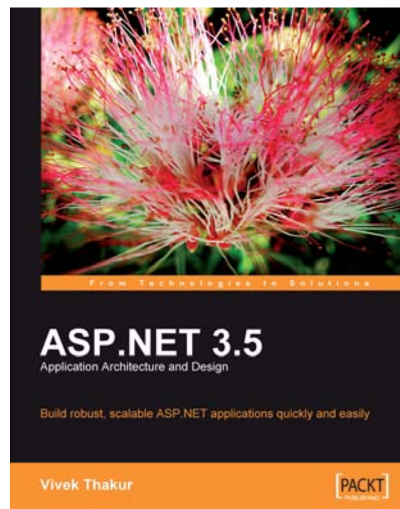
ASP.NET 3.5 Application Architecture and Design

Vivek Thakur



Chapter No. 5
"Model View Controller"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.5 "Model View Controller"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Vivek Thakur is passionate about architecting and developing applications based on the Microsoft .NET platform using ASP.NET, C#, VB.NET, and MS AJAX. He has authored several technical articles on ASP.NET and has also been an All-Star-level contributor on the ASP.NET forums. Vivek's passion for ASP.NET has been formally recognized by way of the Most Valuable Professional (MVP) award given to him by Microsoft in April 2007, and again in 2008. He is also a Subject Matter Expert for Microsoft ASP.NET 3.5 Certification Exams. He is a leading contributor and moderator in the CodeAsp.Net forums. Vivek is currently working as the Managing Partner in Axero Solutions LLC, a US-based software product development and business consulting firm.

Although his expertise lies in Microsoft's .NET platform, Vivek is also knowledgeable on J2EE and C/C++. He has a deep interest in programming, chaos theory, and artificial intelligence, and is a strong advocate of chaos theory in software systems and management.

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

Besides his love for software architecture and design, Vivek also focuses on project management skills and has substantial experience in managing small to medium sized projects. He has also conducted numerous training sessions and provided concept-based tutoring for different software firms across India.

Vivek received his Bachelors degree in engineering from the Indian Institute of Technology (IIT), New Delhi, India.

Writing this book would not have been possible without the support of my family and friends. My sincere gratitude to my mother Sharda Thakur, father G.C Thakur, and my sisters Isha and Shikha Thakur for their continued support and encouragement while I was writing this book. Special thanks to my friend Kritika Srinivasan for her incessant support throughout.

I would like to acknowledge Tim Eisenhauer for his extended support while technically reviewing the book, and also for his extended efforts in discussing and providing feedback on many topics.

I would like to thank Ian Robinson and Jerry Spohn for their relentless efforts in technical reviews and making sure that I do not miss out on core technical issues. Thanks to Ved for his detailed feedback and help in solving basic queries.

Also, many thanks to our technical editors Rakesh and Shadab, and our Production Coordinator Shantanu.

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

ASP.NET 3.5 Application Architecture and Design

The world of web development, as we see today, has undergone many dynamic changes shaped by multiple new technologies and platforms. Over the last few years Microsoft ASP.NET has quickly evolved to become one of the most famous platforms for developing web-based solutions. Since early 2002, when the first version (1.0) of ASP.NET was released, Microsoft has continuously added many out-of-the-box features and components, making web development easier for the end developer. In a very short time span, the ASP.NET platform has grown and matured into a stable object-oriented framework, with a large set of useful tools and a huge class library, attracting widespread interest in the developer communities around the world. With the introduction of LINQ, MS AJAX, WCF, WPF, and a lot of exciting new tools, the .NET framework has not only grown large but also flexible, in terms of the choices and options being offered to the developers.

With all of these new technologies hogging the limelight, an ever-increasing gap was created in the mindset of new developers, due to a shift in priorities. Developers, especially beginners, were attracted by the buzz created by these new, cool tools, and started interpreting them as a solution for better architecture and design, losing focus on the core concepts in the process. A developer, who has just learnt the basics of ASP.NET, was more eager to devote his or her time to technologies such as AJAX and LINQ instead of learning and implementing design patterns.

One reason for this paradigm shift was the lack of books that could showcase a better way to structure and develop ASP.NET-based web solutions, explaining with examples how to use different architectural styles and design patterns in real-life ASP.NET code. This book aims to bridge that gap.

I won't be focusing on deep and detailed theoretical concepts, as this book is not a "pure" architecture and design guide. Rather, the goal is to show you how to design a web site in ASP.NET the correct way, focus on different design options, analyze and study what architectural options we have, and decide when to use which architectural solution. It is very important to understand that there is no one perfect or best way in architecture and design. We need to improvise, and adapt to each project's unique requirements. Understanding core application architecture and design patterns can be tough for many developers, and so this book aims to elucidate these through the use of real-life examples and code in ASP.NET. This book will also shed some light on the basics of better application structure, coding practices, and database design, and will demonstrate, with suitable examples, how the correct architectural decisions can greatly impact overall application stability and performance.

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

What This Book Covers

Chapter 1 will introduce you to architecture and design in ASP.NET, including tiers, layers, and logical structuring.

Chapter 2 discusses the advantages and disadvantages of using the simplest and easiest 1-tier, 1-layer default architecture in ASP.NET. You will also understand when and why we should use out-of-the-box data source controls, and how the 1-tier, 1-layer style is tightly-coupled and is not flexible or scalable.

Chapter 3 discusses what an ER diagram is, the domain model, the basics of UML, and what an n-layer design is, and how it increases the flexibility and maintainability of the code when compared to a 1-layer architecture. A sample project is explained with code in a 3-layer model. The drawbacks or limitations of this model are also discussed.

Chapter 4 talks about n-tier architecture in ASP.NET and how to implement it. It also explains Data Transfer Objects and how to use them with 4-tier and 5-tier web solutions.

In *Chapter 5,* you will learn and understand what MVC design is, and how the ASP.NET MVC framework helps us quickly implement MVC design in our web applications.

In *Chapter 6,* you will learn how and when to use the most common design patterns in ASP.NET: Factory, Dependency Injection, Singleton, and others.

Chapter 7 explains why we need SOA, explaining the advantages of SOA for a beginner. A sample project using SOA architecture is discussed. The chapter also explains how the Windows Communication Framework (WCF) compliments SOA.

Chapter 8 deals with the importance of a well-designed database, balanced normalization, logical and physical models, and tips and tricks for better database models.

Chapter 9 covers localization for ASP.NET applications, the deployment of localized applications, the localization framework, and best practices.

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

5
Model View Controller

These days, **Model View Controller** (**MVC**) is a buzzword in the ASP.NET community, thanks to the upcoming ASP.NET MVC framework that Microsoft is expected to launch soon (at the time of writing of this book, only Preview 5 was available). This chapter is dedicated to MVC design and the ASP.NET MVC framework.

In this chapter, we will learn about MVC design patterns, and how Microsoft has made our lives easier by creating the ASP.NET MVC framework for easier adoption of MVC patterns in our web applications. The following are some highlights of this chapter:

- Understanding the Page Controller pattern
- Understanding the need for the MVC design pattern
- Learning the basics of MVC design
- Understanding the Front Controller design pattern
- Understanding REST architecture
- Understanding the ASP.NET MVC framework
- Implementing the ASP.NET MVC framework in a sample application

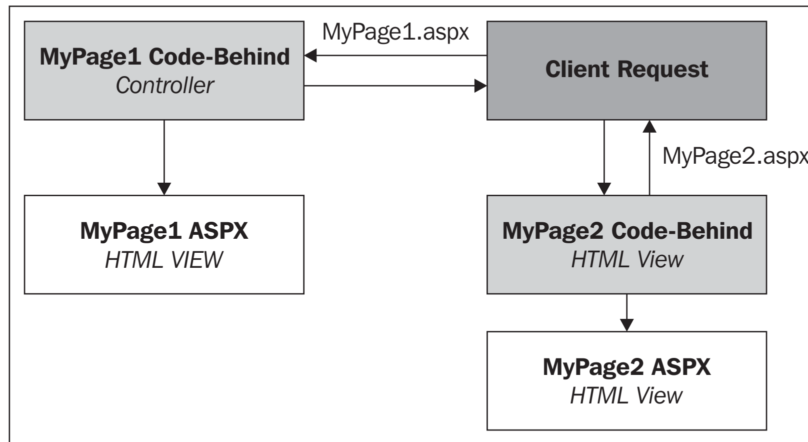
Page Controller Pattern in ASP.NET

So far, all web pages we have created in our coding samples are based on the page controller pattern, which is the default architecture in the ASP.NET web forms. Let us understand page controller in detail.

In Chapter 2, we noticed that inline coding samples in ASP and ASP.NET had HTML and code scripts mixed together, creating a hard-to-maintain code base. Then we studied how code-behind classes "modularized" the architecture by separating the logic from the HTML. This code-behind architecture is a page controller based design, where by controller we mean the components that control the rendering of the HTML, which in the case of ASP.NET web forms are the code-behind classes.

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

Each page has a code-behind class, and the URL requested by the client is directly handled by individual pages. Any button or server control causing postbacks (such as a DropDownList control) is handled directly by the page code-behind class.
So understanding the page life cycle is very important in a page controller based architecture. Here is a diagram that shows how a page controller pattern works in ASP.NET:



So for every page, its code-behind will act as a controller and handle all requests, and return processed HTML to the client browser.

Problems with Page Controller Design

In the page controller design we have a controller for each distinct page in our application (a separate code-behind class having all of the logic that fires sequentially as each page loads according to the ASP.NET page life cycle). So for big projects, there could potentially be a lot of code in the code-behind files, creating problems in code maintenance and support.

GUI Unit Testing

Separating business logic and data access code from the GUI is one of the steps leading towards a better design. In the previous chapters, we saw how to implement a basic n-tier architecture using tiers and layers to achieve loose coupling. But testing the application, especially the GUI and the code-behind classes in a page controller based model, is very difficult because the only way to test something like a button click's code-behind event handler is to click the button itself! This means that if we put more and more code in code-behind classes (which inevitably becomes the case in large web applications with lots of UI controls),

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

we will not be able to run unit tests on the UI code. So the only way to test the application would be to manually test the GUI. The page controller based design does not support unit testing, and we would not be able to use automated unit testing tools such as NUnit, MBUnit and so on (which we can easily use to test the other layers such as BL and DAL).

There are ways to perform automated testing in GUI using testing tools that use Javascript to actually perform the button click events through the code, although they are clumsy and difficult to use. Even if we write scripts today, they will need to change if the GUI changes in the future, which is very much possible as the GUI can be changed many times during a project's lifetime and also after it is finished. This makes unit testing more difficult as one would need to rewrite the automated testing scripts on every GUI change. So most people tend to use brute force testing, which involves clicking all possible UI controls (such as buttons and so on) and verifying whether the code works as expected. This is a very time-consuming task, and if the GUI changes, the testing needs to be carried out again.

We will now see how MVC design helps us implement a clean separation between the UI and the controller, and also make our UI unit-testable.

MVC Design: A Front Controller based Approach

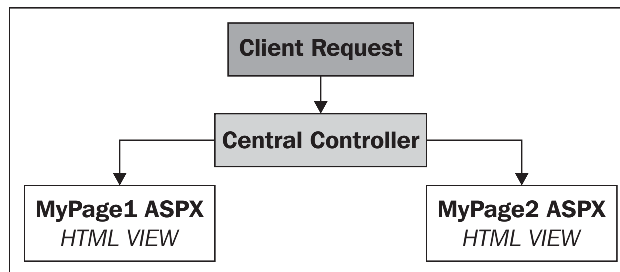
MVC, which stands for Model View Controller, is a design pattern that helps us achieve the decoupling of data access and business logic from the presentation code , and also gives us the opportunity to unit test the GUI effectively and neatly, without worrying about GUI changes at all. In this section, we will first study the basic MVC pattern and then move on to understanding the ASP.NET MVC framework.

A framework is a set of tools that includes libraries or methods developed according to a certain architecture, so that applications do not need to re-invent the wheel. Instead of re-writing the basic implementation each time, they can use the framework and abstract themselves from the internal framework implementation details.

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

Front Controller Design

MVC is based on a front controller design, where we have a centralized controller instead of multiple controllers, as was the case in the page controller based design that we saw earlier. By default, ASP.NET is page controller based. So making a front controller based project would require a lot of work (using HttpHandlers to route the requests manually). Basically, in a front controller design, we trap all of the client requests and direct them to a central controller, and the controller then decides which view to render (or which ASPX page to process). Here is how a basic model of a front controller design works:



As you can see, the front controller sits at the "front" of all of the pages and renders a view based on logic in the central controller file. In the next section we will study and analyze exactly what goes on inside a controller, a view, and a model.

Basics of MVC

Let's first get into the theoretical aspects of MVC. MVC design has three major parts:

- **Model:** This refers to the data that is shown in the UI. This data can come from different sources, for example, a database.
- **View:** This refers to the user interface (UI) components that will show the model data.
- **Controller:** This controls when to change the view, based on user actions, such as button clicks.

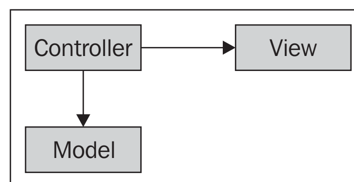
For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

In terms of ASP.NET web applications, the model, view, and controller participants can be identified as:

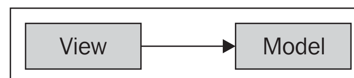
- **View:** This refers to HTML markup in ASPX pages, minus the code-behind logic. This view is rendered in the presentation tier (the browser).
- **Controller:** This refers to the special controller classes that decide which model needs to be shown to which particular view.
- **Model:** This refers to the data coming from the data layer, which may be processed by the business layer.

Before moving ahead, an important point to understand is that the MVC design is not a replacement to the n-tier architecture. MVC is more focused on how to keep the UI separate from the logic and the model; the model itself can be broken into separate tiers.

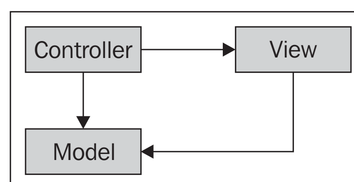
In the MVC design, the model, the view, and the controller are not related directly to the layers, or to the physical tiers; they are logical components that operate together in a certain pattern. The controller is related directly to the model and the view. Based on user actions (in the view), it fetches the data (the model) and populates the view. The relationship between the controller, the model, and the view can be depicted as:



The view is based on the model, which means that its job is to simply render the model that the controller passes to it:



So the net relationship between the three components can be described as:



For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

A few important points to note from the above diagram:

- We can see that the model depends neither on the view nor on the controller, which is logical. Think of it like this: we have some data in the database tables; we use DAL code to handle this data and BL code to operate on this data as per certain business rules. Now, it is up to the UI to present and show this data. But the data itself is not dependent on the graphical user interface (GUI). So the model is independent of the view and the controller.
- The view does not depend on the controller; rather, the controller is associated with the view. That means we have a separation between the view and the controller, allowing us to change views independent of the controller.
- The view depends on the model, and is updated when the model's state has changed. As the view cannot contain any logic (which is stored inside the model), the view depends on the model; that is, the model is in charge of updating the contents or displaying the view.

Now we will look at the practical aspects of implementing this MVC design using the ASP.NET MVC framework, which will help us implement our web applications. MVC will be ready in no time. But before going ahead with the actual code, we need to understand another important aspect of ASP.NET MVC framework, that is, REST!

REST: Representation State Transfer

REST means Representational State Transfer, an architectural pattern used to identify and fetch resources from networked systems such as the World Wide Web (WWW). The REST architecture was the foundation of World Wide Web. But the term itself came into being around the year 2000, and is quite a buzzword these days. The core principle of REST is to facilitate the sharing of resources via unique identifiers, just as we use Uniform Resource Identifiers (URIs) while accessing resources on the Web. In simple terms, REST specifies how resources should be addressed, including URI formats, and protocols such as HTTP. The term resources include files such as ASPX pages, HTML files, images, videos, and so on.

In the default page controller based design in ASP.NET, we don't follow a strict REST-based architecture. If we use a pure REST-based architecture, then all of the information required to access a particular resource would be in the URI. This means that we don't need to check if a postback happened or not, because each request is unique in itself and will be treated differently (via unique URLs). Whereas in ASP. NET, we can use the postback technique to make the same requests using the same URLs and do different processing based on whether it is a postback or not. Many a times, in numerous projects, we come across the following coding style in ASP.NET code-behind files:

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

```
btnSave_Cick()
{
    Response.Redirect("~/MyPage.aspx");
}
```

Here, on the postback (button click), we are redirecting the user to another resource (`mypage.aspx`), and this approach goes against the REST principle as we are delegating the responsibility to load a resource to another page based controller's postback event. This is not REST-like behavior. Now, we will see how MVC compliments the REST approach.

MVC and REST

MVC is radically different from the default page controller based design in the ASP.NET framework as it implements a front controller based design. In our normal applications, we use a lot of postbacks and make use of ViewState, and the development is centered around web forms. For each functional aspect, we may have a single webform; for example, for adding customers, we might create something like `AddCustomer.aspx`, and for showing a list of customers, we might use `CustomerList.aspx`.

But in an MVC architecture, webforms lose their importance. We don't create webforms in the same way that we do in standard ASP.NET applications. In the MVC framework, we use URL routing, which means that all URLs have some specific format, and the URLs are used based on the settings in a `config` file. In a standard ASP.NET application, the URL is linked to a specific ASPX file, say `http://localhost/CustomerList.aspx`. In MVC, the URL routes are defined in a REST-like fashion: `http://localhost/customer/list/`.

So in MVC, ASPX pages are reduced to simply showing the view; they will not have any code in their code-behind classes. What needs to be shown on an ASPX page will be handled by the Controller classes. ASPX will just be a kind of view engine and nothing else. ASPX will not have control-level event handlers or any kind of logic in the code-behind. In the next section, we will see how the ASP.NET MVC framework makes our life easier in adopting an MVC based approach in our projects.

ASP.NET MVC Framework

The ASP.NET MVC framework was released by Microsoft as an alternative approach to web forms when creating ASP.NET based web applications. The ASP.NET MVC framework is not a replacement or upgrade of web forms, but merely another way of programming your web applications so that we can get the benefits of an MVC design with much less effort.

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

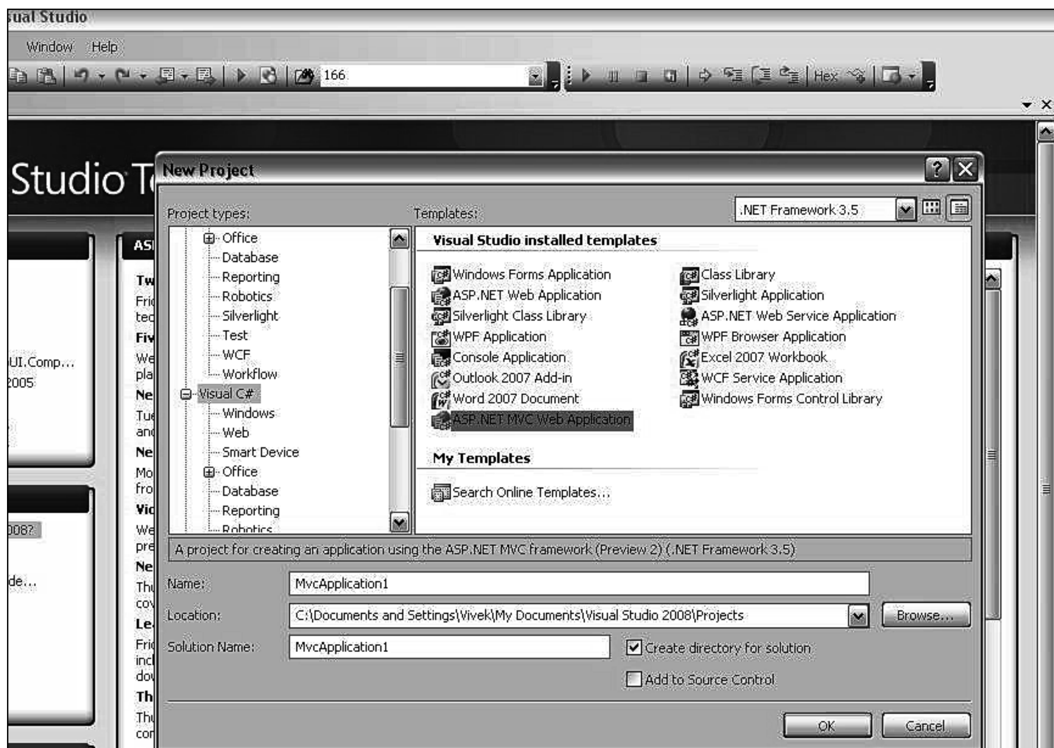
As of now, the ASP.NET MVC framework is still in CTP (Community Technology Preview, which is similar to an advanced pre-stage), and there is no certain date when it will be released. But even with the CTP 5, we can see how it will help MVC applications follow a stricter architecture.

We will quickly see how to use the ASP.NET MVC framework through a small example.

Sample Project

First, download the ASP.NET MVC framework from the Microsoft website and install it. This installation will create an MVC project template in VS 2008.

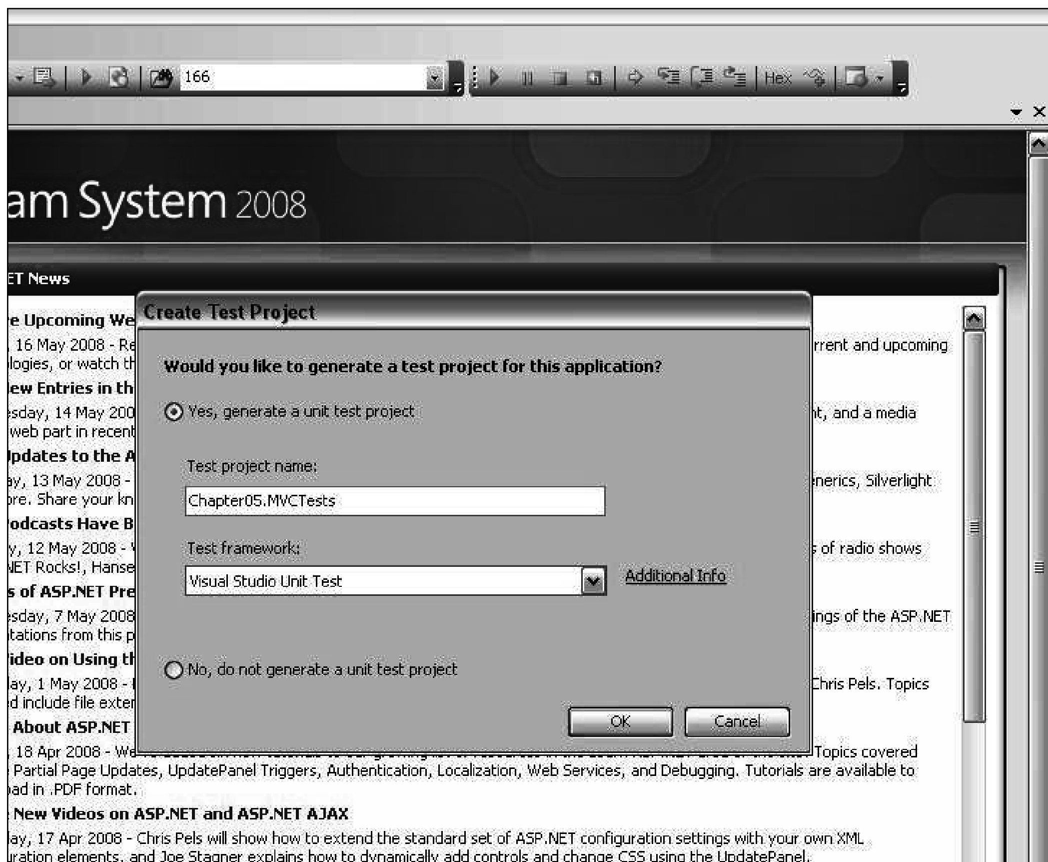
Start VS 2008, select the **File | New Project** menu item and then choose the **ASP.NET MVC Web Application** template to create a new web application using this template.



For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

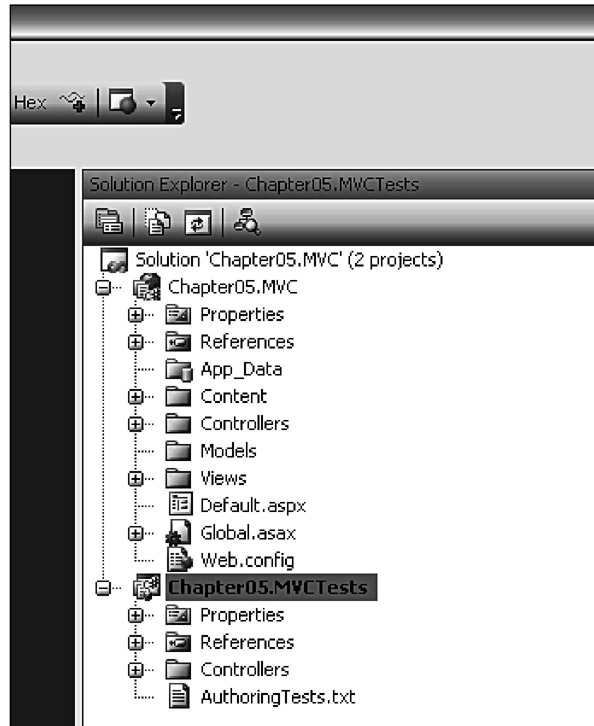
By default, when you create a new application using this option, Visual Studio will ask you if you want to create a Unit Test project for your solution, with a drop-down list pre-populated with the different possible types of test frameworks (the default would be VS test, but in case you installed MBUnit or NUnit, these would also be populated here):

> There are many free unit testing frameworks available for ASP.NET projects, and NUnit and MBUnit are two of the most popular ones. Here are the links:
>
> MBUnit: `http://www.mbunit.com/`
>
> NUnit: `http://www.nunit.org/index.php`



For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

Select the default option and click **OK**. You will notice that two projects have been added to the solution that VS has created. The first project is a web project where you'll implement your application. The second is a testing project that you can use to write unit tests against.



In our custom MVC code project, we had different projects (class libraries) for the model, the view, and the controllers.

The default directory structure of an ASP.NET MVC Application has three top-level directories:

- `/Controllers`
- `/Models`
- `/Views`

When the project becomes large, it is recommended that the Model, Views and Controllers are put in separate class library projects of their own so that it's easy to maintain them. But for the purpose of illustrating the ASP.NET MVC framework, this default structure is fine for us.

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

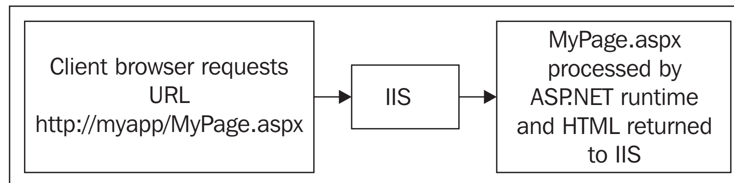
We will create a simple customer management application. For this, we first create some ASPX pages in the `Views` folder. Note that VS has already created these subfolders for us, under `Views`:

- **Home:** Contains the and Index views
- **Shared:** Contains shared views such as master pages

Before we go on to adding custom code in this project, let us understand what VS has done for us while creating this MVC project.

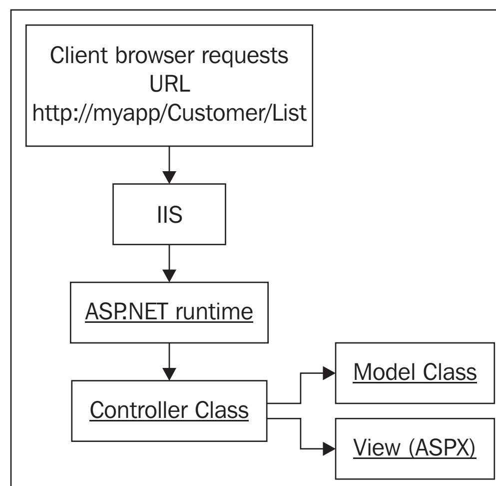
URL Routing Engine

In the standard ASP.NET model (or Postback model), the URLs map directly to the physical files:



So when we make a request to a page, say `MyPage.aspx`, the runtime compiles that page and returns the generated HTML back to IIS to be displayed by the client browser. So we have a one-to-one relationship between the application URLs and the page.

But in the MVC framework, the URLs map to the controller classes.



For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

Therefore, the URL is sent to IIS and then to ASP.NET runtime, where it initiates a controller class based on the URL, using the URL routes, and the controller class then loads the data from the model, with this data finally being rendered in the view.

The controller classes uses URL routing to map the URLs, which in simpler terms means rewriting URL. We can set up the rules for which URL is to be routed to which controller class. The routing will pick up the appropriate controller and pass in the query string variables as necessary.

Open the `global.asax.cs` file and examine the following code:

```
public class GlobalApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.MapRoute(
                "Default",
// Route name
                "{controller}/{action}/{id}",
// URL with parameters
                new { controller = "Home", action = "Index", id = ""
}// Parameter defaults
            );
        }
        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
```

The `RegisterRoutes()` method contains the URL mapping routes. Initially we have only the default rule set:

```
            routes.MapRoute(
                "Default",
// Route name
                "{controller}/{action}/{id}",
// URL with parameters
                new { controller = "Home", action = "Index", id = "" }
// Parameter defaults
            );
```

This URL mapping engine comes from `System.Web.Routing.dll`, which can be used independently, without the ASP.NET MVC framework, to rewrite URLs in your standard ASP.NET web applications.

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

The `MapRoute()` method, which handles URL routing and mapping, takes three arguments:

- Name of the route (string)
- URL format (string)
- Default settings (object type)

In our case, we named the first route "Default" (which is the route name) and then set the URL as:

```
Controller/action/id
```

The `Controller` here is the name of the controller class. `action` will be the method that needs to be invoked inside that controller class. `id` would be the parameters that need to be passed, if any.

In the default arguments, we create a new object and call it "Home", set the action to Index, and do not pass parameters to it. Note the new anonymous type syntax used to create parameter defaults:

```
new { controller = "Home", action = "Index", id = "" }
```

The `var` keyword and anonymous types: We normally use classes to wrap behavior and properties, but in C# 3.0, we can create the types anonymously without needing to create classes for them. This can be useful when we need to create light weight classes that have only read-only properties. We can use the anonymous syntax to create those types without the need to create a class for them. We can use the new "`var`" keyword to hold such anonymous types, for example:

```
var ch = new { readOnlyProperty1 = value1,
readOnlyProperty2 = value2 };
```

It is important that we name and assign a value to each of the properties that we are creating. What will be the type of the properties? They will automatically be cast to the data types of the values of the properties specified. The anonymous types will always be derived from the base object class directly. They can only be used within class members and cannot be passed as method arguments (unless they are boxed), return values, or be specified as class-level variables. Once the type is created, it cannot be changed into another type.

So we create a new anonymous type as the last argument of the `MapRoute()` method, passing in variable defaults with three properties, namely controller, action, and parameter.

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

Now have the `Default.aspx` page under the root directory, which acts as a redirecting page to the main home page of the site (which is `/View/Home/Index.aspx`).

We cannot directly set that as the "default" page since we are using URL routes to process pages instead of using physical files in the URLs.

So in the code-behind of our `Default.aspx` page, we have a simple redirect:

```
public void Page_Load(object sender, System.EventArgs e)
    {
        Response.Redirect("~/Home");
    }
```

So the runtime will first set up routes in the `global.asax` page, then it will process the `Default.aspx` page. Here it faces a redirect to this URL: `/Home`.

The Controller

The MVC framework maps this URL to the route set in the global route table, which currently has only the default one, in this format:

```
Controller/action/id
```

So `/Home` corresponds to a controller named Home, and because we have not specified any action or ID, it takes the default values we specified in the `RegisterRoutes()` method in the `globals.asax.cs`. So the default action was Index and the default parameter was an empty string. The runtime initializes the `HomeController.cs` class, and fires the Index action there:

```
public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewData["Title"] = "Home Page";
            ViewData["Message"] = "Welcome to ASP.NET MVC!";
            return View();
        }}
```

In this `Index()` method, we set the data to be displayed in the View (aspx/ascx pages) by using a dictionary property of the base Controller class named `ViewData`. `ViewData`, as the name suggests, is used to set view-specific data in a dictionary object that can hold multiple name/value pairs. When we call the `View()` method, the `ViewData` is passed by the Controller to the View and rendered there.

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

The View

Let us now look at the View. How does the framework know which View or aspx page to call? Remember that we passed the value "Index" in the action parameter (in the default route in the `global.asax.cs` file), so the `Index.aspx` will get called. Here is the code-behind of `Index.apsx`:

```
public partial class Index : ViewPage
    {
    }
```

There is absolutely no code here, which is a very important characteristic of the MVC design. The GUI should have no logical or data fetching code. Note that the `Index` class is derived from the `ViewPage` class. Using this `ViewPage` class, we can access all of the items in the `ViewData` dictionary that were set in the controller's `Index()` method and passed on to the View. Here is how we are accessing the `ViewData` in HTML:

```
<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent"
                 runat="server">
    <h2><%= Html.Encode(ViewData["Message"]) %></h2>
    <p>
        To learn more about ASP.NET MVC visit <a href="http://asp.net/
mvc" title="ASP.NET MVC Website">http://asp.net/mvc</a>.
    </p>
</asp:Content>
```

We can directly access the `ViewData` dictionary in HTML. Now that we have seen how MVC works, we will create a new page to learn how to show data using a custom DAL and strongly typed objects, instead of the `ViewData` dictionary. Our example page will show a list of all the customers.

The Model

We will use the 5-Tier solution we created in the previous chapter and change the GUI layer to make it follow the MVC design using the ASP.NET MVC framework. Open the solution we created in the previous chapter and delete the ASP.NET web project from it. The solution will then only contain `5Tier.BL`, `5Tier.DAL` and `5Tier.Common` projects.

Right click the solution in VS, and select **Add New Project**, and then select **ASP.NET MVC Web Application** from the dialog box. Name this new web project as `Chapter05.MVC`. This web project will be the new MVC based UI tier of our OMS application in this chapter.

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

The `Customer.cs` and `CustomerCollection.cs` class files in the business tier (5Tier. Business class library) will be the Model in our MVC application. To show a list of customers, the `CustomerCollection` class simply calls the `FindCustomer()` method in `CustomerDAL.cs`. We have already seen these classes in action in the previous chapter. So we can use an n-tier architecture in an MVC application, hence this shows that MVC and n-tier are not mutually exclusive options while considering the application architecture of your web application. Both actually compliment each other.

We can also create a utility class named `CustomerViewData` to transfer the Model objects to the View. There are multiple ways to pass- in the Model to the View through the Controller, and creating `ViewData` classes is one of them. Here is the `CustomerViewData` class created in the `CustomerComtroller.cs` file in the `Chapter05.MVC` web project:

```
#region ViewData
        /// <summary>
        /// Class used for transferring data to the View
        /// </summary>
        public class CustomerViewData
        {
            public CustomerViewData() { }

            public CustomerViewData(Collection<Customer> customers)
            {
                this.customers = customers;
            }

            public Collection<Customer> customers;

            public Customer customer;
        }
        #endregion
```

Notice that this `ViewData` class is simply wrapping the business object inside it so that we can use this class in the UI layer instead of directly passing and manipulating domain objects.

Wiring Controller, Model, and View

We will now create routes in the `global.asax` file under the existing home page route as follows:

```
routes.MapRoute(
"Customer", "Customer/{action}/{id}", new {
 controller = "Customer", action = "Show", id="" } );
```


For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

This new route will simply fire the Show action in the customer controller.

Now we create the controller class, `CustomerController`, as:

```
using NTier.BL;
using NTier.Common;

namespace Chapter05.MVC.Controllers
{
    public class CustomerController:Controller
    {
        #region ViewData
        /// <summary>
        /// Class used for transferring data to the View
        /// </summary>
        public class CustomerViewData
        {
            public CustomerViewData() { }

            public CustomerViewData(Collection<Customer> customers)
            {
                this.customers = customers;
            }

            public Collection<Customer> customers;

            public Customer customer;

        }
        #endregion

        public ActionResult Show()
        {
            CustomerViewData customerViewData = new CustomerViewData();
            CustomerCollection customers = new CustomerCollection();
            customerViewData.customers
            = customers.FindAll(LoadStatus.Loaded);

            return View("Show", customerViewData);
        }
    }//end class
}//end namespace
```


For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

In this class, we first create a subclass called `CustomerViewData`, which is a wrapper to hold the `Customer` and `CustomerCollection` business objects. We will transfer this class object to the actual view through the controller as:

```
public ActionResult Show()
        {
            CustomerViewData customerViewData = new CustomerViewData();
            CustomerCollection customers = new CustomerCollection();
            customerViewData.customers
            = customers.FindAll(LoadStatus.Loaded);

            return View("Show", customerViewData);
        }
```

In this controller action, we simply get the list of customers from the database and pass it to the `Show.aspx` page using the `View()` method.

To create the `aspx` pages, we will create a folder named `Customers` under the `View`, and add an `aspx` file named `Show.aspx` that will display a list of all the customers.

```
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent"
runat="server">
    <asp:Repeater ID="rptCustomer" runat="server">
        <ItemTemplate>
            <%# Eval("Name")%>
        </ItemTemplate>
    </asp:Repeater>
</asp:Content>
```

In the code-behind, we simply bind the data as:

```
using System.Web.Mvc;
namespace Chapter05.MVC.Views.Customer
    {
        public partial class Show :
         ViewPage <Controllers.CustomerController.CustomerViewData>
        {
            protected void Page_Load(object sender, EventArgs e)
            {
            rptCustomer.DataSource = ViewData.Model.customers;
            rptCustomer.DataBind();
            }
        }
    }
```


For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

So the code-behind is kept very light and has no extra code besides data binding to the repeater. Note that usually in the standard postback model, we bind the data under the `if(!IsPostBack)` condition so that data binding happens only once, on the page load, and not on the postback.

Here we cannot follow the same pattern as there is no concept of a postback in MVC. Each request will be as unique as the RESTful URL.

If we want, we can also bind the data in the ASPX using inline code without using code-behind, as shown here:

```
<h2>Customer list</h2>
    <%foreach (var c in ViewData.Model.customers) { %>
        <div>
        <%=c.Name %>
        <br />
        </div>
    <%} %>
```

Note the use of the "var" type to get the `CustomerData` object. This helps us in avoiding explicit casting to get the `Customer` object.

In the same way, we can edit and add objects. An important point to note is that we don't need to use the standard ASP.NET button controls any more, because we don't want the page to postback to itself. Instead, when we add a customer, we can use something like this:

```
<form method="post" action="Customer/Add">
    <input type="text" name="customerName" value="" />
    <input type="submit" name="Add" value="Add" />
</form>
```

Notice that there is no `runat="server"` tag here in these controls as they are not server controls but simple HTML controls. On clicking the **Add** button, the page will post with the action `Customer/Add`. This means that, in the `CustomerController`, it will fire the `Add` method as shown in the sample code (just for demonstration purposes):

```
public class CustomerController : Controller
    {
    public ActionResult Add(string customerName)
        {
        //create a business object and fire the add method
        Customer customer = new Customer();
        customer.Name = customerName;
        CustomerCollection customers = new CustomerCollection();
        customers.Add(customer);
        return View("Home");
        }
    }
```


For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

ASP.NET MVC will automatically set the parameter, `customerName`, with the value of the textbox from the form's post data, and will pass this value in the `Add` method of the controller. So we did not have to create the entire page object again, as the page did not postback to the same form. Hence, we avoided recreating the page class on every request or postback.

Because this chapter is more about understanding the MVC framework from the architecture's perspective, we will not go into the details of the edit and add actions; it is best to refer to the following post for these details:

`http://weblogs.asp.net/scottgu/archive/2007/12/09/asp-net-mvc-framework-part-4-handling-form-edit-and-post-scenarios.aspx`

Unit Testing and ASP.NET MVC

Unit testing is the process where the developer himself tests the code that he has written. The word "unit" here refers to modules of code that he writes, such as methods, functions, and so on. The developer would test such "units" code to verify whether everything is working as expected. This will make sure that at least the individual methods of a class work without errors. Unit testing is different from the function or integration testing that a QA (quality assurance) person performs on a working model after the development phase is over. Unit testing is more closely related to the actual code testing and ensures that most of the bugs are taken care of before the project reaches the actual testing stage. Because unit testing checks the actual methods in the code, it can easily be automated by creating mock test cases in the code using one of the many available unit testing frameworks, such as NUnit and MBUnit.

Earlier in this chapter, we stressed the need for unit testing the GUI of our ASP.NET projects, and how difficult it is to do so under the standard page controller model. But now with ASP.NET MVC, we have "thin" code-behind classes, with almost little or no code. There are almost no Session and ViewState related issues, as all URLs are handled directly by the central controller. There are no button clicks, no code-behind event handlers, and so on. So it is easy to set up unit test cases for the controller classes because then it is the same as testing any normal C# class methods, such as the DAL, the BL, and so on.

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

Summary

ASP.NET MVC is a very good platform for creating unit-testable applications that are maintainable in the long run, as well as for achieving a clearer separation between the UI and the UI-handling logic. In the previous chapter, we learnt the n-tier architecture, but the GUI in such an architecture was still not easily testable as we had a lot of embedded UI code in code-behind classes that was dependent on ViewState and postbacks. But in MVC, the concept of ViewState and postback does not exist as each request is unique in and of itself. This also presents some technical challenges if you have a complex UI, such as a GridView with inline edit or update functionality with a lot of AJAX functionality, where MVC may not work as expected (though the upcoming ASP.NET MVC framework releases will offer more flexible solutions to handle such cases).

Also, server controls such as `DropDownList` may not work as expected in the ASP. NET MVC framework, since there would be no control-level events in the code-behind and the control won't be able to postback. The core principle of the ASP.NET MVC framework is that the URL should talk directly to the requested resource in the web application. This is also the core principle of REST. So if we are using control-level events such as the `selected_change()` method of a `DropDownlist` in code-behind to process some logic, we will be breaking this very REST/MVC principle. The reason is that during such an event, the form will postback without any change in the URL, hence the concept of REST will not hold true (remember that a resource on the web can be accessed by a unique RESTful URL). ASP.NET MVC also has an out-of-the-box powerful URL rewriting capability so that we can have SEO friendly URLs in our application.

This implies that the entire page lifecycle will have no value in the MVC framework. Using MVC means going back to using standard HTML controls instead of rich server controls (even though Microsoft is coming up with a separate set of MVC server controls which will help us in adopting MVC in our applications easily).

The ASP.NET MVC framework was still evolving during the time of writing of this book. There can be a lot of upcoming improvements in the framework in terms of added functionalities and utilities. These changes will cause the final ASP.NET MVC framework release to be somewhat different in terms of syntax shown in this chapter. But the core principle of MVC will remain the same and it will be quite easy to use the latest framework release once we understand the basic principles of the MVC design.

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

Using MVC in real world commercial applications needs more skill than using the standard ASP.NET postback model. But it gives us the unique benefit of unit testing the GUI layer along with the BL and the DAL, which really helps in Test Driven Development (TDD). So, the ASP.NET MVC framework is not a silver bullet, and you should carefully consider when and in which projects to use it. There will also be a learning curve associated with it for developers coming from a webforms background. It is not a replacement for webforms, but now developers have a choice of using either the MVC or the standard webforms page-centric postback model. ASP.NET MVC can easily be used with an n-tier architecture, as we saw in our code sample.

So ASP.NET MVC is a very good choice for creating a unit-testable and search engine friendly web application which makes our web UI much cleaner by having a clear separation between the UI and code logic.

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book

Where to buy this book

You can buy ASP.NET 3.5 Application Architecture and Design from the Packt Publishing website:
`http://www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book