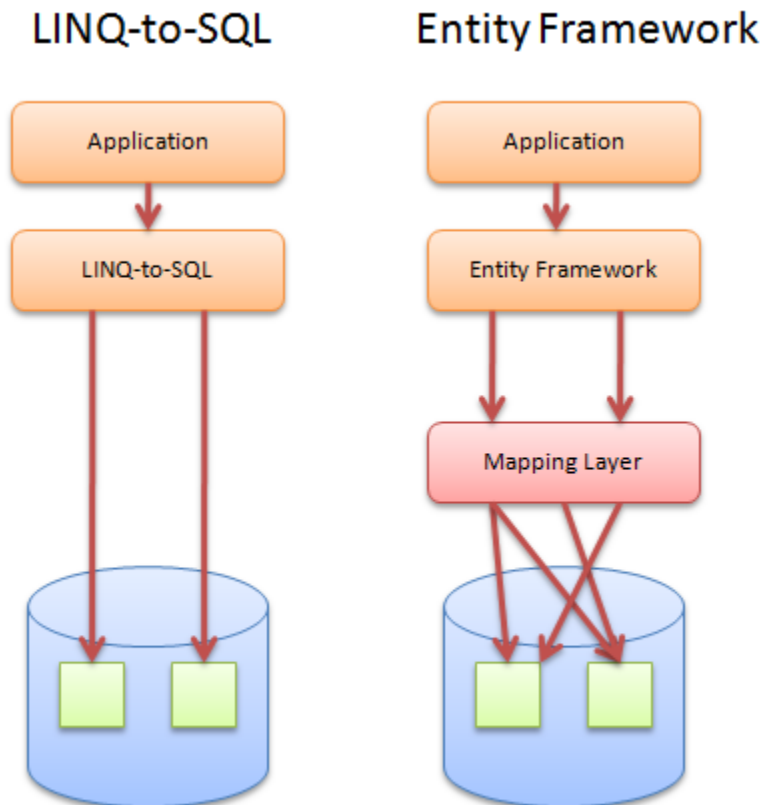


When comparing the Entity Framework implementations to Linq-to-Sql you will see more similarities than differences. In particular the LINQ expressions appear pretty much the same. In reality, however, Linq-to-Sql is only a light-weight version of the Entity Framework. Among other things it only works with SQL Server whereas the 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 today we have ADO.NET, Linq-to-Sql, and Entity Framework. Some organizations struggle to standardize their data access stack and today they are faced with these primary choices offered by Microsoft: ADO.NET, Linq-to-Sql, or Entity Framework.

ADO.NET is a fast, effective, and a proven technology. It gives you full control over the 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 mentioned before, Linq-to-Sql is being phased out. This leaves the Entity Framework which is a large system with a fairly steep learning curve. Also, it does not always provide the flexibility and control you need. It is true that the Entity Framework offers the ability to quickly build object models and/or data models with options like Code First, Database First, Model First, etc. but at the cost of subsequent flexibility and performance.

Talking about performance here is an interesting, albeit extreme, example. Stackoverflow.com never used the Entity Framework, but was originally built on Linq-to-Sql. Although Linq-to-Sql is lightweight and fast, there came a point where their web site could barely support the enormous traffic it receives. Something needed to be done.

The Stackoverflow.com team decided to write a small, specialized, and highly optimized data access library that would serve their particular performance needs. The project was completed successfully and this library is what supports the site today. The library is called *Dapper* and the source code is available on Github. The code is small, but rather complex as it includes the use of Reflection Emit. One of the downsides of Dapper is that all SQL is hand-coded.

Dapper is what is called a Micro ORM (Object Relational Mapper). Other popular MicroORMs include Massive and PetaPoco. Interest in Micro ORMs is increasing as part of the 'small is beautiful' trend where developers need the ability to quickly create

applications that are simple, light-weight, and very fast. Our own *Spark 4.5* is also Micro ORM, although it goes beyond that by offering more functionality. All this and more is explained in the Spark 4.5 documentation and reference application.

For now, let's get back to Linq-to-Sql and the Entity Framework and see how to best handle dynamic queries.

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 a situation where you end up having to build lengthy case statements when constructing a late-bound LINQ expression.

To solve this, Microsoft has made available 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 4.5* it is used at several places in both Linq-to-Sql and Entity Framework. To see an example, open the `ProductDao` class in the `\EntityFramework` folder. 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 be some something like: "ProductName Desc". The relevant `OrderBy` method is underlined below.

```
public List<Product> GetProductsByCategory(int categoryId, string sortExpression)
{
    using (var context = new actionEntities())
    {
        var products = context.ProductEntities.Where(p => p.CategoryId == categoryId)
            .AsQueryable().OrderBy(sortExpression).ToList();
        return Mapper.Map<List<ProductEntity>, List<Product>>(products);
    }
}
```

It is interesting to note that to make Entity Framework more flexible, it resorts to the flexibility of native SQL, in the form of SQL snippets.

A file named `DynamicQuery` is located under the `\EntityFramework` folder that supports all these dynamic string-based query features. It is a nice addition to your personal LINQ toolset.

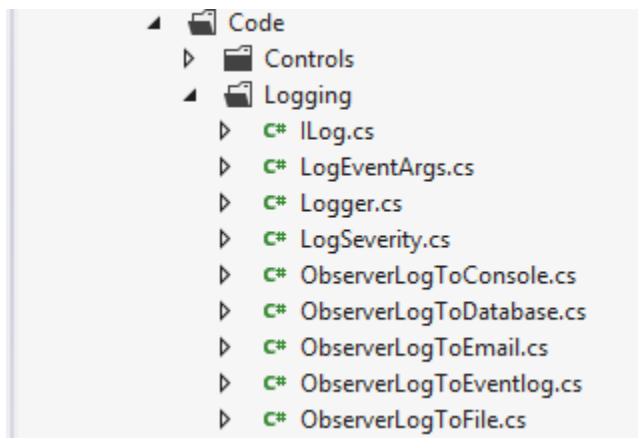
DAO Review

The Dao (Data Access Object) pattern in *Patterns in Action* offers data store independence. With a simple change in `web.config` you can run an entirely different data access provider. Using the Dao model, forces the business objects to expose all their properties and their data to allow the Dao objects to set and get the values. This limits the ability of the business objects to encapsulate their properties and data values, so this can be seen as a disadvantage of Dao.

As you will discover later on, *Spark 4.5* follows a very different philosophy in which business objects handle their own persistence. They have methods such as `Insert`, `Update`, `Delete`, etc. The idea is that the objects themselves have all the necessary information to determine what, where, and how to persist their data values. No 'outside knowledge' is required. To be clear, everything is a tradeoff and there is no right or wrong in this discussion, just a different point of view and preference. We prefer the business objects to handle their own persistence.

Error Logging

Error Logging is demonstrated in the Web Forms project. It is useful because it allows you to log warnings and errors that occur in the application. At this same time it demonstrates several patterns really well. Select this app and open the `\Logging` folder.



The files in this folder offer a logging facility that allows you to log application errors to any kind of storage or output device. It comes with five sample classes that each log to a different device: a database, email, event log, a file, and the console output window. You can easily extend these further to other output devices.

The Observer design pattern plays a key role. 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 4.5* because it requires that the application has write privileges to the output logging device. In particular when running a web application this can be a bit tricky. But here is how you would use this facility with the proper privileges.

First, 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 true production system you would typically have only one output device and possibly two, if you wanted 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 `InitializeLogger` method, but the default level is Error. Usually, this is all you do and then let the `<customErrors>` setting in `web.config` determine to which error page the user is redirected.

'Last resort' error handlers are a good place to log web application errors. It limits the need for try/catch blocks throughout the application which is reasonable because meaningful exception handling within a page is usually impossible. In other words, there is usually nothing you can do, other than logging the error and investigating later what happened. Anyhow, the pattern you use within an application with exception logging is something like this:

```
// C#

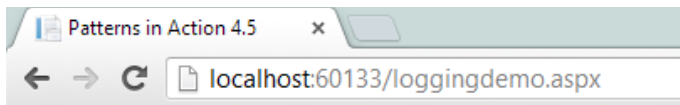
try
{
    int y = 0;
    int x = 10 / y;
}
catch (ApplicationException ex)
{
    SingletonLogger.Instance.Warning("Divide by zero in DoThis()");
    throw ex;
}
```

```
' VB

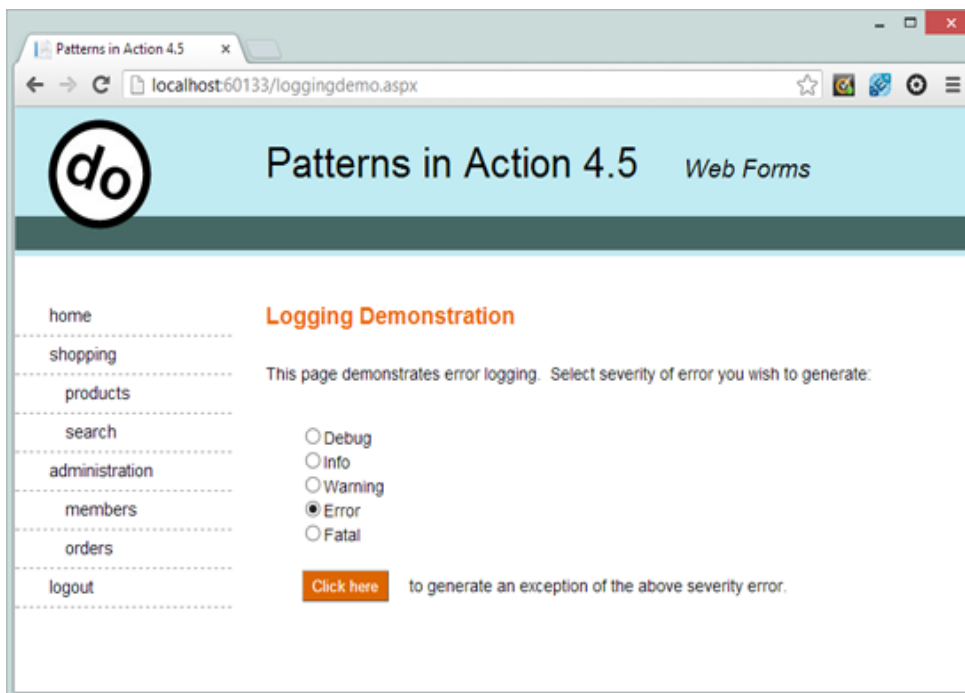
Try
    Dim y As Integer = 0
    Dim x As Integer = 10 / y
Catch ex As ApplicationException
    SingletonLogger.Instance.Warning("Divide by zero in DoThis()")
    Throw ex
End Try
```

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 Web Forms 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:



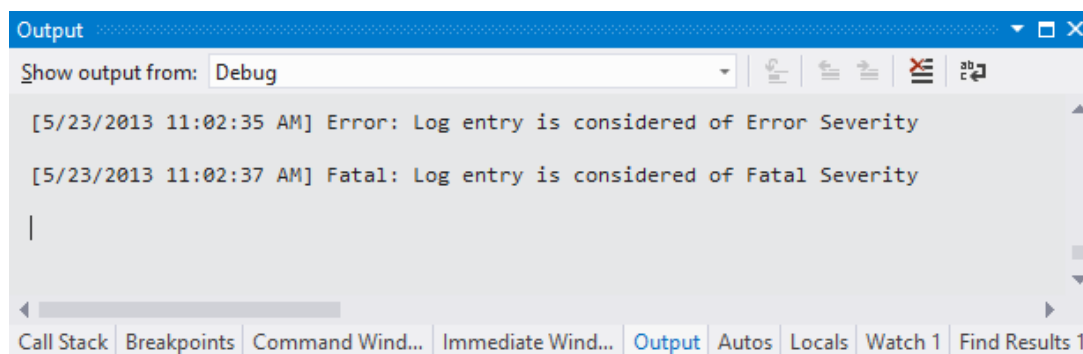
First select a radio button with the severity of the error you wish to generate. Then click the orange button. In the code behind of this page a log entry is generated of the selected severity level, that is, as a Debug entry, an Info entry, a Warning entry, an Error entry or a Fatal entry. This logging 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:

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

With the above code in place, the global configuration (which set in the first line in `InitializeLogger` in `Global.asax` file) determines which log entries actually get logged. In `InitializeLogger` you see that we have set a global '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 earlier, in the `global.asax` file we've established two active 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 `Global.asax` file (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.



Design Patterns and Practices

Patterns in Action 4.5 lets you explore how design patterns are used in a real-world e-commerce environment. The design patterns and associated best practices fall into four 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.

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 1994, 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 important today, but over time developers have found 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 business applications. These include Factory, Proxy, Singleton, Façade, and Strategy. Many well-designed, mission-critical applications make 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 GoF patterns is more applicable to specialized, niche-type applications. The Flyweight pattern is a good 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 GoF 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 fall into a category that is important but at a more granular and localized level (as opposed to 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 local 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 refer to 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 4.5* we have also adopted this usage and simply refer to the Factory pattern.

Microsoft introduced the Provider design pattern in .NET 2.0. Although not a GoF pattern (i.e. 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 4.5* reference application demonstrates where and how patterns are used in full stack applications. It includes only the most relevant and the most frequently used patterns. We could have possibly crammed all 23 GoF patterns in the application, but that would have skewed reality by not reflecting the real-world usage of design patterns.

In *Spark 4.5* we go one step further and reduce the use of patterns to patterns & practices that are considered essential to building simple, light-weight, and fast apps.

Anyhow, going back to *Patterns in Action 4.5* the table below summarizes the GoF patterns used, their location, and their usage. These patterns are referenced also in the code comments.

GoF Design Pattern	Project	Class	Usage
Facade	ActionService	Service	The service API for all UI clients
Factory	DataObjects	DaoFactories DaoFactory	Used in the manufacture of other classes. Each data access technology has its own Factory which creates provider specific data access classes.
Factory	DataObjects	Db	The DbProviderFactory is a built in factory class.
Factory Method	DataObjects	DataContextFactory	Created DataContext objects rapidly using cached values
Composite	Controls	MenuComposite MenuCompositItem	Used to represent the hierarchical tree structure of the menu.

Observer	Web Forms	ObserverLogToDatabase ObserverLogToEmail ObserverLogToEventLog ObserverLogToFile	Used to 'listen to' error events and log messages to a logging output device, such as email, event log, etc.
Provider (Microsoft Pattern)	Web Forms	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.
Decorator	Transactions	TransactionDecorator	Used to 'embrace and extend' the functionality of the built-in TransactionScope class.

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 strong slant towards Java. Fowler 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 matches the model employed in *Patterns in Action 4.5* (and also the model in *Spark 4.5* as you will see later).

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 the Enterprise patterns built-in which shields .NET developers from having to write any code that implement these. In fact, most .NET developers are not even aware that there is a common pattern or best practice underlying the feature under consideration. Here are some examples of these seemingly trivial patterns: 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 in Web Forms -- i.e. the code behind)

This being the case, Fowler's book is still 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.

Below is a summary of Enterprise patterns in *Patterns in Action 4.5*.

Enterprise Design Patterns	Project	Class	Usage
Domain Model	Business Objects	Catalog, Product, Member, Order, Order Detail	Business Objects is essentially a different name for Domain model.
Identity Field	Business Objects	Catalog, Product, Member, Order, Order Detail	The identity value is the only link between the database record and the business object.
Foreign Key Mapping	Business Objects	Product, Order, OrderDetail	Points to Category, Member and Order which are the foreign key parent business objects.
Remote Façade	Action Service	ActionService	The API is course grained because it deals with business objects rather than attribute values.
Service Layer	Action Service	ActionService	A service layer that sits on top of the Domain model (business objects)
Transaction Script	Action Service	ActionService	The Façade API is course grained and each method handles individual presentation layer requests.
Transform View	Web Forms	MenuComposite	Menu items are processed and then transformed into HTML
LazyLoad	Web Forms	ViewStateProviderService	ViewState Providers are loaded only when really necessary
Data Transfer Object	MVC	All objects in the Models folder: MemberModel, MembersModel, OrdersModel, etc.	Provides a way to transfer data between processes. These are objects that only hold data; they don't have behavior (methods).
Data Mapper	Data Objects	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 previous release of *Patterns in Action* version 4.0 was built with WCF and contained a large section on SOA (Service Oriented Architecture) and Messaging Patterns. Most SOA implementations use SOAP which is a bulky and complex XML based protocol.

SOA and SOAP are still widely used, but today there is a huge shift taking place in favor of a simpler and easier approach called REST (REpresentational State Transfer). Microsoft now also embraces REST with the addition of Web API projects to Visual Studio 2012 which is a great platform to build RESTfull apps.

In this release we removed WCF which makes the *Patterns in Action 4.5* application much more light-weight and simpler. For a REST review we refer to *Spark 4.5*.

To explore and learn about SOA and Messaging patterns we refer to the prior version of this package, which you can download for free from your download page on our website at dofactory.com. The documentation on this topic is very thorough. Please note that it run on VS 2010.

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: ASP.NET MVC being a prime example.

In its original form MVC was rather rigid which limited its usefulness in modern UI platforms. As a result, several MVC-derived patterns have evolved, two of which are included in our *Patterns in Action 4.5* 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*

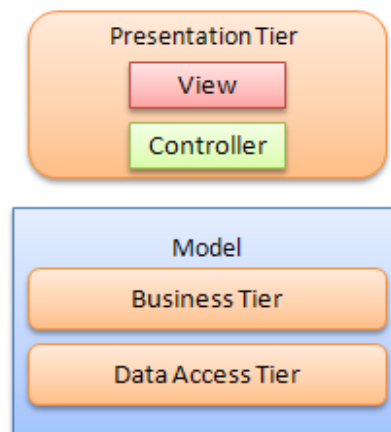
4.5 are MVC, MVP, and MVVM, one for each of the UI technologies. We'll explore each of these next.

MVC (Model-View-Controller) Design Pattern

Well-designed .NET applications are usually built with three layers: Presentation, Business, and Data. When describing the MVC application we focused on the Presentation tier. As a result the relationship between MVC and the 3-layer architecture may not be so obvious. Let's review this situation.

If you look carefully at MVC you'll notice that the Presentation layer holds just two of the three MVC components: the *Controllers* and *Views*. All layers 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.

This may all be just semantics, but what is important is that MVC is not to be confused with a three-layer architecture; these are quite different concepts. Their topological relationship is depicted in the figure below.

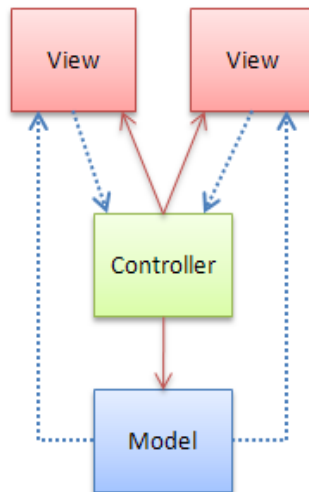


Relationship between MVC and 3-layer architecture

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 refinements 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 4.5* 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 (.cshtml or .vbhtml 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 a 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 Shop Area you find a model (ViewModel) called ProductModel. It is used to send product detail data to the view. It has a property called UnitPrice that holds the price of the product. 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 inject the value at the proper place on the page. ViewBag is another way to send data to from the Controller to the View.

Please note that these Model objects are *not* the M in MVC, but simply helper classes that facilitate data transfer from the Controller 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. By the way, the Model objects are an implementation of the DTO (Data Transfer Object) pattern; these are objects whose only purpose is to transport data and therefore they have no methods.

Microsoft has given this ASP.NET platform the moniker MVC. However they have been adding functionality that makes it less pure in the eyes of some pattern purists who argue that it breaks with the MVC pattern. In fact, these purists 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, if you think about this, 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. From what we can tell, most architects and developers are on the pragmatic side.

MVP (Model-View-Presenter) Design Pattern

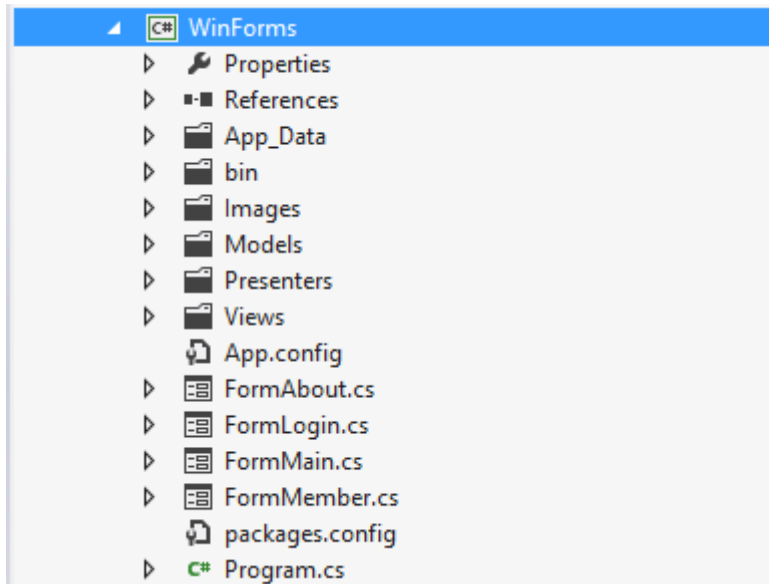
The MVP pattern is an extension of MVC. In MVP the Controller is assigned more responsibility by giving it access to the View and its user gestures. User gestures are events created by the keyboard and mouse, such as, clicking a button, entering text, dragging an icon, etc. When a Controller can respond 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? A big one 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 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 prevents 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 4.5* uses MVP and demonstrates how you can apply this pattern.

Our Windows Forms application contains 3 folders that clearly explain what part of the MVP they represent. They are Models, Presenters, and Views.



The \Models folder has local Business Model objects: they are, MemberModel, OrderModel, and OrderDetailModel. Model objects are essentially the same as business objects in the Business Layer.

The \Views folder 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 is one or more derived IViews. For example, FormMember implements IMemberView and FormLogin implements ILoginView, but FormMain implements two interfaces: IMembersView and IOOrdersView. These interfaces contain the fields that are displayed on their respective forms. For instance, ILoginView has two data members that are read only: Email and Password. This exactly matches the Login screen where we read the email and password provided by the user.

The \Presenter folder 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. MemberPresenter 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 they did, the unit tests 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

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 4.5* 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 4.5* 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 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 4.5*, the Provider consumes the services offered by the service layer.

Next, we explore the MVVM components in more detail. First of all, the Views are represented by standard WPF XAML windows. These windows reside in the root of the WPF project.

The ViewModels folder has just three classes: a *ViewModelBase*, an abstract *CommandModel* which is a thin wrapper around the built-in *RoutedUICommand*, and a *CustomerViewModel* class which 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 Models folder 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.

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 the Models\Provider\ folder 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 services layer. AutoMapper maps business 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 4.5* 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 we decided 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 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 4.5* 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.

MV Patterns Summary

Below is a summary of Model-View Design Patterns in *Patterns in Action 4.5*.

Model-View Design Patterns	Project	Classes / Projects
Model View Controller (MVC)	MVC	All Model, View, and Controller classes under the different Areas
Model View Presenter (MVP)	Windows Forms	Numerous classes organized in appropriately named folders.
Model View ViewModel (MVVM)	WPF	Numerous classes organized in appropriately named folders.

Summary

Patterns in Action 4.5 is a comprehensive solution that demonstrates when, where, and how design patterns are used in a multi-layer, e-commerce environment. We are hopeful that after exploring this resource 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.

The next step in your pattern journey is *Spark 4.5* which is a light-weight, pattern-based rapid application development (RAD) framework which allows developers to quickly create .NET apps that are simple, robust, and fast.