A Developer.com Site

http://www.devx.com/dotnet/Article/33695

Design Patterns for ASP.NET Developers, Part 1: Basic Patterns

By Alex Homer
Feb 8, 2007

For the past year or so, I've been involved in documenting frameworks that help developers write better code, and create applications that are more efficient and easier to test, debug, maintain, and extend. During that time, it has been interesting to see the continuing development of best-practice techniques and tools at one of the leading software companies in our industry. Most of the work was outside my usual sphere of ASP.NET and web development, concentrating mainly on Windows Forms applications built using .NET 2.0. This is an area where standard design patterns that have evolved over many years are increasingly being refined and put into practice.

However, I regularly found myself wondering just how many of these patterns are equally applicable and advantageous within ASP.NET applications, where we now have the ability to write "real code" in .NET languages such as Visual Basic .NET and C# -- rather than the awkward mix of script and COM components upon which classic ASP depended. Surely, out of the 250+ patterns listed on sites such as the PatternShare Community, some must be useful in ASP.NET applications. Yet a search of the web revealed that -- while there is plenty of material out there on design patterns in general, and their use in executable and Windows Forms applications -- there is little that concentrates directly on the use of standard design patterns within ASP.NET.

One very useful document is "Enterprise Solution Patterns Using Microsoft .NET," which discusses what design patterns are, how they are documented, and their use in .NET Enterprise applications. The article does not aim solely at ASP.NET, but has plenty of ASP.NET coverage.

About Design Patterns

Because there is so much general material available on the aims and the documentation of design patterns for executable applications, I will restrict this discussion to a few basic points before moving on to look at the patterns I find most useful in ASP.NET. The References sidebar that accompanies each article in this series contains links to many useful resources and web sites related to design patterns in general.

While many people (myself included) find the term "design patterns" just a little scary -- due not least to the way that they are usually described and documented -- most developers use informal patterns every day when writing code. Constructs such as `try...catch`, `using`, and `switch` (`Select Case`) statements all follow standard patterns that developers have learned over time. In fact, patterns can be:

- Informal Design Patterns -- such as the use of standard code constructs, best practice, well structured code, common sense, the accepted approach, and evolution over time
- Formal Design Patterns -- documented with sections such as "Context", "Problem", "Solution", and a UML diagram

Formal patterns usually have specific aims and solve specific issues, whereas informal patterns tend to provide guidance that is more general. As Brad Appleton, author of the book "Software Configuration Management Patterns: Effective Teamwork, Practical Integration", describes design patterns, pattern languages, and the need for them in software engineering and development:

> "Fundamental to any science or engineering discipline is a common vocabulary for expressing its concepts, and a language for relating them together."
>
> "... a body of literature to help software developers resolve recurring problems encountered throughout all of software development."
>
> "... a shared language for communicating insight and experience about these problems and their solutions."
>
> "A pattern is a named nugget of insight that conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns."

This article and the following two articles in this series demonstrate how you can use some of the common formal patterns (some with minor adaptations to suit ASP.NET requirements) in your web applications to achieve the aims set out so succinctly by the Hillside Group. You can download the example application you will see in the following sections and open it in Visual Studio 2005, or install it to run under Internet Information Services (IIS). The download file includes a `~readme.txt` file that describes the setup requirements.

Basic Design Patterns and Groups
Design patterns fall into several groups based on the type and aims of the pattern. For example, some patterns provide presentation logic for displaying specific views that make up the user interface. Others control the way that the application behaves as the user interacts with it. There are also groups of patterns that specify techniques for persisting data, define best practices for data access, and indicate optimum approaches for creating instances of objects that the application uses. The following list shows some of the most common design patterns within these groups:

  - Presentation Logic
  - Model-View-Controller (MVC)
  - Model-View-Presenter (MVP)
  - Use Case Controller
  - Host or Behavioral
  - Command
  - Publish-Subscribe / Observer
  - Plug-in / Module / Intercepting Filter
  - Structural
  - Service Agent / Proxy / Broker
  - Provider / Adapter

- Creational
- Factory / Builder / Injection
- Singleton
- Persistence
- Repository

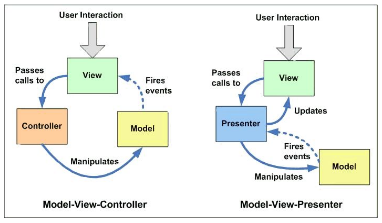


Figure 1. The Model-View-Controller and Model-View-Presenter Patterns: These patterns separate the data held by the Model from the View, linking them via a Controller or Presenter.

The Model-View-Controller and Model-View-Presenter Patterns
The Model-View-Controller (MVC) and Model-View-Presenter (MVP) patterns improve code reusability by separating the three components required to generate and manage a specific user interface (such as a single web page). The Model contains the data that the View (the web page) will display and allow the user to manipulate. The Controller or Presenter links the Model and the View, and manages all interaction and processing of the data in the Model (see Figure 1).

In the MVC pattern, user interaction with the View raises events in the Controller, which updates the Model. The Model then raises events to update the View. However, this introduces a dependency between the Model and the View. To avoid this, the MVP pattern uses a Presenter that both updates the Model and receives update events from it, using these updates to update the View. The MVP pattern improves testability, as all the logic and processing occurs within the Presenter, but it does add some complexity to the implementation because updates must pass from the Presenter to the View.

The Provider and Adapter Patterns

The Provider and Adapter patterns allow otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the other. In more practical terms, these patterns provide separation between components that allows behavioral changes to occur without prior knowledge of requirements. The application and any data sources it uses, outputs it generates, or classes it must interact with, can be created independently yet still work together (see Figure 2).

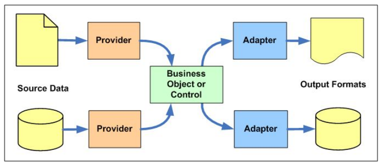


Figure 2. The Provider and Adapter Patterns: These patterns provide w ays for otherw ise incompatible components to interact.

The Provider pattern separates source data from data processing objects and the application. It allows application code to be independent of the data source type and data format. A Provider component or service exposes standard methods that the application can call to read and write data. Internally, it converts these calls to the equivalents that match the data source, letting the application work with any source data type (such as any kind of database, XML document, disk file, or data repository) for which a suitable provider is available.

The Adapter pattern has the same advantages, and works in a similar way. Often, the target of an Adapter is some kind of output. For example, a printer driver is an example of an Adapter. ASP.NET itself, and other frameworks such as Enterprise Library, make widespread use of the Provider and Adapter patterns.

The Service Agent, Proxy, and Broker Patterns

Various patterns exist that remove dependencies between a client and a service by using intermediate brokers. There are many different implementations of the basic pattern, some of which use an extra service-agent logic component to connect the client with the local proxy or gateway interface (see Figure 3).

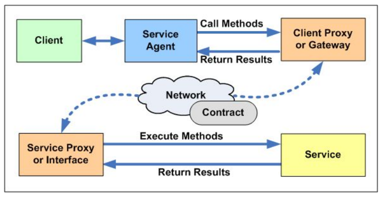


Figure 3. The Service Agent, Proxy, and Broker Patterns: These patterns remove dependencies betw een a service and a client by using intermediate brokers.

The aim of all these patterns is to allow remote connection to, and use of, a service without the client having to know how the service works. The service exposes a contract that defines its interface, such as a Web Service Description Language (WSDL) document for a Web service. A client-side proxy or gateway interface uses the contract to create a suitably formatted request, and passes this to the service interface. The service sends the formatted response back through its gateway interface to the client proxy, which exposes it to the client. In effect, the client just calls the service methods on the client proxy, which returns the results just as if the service itself were a local component.

In the Service Agent pattern, an extra component on the client can perform additional processing and logic operations to further separate the client from the remote service. For example, the Service Agent may perform service address lookup, manipulate or format the client data to match the proxy requirements, or carry out any other kind of processing requirements common to different clients that use the service.

The Repository Pattern

The Repository pattern virtualizes storage of entities in a persistent medium, such as in a database or XML file. For example, a repository may expose database table data as strongly typed Customer and Order objects rather than data sets or data rows. It effectively hides the storage implementation from the application code, and lets applications use a common set of methods without requiring knowledge of the storage mechanism or format. Often, the repository uses a series of providers to connect to the source data (see Figure 4).

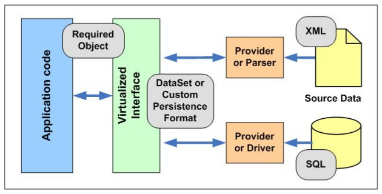


Figure 4. The Repository Pattern: This pattern connects applications to data repositories, isolating the applications from having to know the storage mechanism or format.

The Singleton Pattern
The Singleton pattern defines the creation of a class for which only a single instance can exist. It is useful for exposing read-only data, and static methods that do not rely on instance data. Rather than creating an instance of the class each time, the application obtains a reference to an existing instance using a static method of the class that implements the Singleton pattern.

If this is the first call to the method that returns the instance, the Singleton creates an instance, populates it with any required data, and returns that instance. Subsequent calls to the method simply return this instance. The instance lifetime is that of the application domain—in ASP.NET this is usually the lifetime of the domain `Application` object.

Implementing the Basic Patterns in ASP.NET
The previous sections briefly describe five common and simple types of design pattern. While they are not generally explained in terms of ASP.NET, they are all applicable to ASP.NET and reasonably easy to implement. In fact, ASP.NET and the .NET Framework automatically implement several of them. Figure 5 shows schematically how these patterns relate to ASP.NET applications.

ASP.NET and the Model-View-Presenter Pattern
The ASP.NET code-behind technology, which uses partial classes, provides a natural implementation of the MVP pattern. The code-behind file (the Presenter) contains all the logic and processing code, and populates the page (the View). Event handlers within the code-behind file handle events raised by controls or by the page itself to perform actions when a postback to the server occurs. To complete this pattern, the code-behind file can use a separate data access layer or component (the Model) to read from, write to, and expose the source data—usually accessed through built-in providers that are part of the .NET Framework.

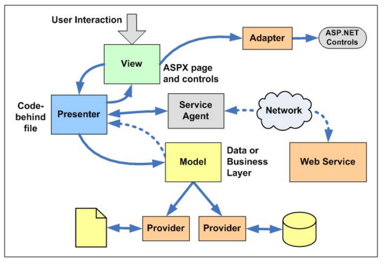
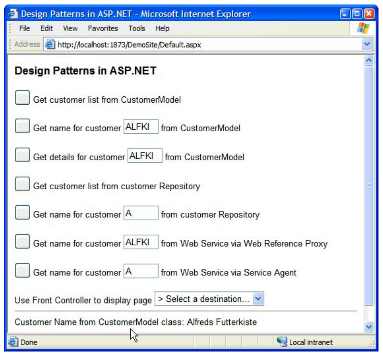


Figure 5. ASP.NET Design Patterns: The schematic show s how ASP.NET implements some basic design patterns.

For example, a page might use a separate class that exposes methods to read and update a database table. This class might use the `SqlClient`, `OleDb`, or another data provider to connect to the database and execute dynamic SQL queries or stored procedures. The code in the code-behind file uses this class to retrieve the data, and populates the controls defined in the ASPX file that generates the user interface. When the user interacts with controls in the page, perhaps changing values in the data, code in the code-behind file executes during the postback and uses the data access class to push the changes back into the database.

The sample application uses the code-behind model throughout, thereby implementing the MVP pattern on each page. The default page (`Default.aspx`) demonstrates the ASP.NET application of the MVP pattern by displaying a View that contains a series of controls that allow users to execute any of a range of functions that fetch and display data (see Figure 6).

Clicking one of the buttons causes a postback to the server and raises the event that corresponds to that button. For example, the interface component for the second button shown in Figure 6 is an ASP.NET `Button` control, declared within the View (the ASPX page), which initiates an event handler named `btn_CustModel_Click` in the Presenter (the code-behind file) to handle the `Click` event:


Design Patterns in ASP.NET - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address
Done
Local intranet

http://localhost:1873/DemoSite/Default.aspx

Design Patterns in ASP.NET

Get customer list from CustomerModel

Get name for customer ALFKI from CustomerModel

Get details for customer ALFKI from CustomerModel

Get customer list from customer Repository

Get name for customer A from customer Repository

Get name for customer ALFKI from Web Service via Web Reference Proxy

Get name for customer A from Web Service via Service Agent

Use Front Controller to display page > Select a destination...

Customer Name from CustomerModel class: Alfreds Futterkiste

Figure 6. MVP Implementation: The View shown here (the Default.aspx HTML page) gets its data from the Presenter (the ASP.NET page), which collates and formats the data from the Model (the data access layer and database).

```
<asp:Button ID="btn_CustModel" runat="server"
   Text="&nbsp;&nbsp;&nbsp;"
   OnClick="btn_CustModel_Click" />
Get name for customer
<asp:TextBox ID="txtID_CustModel" runat="server"
   Text="ALFKI" Columns="3" />
from CustomerModel<p />
```

The Presenter (the code-behind file) contains the declaration of the `btn_CustModel_Click` method, which creates an instance of the Model (a class file named `CustomerModel.cs` in the `App_Code` folder) and calls one of its methods to get the name of the customer identified by the customer ID users enter into the text box next to that button:

```
protected void btn_CustModel_Click(object sender, EventArgs e)
// display the name of the specified customer
{
   try
   {
      // use method of the CustomerModel class
      CustomerModel customers = new CustomerModel();
      Label1.Text += "Customer Name from CustomerModel class: ";
      Label1.Text += customers.GetCustomerName(
         txtID_CustModel.Text);
   }
   catch (Exception ex)
   {
      Label1.Text += "PAGE ERROR: " + ex.Message;
   }
   Label1.Text += "<p />";
}
```

You can view the contents of the files `Default.aspx` (the View), `Default.aspx.cs` (the Presenter), and `CustomerModel.cs` (the Model) that implement the MVP pattern in this example to see all the UI control declarations and code they contain. As you saw in Figure 6, there are buttons to execute a range of CustomerModel methods, as well as methods of the Repository, Web Reference, and Service Agent implementations discussed in the following sections.

Notice that the `Default.aspx` page (see Figure 6) contains a drop-down list where you can select a page or view you want to see. The application loads the selected page using an implementation of the Front Controller pattern, described in more detail in the next article in this series. For now, it's sufficient to explain that you can use the drop-down list to navigate to other pages to see the examples, such as the Use Case Controller example page or the Publish-Subscribe example page.

ASP.NET and the Provider Pattern

ASP.NET uses the Provider pattern in a number of places. These include providers for the

Membership and Role Management system (such as the SqlMembershipProvider and SqlRoleProvider), the Site Navigation system (the XmlSiteMapProvider), and the user profile management system (such as the SqlProfileProvider). The various data access components provided with ASP.NET, such as the SqlDataSource, also rely on providers. These providers are part of the .NET Framework data-access namespaces, and include `SqlClient`, `OleDb`, `Odbc`, and `OracleClient`.

Each provider is configured in the `web.config` file, with the default settings configured in the server's root `web.config` file. Developers can create their own providers based on an interface that defines the requirements, or by inheriting from a base class containing common functionality for that type of provider. For example, developers can extend the `DataSourceControl` class to create their own data source controls that interface with a non-supported or custom data store.

ASP.NET and the Adapter Pattern
ASP.NET uses adapters to generate the output from server controls. By default, server controls use adapters such as the `webControlAdapter` or `DataBoundControlAdapter` to generate the markup (such as HTML) and other content output by the control. Each adapter provides methods such as `Render` and `CreateChildControls` that the server controls call to create the appropriate output.

Alternative adapters are available, for example the CSS Friendly Control Adapters provide more flexibility for customizing the rendered HTML. Developers can also create their own adapters to provide custom output from the built-in controls, or from custom controls.

Visual Studio and the Service Agent and Proxy Patterns
The Proxy and Broker patterns provide a natural technique for connecting to and using remote services without forcing the use of O/S or application-specific protocols such as DCOM. In particular, they are ideal for use with Web services. In Visual Studio, when developers add a Web Reference to a project, Visual Studio automatically collects the Web service description (including a WSDL contract file). This lets the code generate a suitable proxy class that exposes the methods of the remote service to code in the application. Code in the application can then use the remote service by creating an instance of the proxy class and calling the exposed methods defined in the contract. For example, here's the code used in the sample application to call the `CustomerName` method of the `DemoService` service:

```
protected void btn_WSProxy_Click(object sender, EventArgs e)
// display name of specified customer using the web service
{
   try
   {
      // get details from web service
      // use 'using' construct to ensure disposal after use
      using (LocalDemoService.Service svc = new
         LocalDemoService.Service())
      {
         Label1.Text += "Customer name from web service: "
            + svc.CustomerName(txtID_WSProxy.Text);
      }
   }
   catch (Exception ex)
   {
      Label1.Text += "PAGE ERROR: " + ex.Message;
   }
   Label1.Text += "<p />";
}
```

One issue with using Web services is the difficulty in raising exceptions that the client can handle. The usual approach is to return a specific value that indicates the occurrence of an exception or failure. Therefore, one useful feature that a Service Agent (which wraps the Web Reference to add extra functionality) can offer is detecting exceptions by examining the returned value and raising a local exception to the calling code.

The following code, from the `ServiceAgent.cs` class in the sample application, contains a constructor that instantiates the local proxy for the service and stores a reference to it. Then the `GetCustomerName` method can call the `CustomerName` method of the Web service and examine the returned value. If the return value starts with the text "ERROR:" the code raises an exception that the calling routine can handle.

The `GetCustomerName` method also performs additional processing on the value submitted to it. The Web service only matches on a complete customer ID (five characters in the example), and supports partial matches only if the customer ID ends with a wildcard character. The `GetCustomerName` method in the Service Agent checks for partial customer ID values and adds the wildcard character so that the service will return a match on this partial value:

```
// wraps the Web Reference and calls to web service
// to perform auxiliary processing
public class ServiceAgent
{
   LocalDemoService.Service svc = null;

   public ServiceAgent()
   {
      // create instance of remote web service
      try
      {
         svc = new LocalDemoService.Service();
      }
      catch (Exception ex)
      {
         throw new Exception(
            "Cannot create instance of remote service", ex);
      }
   }

   public String GetCustomerName(String custID)
   {
      // add '*' to customer ID if required
      if (custID.Length < 5)
      {
         custID = String.Concat(custID, "*");
      }
      // call web service and raise a local exception
      // if an error occurs (i.e. when the returned
      // value string starts with "ERROR:")
      String custName = svc.CustomerName(custID);
      if (custName.Substring(0,6) == "ERROR:")
      {
         throw new Exception(
            "Cannot retrieve customer name - " + custName);
      }
      return custName;
   }
}
```

To use this simple example of a Service Agent, code in the `Default.aspx` page of the sample application instantiates the Service Agent class and calls its `GetCustomerName` method. It also traps any exception that the Service Agent might generate, displaying the message from the InnerException generated by the agent:

```
protected void btn_WSAgent_Click(object sender, EventArgs e)
// display name of specified customer using the Service Agent
// extra processing in Agent allows for match on partial ID
{
   try
   {
      // get details from Service Agent
      ServiceAgent agent = new ServiceAgent();
      Label1.Text += "Customer name from Service Agent: "
         + agent.GetCustomerName(txtID_WSAgent.Text);
   }
   catch (Exception ex)
   {
      Label1.Text += "PAGE ERROR: " + ex.Message + "<br />";
      if (ex.InnerException != null)
      {
         Label1.Text += "INNER EXCEPTION: " +
            ex.InnerException.Message;
      }
   }
   Label1.Text += "<p />";
}
```

Implementing the Repository Pattern
A data repository that implements the Repository pattern can provide dependency-free access to data of any type. For example, developers might create a class that reads user information from Active Directory, and exposes it as a series of User objects—each having properties such as Alias, EmailAddress, and PhoneNumber. The repository may also provide features to iterate over the contents, find individual users, and manipulate the contents.

Third-party tools are available to help developers build repositories, such as the CodeSmith tools library. Visual Studio includes the tools to generate a typed DataSet that can expose values as properties, rather than as data rows, and this can form the basis for building a repository.

As a simple example, the sample application uses a typed DataSet (CustomerRepository.xsd in the App_Code folder), which is populated from the Customers table in the Northwind database. The DataSet Designer in Visual Studio automatically implements the Fill and GetData methods within the class, and exposes objects for each customer (CustomersRow) and for the whole set of customers (CustomersDataTable).

The class CustomerRepositoryModel.cs in the App_Code folder exposes data from the typed DataSet, such as the GetCustomerList method that returns a populated CustomersDataTable instance, and the GetCustomerName method that takes a customer ID and returns that customer's name as a String:

```
public class CustomerRepositoryModel
{

   public CustomerRepository.CustomersDataTable GetCustomerList()
   {
      // get details from CustomerRepository typed DataSet
      try
      {
         CustomersTableAdapter adapter = new
```

```
               CustomersTableAdapter();
            return adapter.GetData();
         }
         catch (Exception ex)
         {
            throw new Exception(
               "ERROR: Cannot access typed DataSet", ex);
         }
      }

      public String GetCustomerName(String custID)
      {
         // get details from CustomerRepository typed DataSet
         try
         {
            if (custID.Length < 5)
            {
               custID = String.Concat(custID, "*");
            }
            CustomerRepository.CustomersDataTable customers =
               GetCustomerList();
            // select row(s) for this customer ID
            CustomerRepository.CustomersRow[] rows =
               (CustomerRepository.CustomersRow[])customers.Select(
               "CustomerID LIKE '" + custID + "'");
            // return the value of the CompanyName property
            return rows[0].CompanyName;
         }
         catch (Exception ex)
         {
            throw new Exception(
               "ERROR: Cannot access typed DataSet", ex);
         }
      }
   }
```

Implementing the Singleton Pattern
The sample application uses the Front Controller pattern to allow users to specify the required View using a short and memorable name. The Front Controller, described in detail in a later section, relies on an XML file that maps the visible short and memorable names to the actual URLs for this view. To expose the XML document's content to the Front Controller code, the application uses a Singleton instance of the class named `TransferUrlList.cs` (in the `App_Code` folder).

This approach provides good performance, because the class is loaded at all times and the Front Controller has only to call a static method to get a reference to the single instance, then call the method that translates the memorable name into a URL.

To implement the Singleton pattern, the class contains a `private` default constructor (a constructor that takes no parameters), which prevents the compiler from adding a default `public` constructor. This also prevents any classes or code from calling the constructor to create an instance.

The `TransferUrlList` class also contains a static method that returns the single instance, creating and populating it from the XML file if there is no current instance. The class uses static local variables to store a reference to the instance, and—in this example—to hold a StringDictionary containing the list of URLs loaded from the XML file:

```
   // Singleton class to expose a list of URLs for
```

```
// Front Controller to use to transfer to when
// special strings occur in requested URL
public class TransferUrlList
{
   private static TransferUrlList instance = null;
   private static StringDictionary urls;

   private TransferUrlList()
   // prevent code using the default constructor by
   // making the construtor private
   { }

   public static TransferUrlList GetInstance()
   // public static method to return single instance of class
   {
      // see if instance already created
      if (instance == null)
      {
         // create the instance
         instance = new TransferUrlList();
         urls = new StringDictionary();
         String xmlFilePath = HttpContext.Current.Request.MapPath(
            @"~/xmldata/TransferURLs.xml");
         // read list of transfer URLs from XML file
         try
         {
            using (XmlReader reader =
               XmlReader.Create(xmlFilePath))
            {
               while (reader.Read())
               {
                  if (reader.LocalName == "item")
                  {
                     // populate StringDictionary
                     urls.Add(reader.GetAttribute("name"),
                        reader.GetAttribute("url"));
                  }
               }
            }
         }
         catch (Exception ex)
         {
           throw new Exception(
              "Cannot load URL transfer list", ex);
         }
      }
      return instance;
   }
   ...
}
```

A StringDictionary provides good performance, as the method that searches for a matching name and URL in the list can use the `ContainsKey` method, and then access the matching value using an indexer. If there is no match, the code returns the original value without translating it. The following code shows the `GetTransferUrl` method that performs the URL translation:

```
public String GetTransferUrl(String urlPathName)
// public method to return URL for a specified name
// returns original URL if not found
```

```
{
   if (urls.ContainsKey(urlPathName))
   {
      return urls[urlPathName];
   }
   else
   {
      return urlPathName;
   }
}
```

To use the TransferUrlList Singleton class, code in the Front Controller just has to get a reference to the instance, and then call the `GetTransferUrl` method with the URL to translate as the value of the single parameter:

```
...
// get Singleton list of transfer URLs
TransferUrlList urlList = TransferUrlList.GetInstance();
// see if target value matches a transfer URL
// by querying the list of transfer URLs
// method returns the original value if no match
String transferTo = urlList.GetTransferUrl(reqTarget);
...
```

That completes this installment. Part 2 will discuss Controller Patterns for ASP.NET.