1

Hacks Revisited

Hacks exist in an ever-changing world. One day they are the cutting-edge, clever inventions of a developer with a need. Another day in the future, they can become mainstream code or practices that are integrated into a product or process. The mark of a successful hack is that one day it will become the norm.

This chapter is all about hacks that have become successful. We revisit ASP.NET v1.1 to look at a group of hacks that received much attention. They became so useful that Microsoft felt compelled to add support for them in ASP.NET v2.0. Rather than redo hacks that have already been invented, this chapter highlights the pioneers and their work that influenced Microsoft to add support in ASP.NET v2.0. For those of you making the move to ASP.NET 2.0, you'll learn how these hacks have been integrated into the .NET Framework and related Visual Studio 2005 support. Seeing how previous hacks have positively influenced the current version of ASP.NET can give you and others a greater appreciation for hacks, and provide motivation for new hacks that will have a positive influence on future versions of ASP.NET.

Wizards Hacks Replaced by ASP.NET 2.0

Wizards are user interface tools for gently guiding users through a process. This discussion refers to wizards that you add to your applications. Developers have used them for many years, from installation programs to simplifying complex activities through a series of steps. They've traditionally been a normal part of client desktop applications, but have surfaced as hacks in Web applications.

ASP.NET Wizard Pioneers

An early ASP.NET v1.1, a wizard hack by John Peterson was published as a sample on ASP101.com (`asp101.com/samples/wizard_aspx.asp`) titled "ASP.NET version of Wizard (Multi-Page Form)." This was an update from a previous implementation for classic ASP 3.0. After support for ASP.NET 2.0 wizards was announced, Tom Blanchard wrote the article "CodeSnip: Simulating the ASP.NET 2.0 Wizard Control with ASP.NET 1.x." at `123aspx.com/redir.aspx?res=32798`. Another pre-existing

solution that some people used was the Wizard Navigator in the User Interface Process (UIP) Application Block, created by the Microsoft Patterns and Practices Group at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/cabctp.asp`.

> *Microsoft's online links break a lot, but you should still be able to find the UI Application Block with a search of the MSDN Architecture Center's Patterns and Practices resource pages.*

Wizards in ASP.NET v2.0

Wizards are no longer a hack in ASP.NET 2.0. They are easy to use and well supported by Visual Studio 2005. The next section shows how to implement a wizard.

Implementing an ASP.NET 2.0 Wizard

ASP.NET 2.0 wizards are implemented as controls. To use them, simply add a new wizard control to the Visual Designer and start setting properties. This section describes how to implement ASP.NET 2.0 wizards through a sample menu selection application. There are various choices to be made along the way, and the wizard will manage navigation based on input preferences. The following steps explain how to create our sample application:

1. Create a new Web project. To do this, select File➪New➪Web Site, select the ASP.NET Web Site template, and change the directory location to WizardDemo as shown in Figure 1-1.

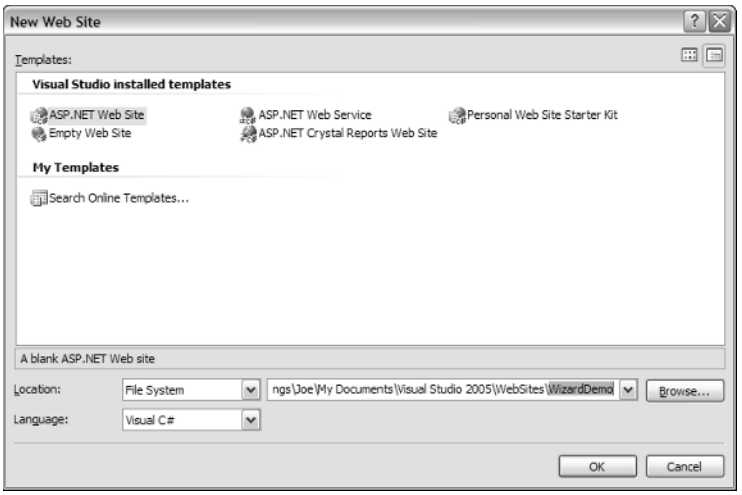

Figure 1-1

2. Click the OK button and Visual Studio 2005 will create a new project, as shown in Figure 1-2.
3. If you are in HTML view, as shown in Figure 1-2, click the Design link at the bottom left of the editor to enter Design view.
4. In Design view, locate the Toolbox, which is on the left side of the IDE. You can open it by hovering over the tab labeled Toolbox. Click on the Standard area of the Toolbox and select the Wizard control. Drag and drop a Wizard control onto the form as shown in Figure 1-3.

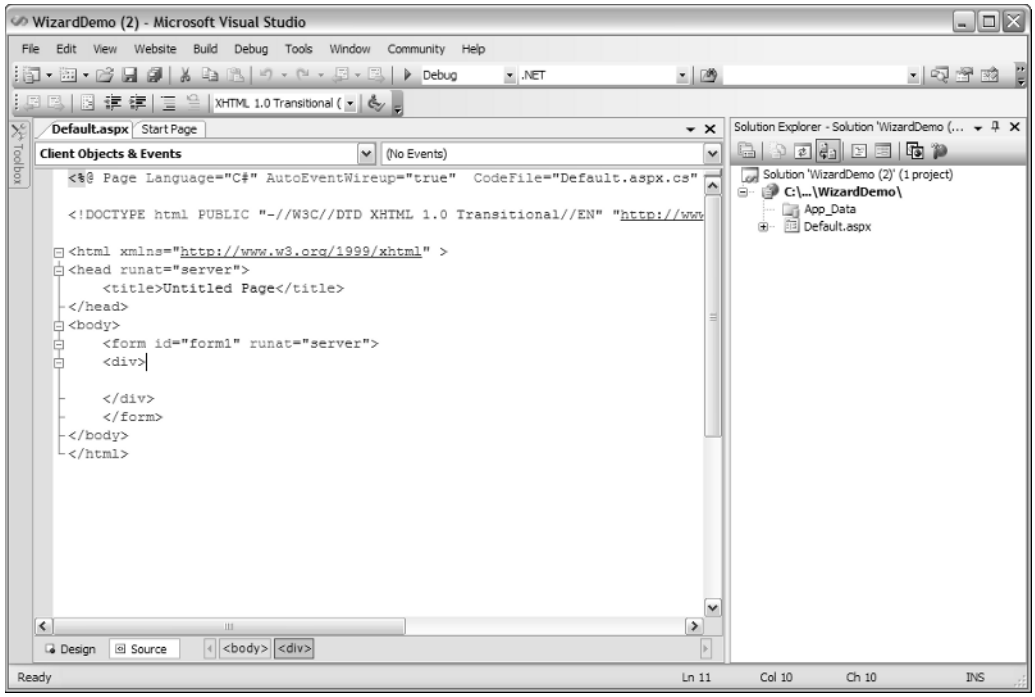

Figure 1-2

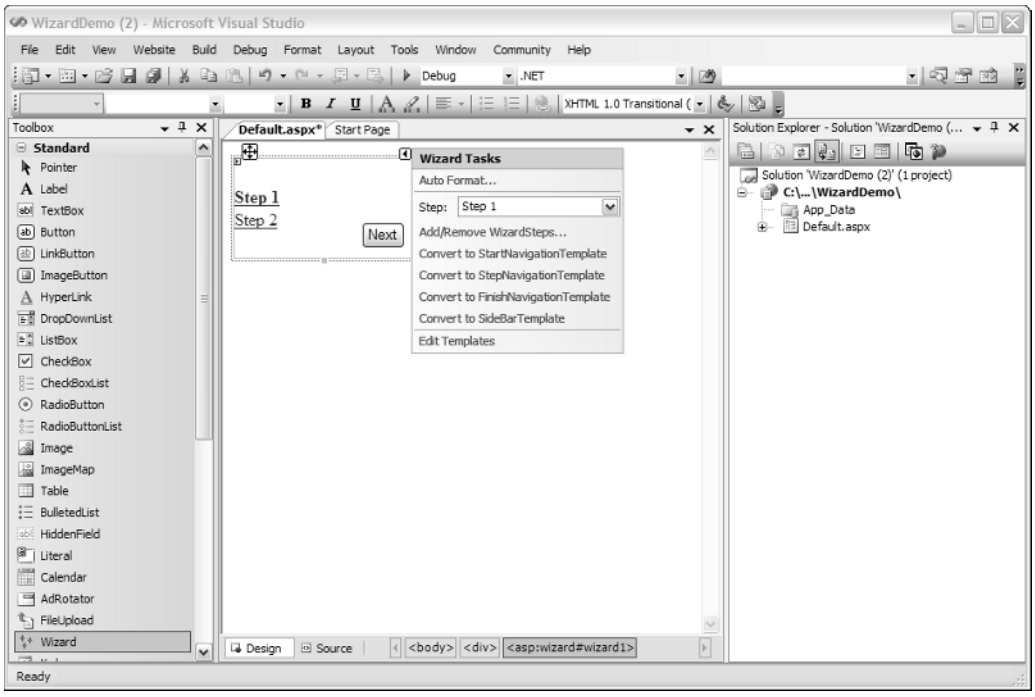

Figure 1-3

5. Select View➪Properties Window to show the Properties window. In the Properties window, set the `HeaderText` property to **Menu Selector**.
6. Enlarge the Wizard control to approximately 400 × 200.
7. The Wizard control shows a couple of links labeled Step 1 and Step 2 the first time it is added to a Web form. Select Step 1 in the Wizard control.
8. Select the edit area in the Wizard control and type **What is your Eating Preference?**
9. Drag and drop a RadioButtonList control from the Toolbox to the Wizard control edit area, select Edit Items on the Action List, and add **Meat Eater** and **Vegetarian** items to the `Text` properties of two new items in the ListItemCollection Editor. Figure 1-4 shows the results of steps 5 through 9 of these instructions.

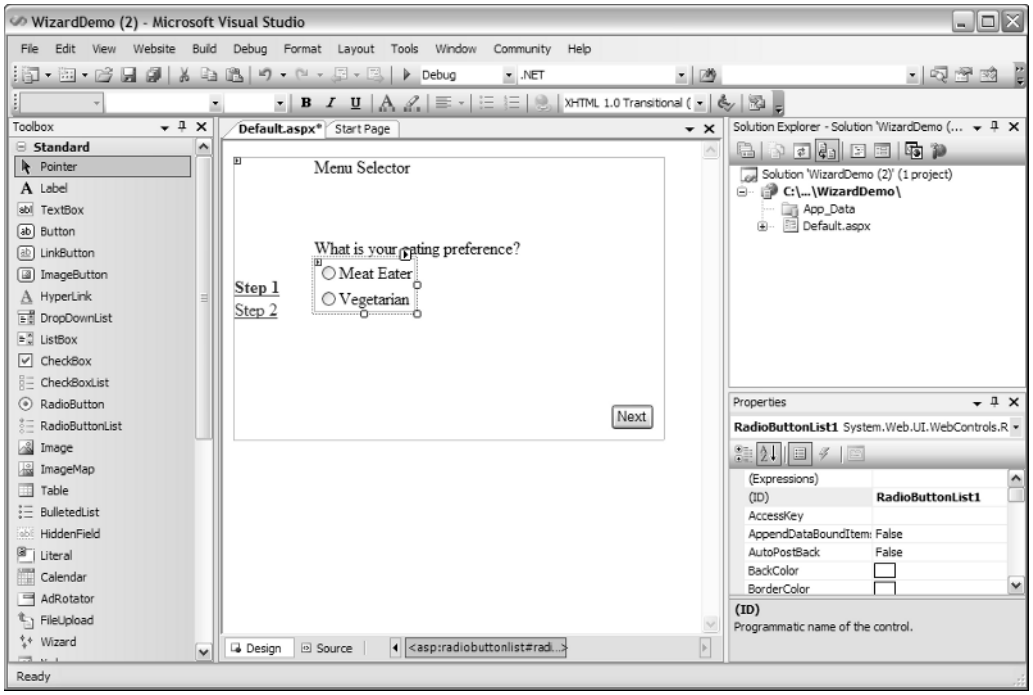

Figure 1-4

10. Select Step 2 in the Wizard control.
11. Type **Please Select Main Course:** in the edit area.
12. Drag and drop a RadioButtonList control from the Toolbox to the Wizard control edit area, select Edit Items in the Action List, and add **Chicken**, **Fish**, and **Steak** items to the `Text` properties of three new items in the ListItemCollection Editor. Figure 1-5 shows the results of steps 10 through 12 of these instructions.
13. Click Add/Remove WizardSteps from the Wizard Tasks list and add Step 3 to the Wizard control.

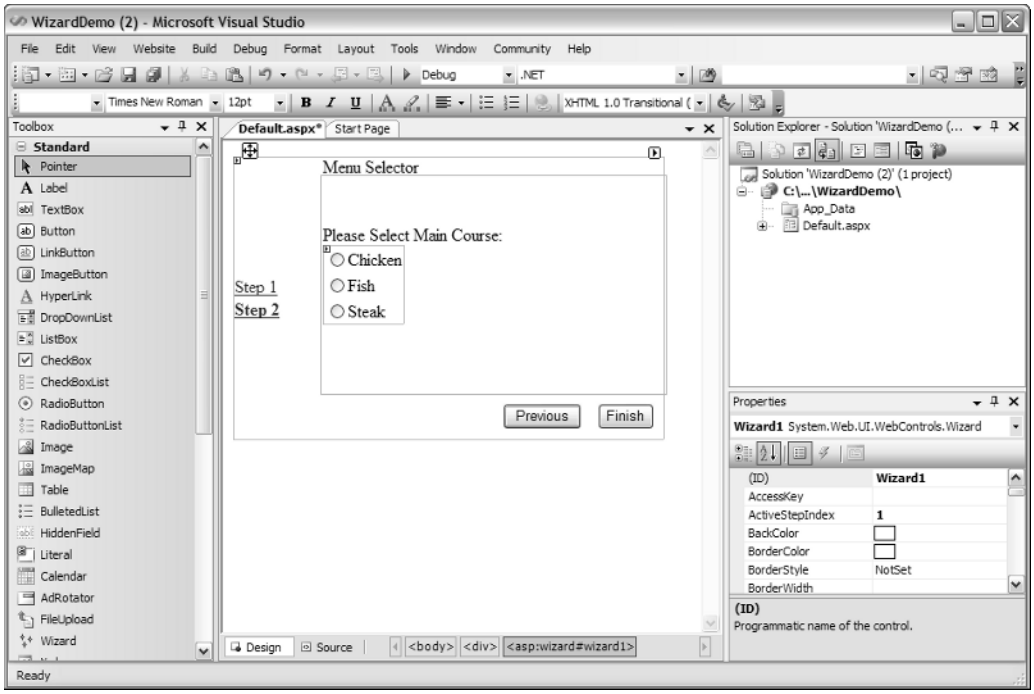

Figure 1-5

14. Select the new Step 3 link that you just added to the Wizard control.

15. Type **Please Select Main Course:** in the edit area.

16. Drag and drop a RadioButtonList control from the Toolbox to the Wizard control edit area. Select Edit Items in the Action List and add **Bread**, **Salad**, and **Veggie Tray** items to the `Text` properties of three new items in the ListItemCollection Editor. Figure 1-6 shows the results of steps 13 through 16 of these instructions.

17. Click Add/Remove WizardSteps from the Wizard Tasks list and add a new Step 4 to the Wizard control.

18. Select the Step 4 link that you just added to the Wizard control.

19. Type **Please Select Beverage:** in the edit area.

20. Drag and drop a RadioButtonList control from the Toolbox to the Wizard control edit area, select Edit Items in the Action List, and add Coffee, Water, and Wine items to the `Text` properties of three new items in the ListItemCollection Editor. Figure 1-7 shows the results of steps 17 through 20 of these instructions.

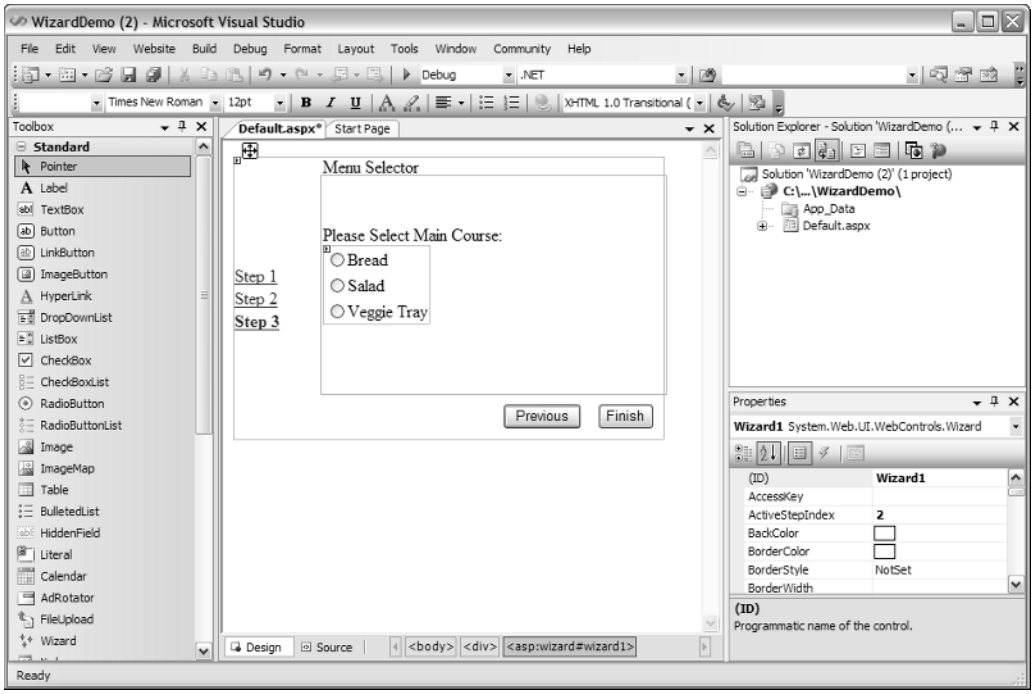

Figure 1-6

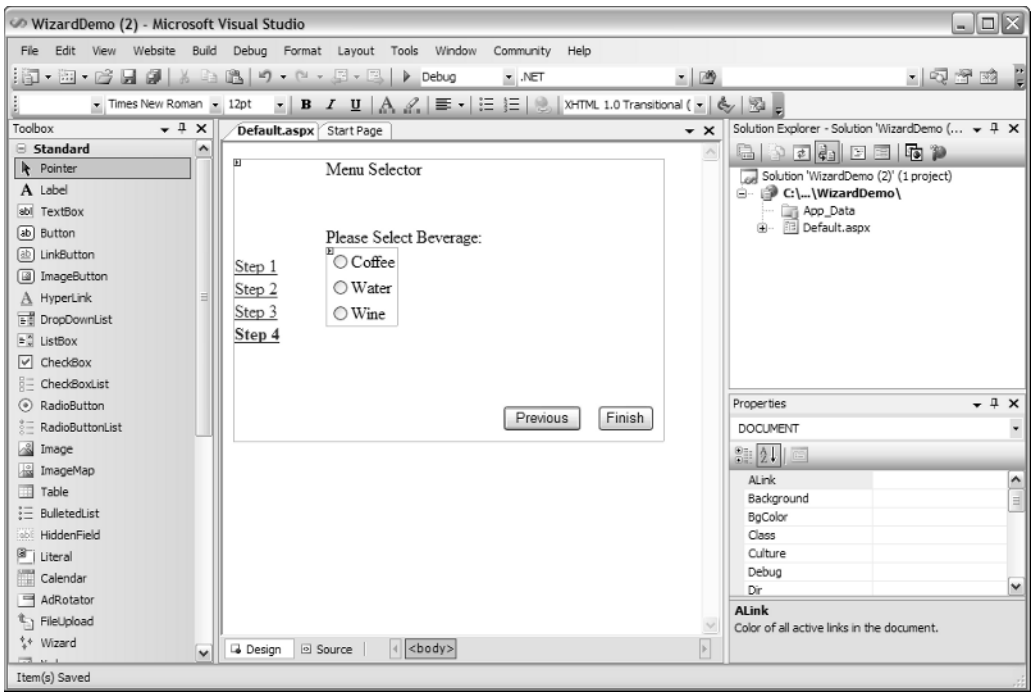

Figure 1-7

This creates HTML with a Wizard that contains multiple steps. Listing 1-1 shows the HTML of this page with modifications made in the later parts of this section. The HTML you see in Listing 1-1 was automatically generated by Visual Studio 2005 as you performed steps 1 through 20. Select the HTML tab at the bottom of the Visual Studio 2005 editor to switch from Design view to HTML view. When you do this, you will see code similar to Listing 1-1.

Listing 1-1: The ASP.NET v2.0 Wizard control with multiple steps

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Wizard Demo</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:Wizard ID="Wizard1" runat="server"
                    ActiveStepIndex="0" HeaderText="Menu Selector"
                    Height="200px" Width="400px"
                    OnFinishButtonClick="Wizard1_FinishButtonClick"
                    OnNextButtonClick="Wizard1_NextButtonClick">
            <WizardSteps>
                <asp:WizardStep runat="server" Title="Step 1">
                    What is your eating preference?<br />
                    <br />
                    <asp:RadioButtonList ID="RadioButtonList1" runat="server">
                        <asp:ListItem>Meat Eater</asp:ListItem>
                        <asp:ListItem>Vegetarian</asp:ListItem>
                    </asp:RadioButtonList>
                </asp:WizardStep>
                <asp:WizardStep runat="server" Title="Step 2">
                    Please select main course:<br />
                    <br />
                    <asp:RadioButtonList ID="RadioButtonList2" runat="server">
                        <asp:ListItem>Chicken</asp:ListItem>
                        <asp:ListItem>Fish</asp:ListItem>
                        <asp:ListItem>Steak</asp:ListItem>
                    </asp:RadioButtonList>
                </asp:WizardStep>
                <asp:WizardStep runat="server" Title="Step 3">
                    Please select main course:<br />
                    <br />
                    <asp:RadioButtonList ID="RadioButtonList3" runat="server">
                        <asp:ListItem>Bread</asp:ListItem>
                        <asp:ListItem>Salad</asp:ListItem>
                        <asp:ListItem>Veggie Tray</asp:ListItem>
                    </asp:RadioButtonList>
                </asp:WizardStep>
                <asp:WizardStep runat="server" Title="Step 4">
```

(continued)

Listing 1-1 *(continued)*

```
                    Please select beverage:<br />
                    <br />
                    <asp:RadioButtonList ID="RadioButtonList4" runat="server">
                        <asp:ListItem>Coffee</asp:ListItem>
                        <asp:ListItem>Water</asp:ListItem>
                        <asp:ListItem>Wine</asp:ListItem>
                    </asp:RadioButtonList>
                </asp:WizardStep>
            </WizardSteps>
        </asp:Wizard>
    </div>
    </form>
</body>
</html>
```

As you can see from the steps of Listing 1-1 and Figure 1-8, the user is either a Meat Eater or a Vegetarian. They should see only the menu items corresponding to their selection.

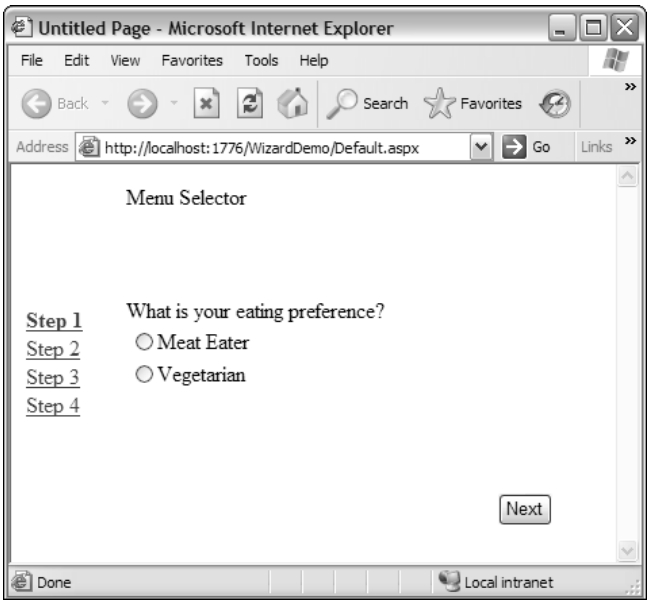

Figure 1-8

To specify different menus, the Wizard control will have to navigate to a different step, depending on what was selected in Step 1. ASP.NET Wizard controls support events, and that is what we'll use to ensure that the proper step appears, depending on eating preference. Listing 1-2 shows how to navigate properly when the user clicks the Next button. Click the lightning bolt icon on the Properties window and double-click the NextButtonClick event to produce an event handler shell, which will have a default name of Wizard1_NextButtonClick. Type in the algorithm for the Wizard1_NextButtonClick event handler as shown in Listing 1-2.

Listing 1-2: Altering the sequential progress of a wizard

```
protected void Wizard1_NextButtonClick(
    object sender, WizardNavigationEventArgs e)
{
    if (Wizard1.ActiveStepIndex == 0)
    {
        if (RadioButtonList1.SelectedValue == "Vegetarian")
        {
            Wizard1.ActiveStepIndex = 2;
        }
    }

    if (Wizard1.ActiveStepIndex == 1)
    {
        Wizard1.ActiveStepIndex = 3;
    }
}
```

By capturing the NextButtonClick event, as shown in Listing 1-2, you can control the navigation sequence of the Wizard. This event is called in the context of the current page when the Next button is clicked. Therefore, check the ActiveStepIndex first to determine the current page, which is indicated by a zero-based integer. All step controls are accessible during the postback, so we can get to their values easily. In this case, the code sets the ActiveStepIndex to the Step 3 page, at index 2, if the eating preference is Vegetarian.

When the eating preference is Meat Eater, the Wizard will naturally navigate to Step 2, at index 1. In this case, clicking the Next button should bring the user to Step 4, at index 3, to select a beverage. The Vegetarian who navigated to Step 3 will naturally move to Step 4 after clicking the Next button.

When the Wizard reaches the end, the Next button will be replaced with a Finish button. You can capture an event from the Finish button to take action when it is clicked, as shown in Listing 1-3.

Listing 1-3: Handling the Finish button event

```
protected void Wizard1_FinishButtonClick(
    object sender, WizardNavigationEventArgs e)
{
    if (RadioButtonList1.SelectedValue == "Vegetarian")
    {
        Response.Write(
            "Your meal will be " +
            RadioButtonList3.SelectedValue +
            " and " +
            RadioButtonList4.SelectedValue);
    }
    else
    {
        Response.Write(
           "Your meal will be " +
           RadioButtonList2.SelectedValue +
           " and " +
           RadioButtonList4.SelectedValue);
    }
}
```

In Listing 1-3, we're primarily interested in whether the eating preference was Meat Eater or Vegetarian so we can extract values from the proper controls. The code emits, via Response.Write, the selected menu items, depending on whether the user selected Meat Eater or Vegetarian when running the Wizard.

Going through the process of adding steps, handling events, and altering properties, you can see how sophisticated the ASP.NET v2.0 Wizard control is. It's an example of an extremely useful hack being transformed into a full product feature.

Master Pages: Then and Now

Master Pages enable you to develop a visual template for a common look and feel across multiple pages of a website. In ASP.NET v1.1, developers used a few different techniques to accomplish similar goals, including user controls, inheriting from a common base page class, and a form of Master Pages that used templates. This section refers you to resources showing how Master Page template hacks were implemented in ASP.NET v1.1, and then describes how they are implemented as a major feature of ASP.NET v2.0.

Master Page Templates in ASP.NET v1.1

The closest thing to ASP.NET v2.0 Master Pages in ASP.NET v1.1 was one of the template management techniques. Paul Wilson, `wilsondotnet.com`, has written several articles on the subject of page templates. In October 2003, he wrote "Page Templates: Introduction" (`http://authors.aspalliance.com/PaulWilson/Articles/?id=14`) for ASP Alliance, which describes the user controls and page inheritance techniques. However, more interesting is his article "MasterPages: Introduction" (`http://authors.aspalliance.com/PaulWilson/Articles/?id=13`) for ASP Alliance. For many developers, this was essentially a first look at what was to come by way of Master Pages in ASP.NET v2.0. Paul Wilson's article shows how to accomplish the same thing with ASP.NET v1.1.

There are other credible ways to accomplish templating in ASP.NET v1.1. In his Code Project article "MasterPages Reinvented: A Component-Based Template Engine for ASP.NET" (`codeproject.com/aspnet/sumitemplatecontrols.asp`), Philipp Sumi shows another method of using Master Page templates in ASP.NET v1.x. There are other examples available, but the articles by Paul Wilson and Philipp Sumi are credible references you can use in ASP.NET v1.1.

Master Pages in ASP.NET v2.0

In ASP.NET v2.0, Master Pages have grown from a great hack to a major feature. They're built into ASP.NET with new page directives, placeholders, and controls that enable you to create a common look and feel across your application or website. Additionally, they are well supported in Visual Studio 2005.

Implementing Master Pages

In ASP.NET v2.0, Master Pages are implemented as a Web page with a default look and feel that content pages can use. Multiple content pages can use this Master Page's common look and feel, and all they need to do is provide content markup, which is merged with the Master Page. The example in this section walks through creating a Master Page and simple content page with VS 2005.

To get started, create a new Web project and delete the `Default.aspx` page that is automatically created. We'll re-create `Default.aspx` later as a content page that uses a Master Page we're about to create.

1. To create a Master Page, right-click on the Web project in Solution Explorer, select Add New Item, select Master Page, give it the name **Company.master**, and click the OK button. This creates a new Master Page.
2. If the Master Page is in HTML view, change it to Design view by clicking the Design button at the bottom left corner of the editor. Add a new table to Company.master by selecting Layout➪Insert Table.
3. On the Insert Table dialog, choose Template and then pick Header, footer, and side from the drop-down list, as shown in Figure 1-9.

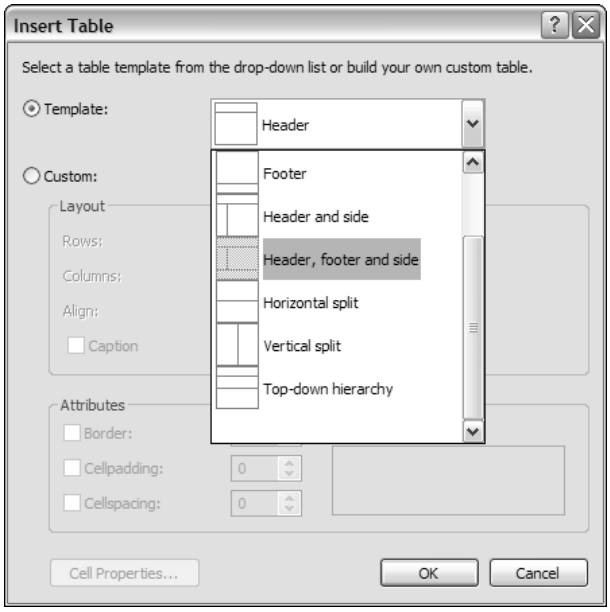

Figure 1-9

4. Once the table has been added to the page, select the cell in the top table row and type **My Company Header**, select the first cell in the second table row and type **Menu Goes Here**, and select the cell in the third (last) row and type **Copyright (C) 2006 My Company, All Rights Reserved**.
5. When a Master Page is first created, it has a ContentPlaceHolder control added to it automatically. Company.master has this also. Drag and drop the ContentPlaceHolder control to the second cell of the second row in the table that you just inserted into the Company.master Master Page. The results should look similar to Figure 1-10.

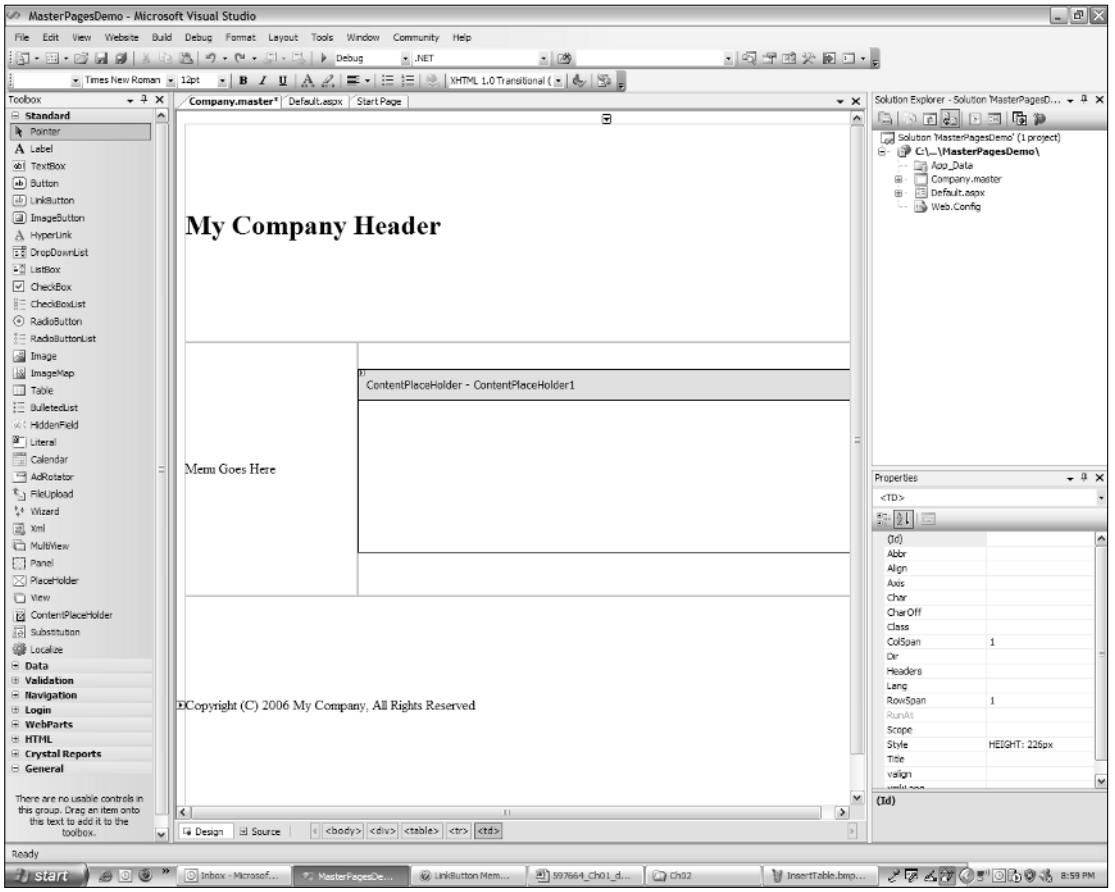

Figure 1-10

After creating and modifying the Company.master Master Page, you can view the automatically generated HTML code by clicking the HTML button at the bottom-left corner of the editor. Listing 1-4 shows what the HTML for the Company.master Master Page looks like after making the modifications described in the previous paragraph.

Listing 1-4: A Master Page with formatting for custom page elements: Company.master

```
<%@ Master Language="C#" AutoEventWireup="true" CodeFile="Company.master.cs"
Inherits="Company" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>My Company</title>
</head>
<body>
```

```
    <form id="form1" runat="server">
    <div>
        <table border="0" cellpadding="0" cellspacing="0"
               style="width: 100%; height: 100%">
            <tr>
                <td colspan="2" style="height: 200px">
                    <h1>My Company Header</h1>
                </td>
            </tr>
            <tr>
                <td style="width: 200px">
                    Menu Goes Here</td>
                <td>
        <asp:contentplaceholder id="ContentPlaceHolder1" runat="server">
        </asp:contentplaceholder>
                </td>
            </tr>
            <tr>
                <td colspan="2" style="height: 200px">
                    Copyright (C) 2006 My Company, All Rights Reserved
                </td>
            </tr>
        </table>
    </div>
    </form>
</body>
</html>
```

The first thing you should notice about Listing 1-4 is that it contains a Master directive at the top of the page. As implied by the codefile and inherits attributes, you can add a code file to the Master Page. Most of the page is primarily HTML formatting markup.

The other pertinent part of this page is the contentplaceholder element. You can place default content in here, but content pages will use the id to specify where their content will appear, which will override any default content.

You can now use this Master Page with any content pages. The process of creating a content page is very similar to the process required to create any other Web Form. Right-click on the Web project in Solution Explorer, select Add New Item, select Web Form, accept the suggested name of Default.aspx, check the Select Master Page option, and click the OK button. Listing 1-5 shows the new content page, after I modified it.

Listing 1-5: Content page that uses a Master Page: Default.aspx

```
<%@ Page
    Language="C#"
    MasterPageFile="~/Company.master"
    AutoEventWireup="true"
    CodeFile="Default.aspx.cs"
    Inherits="_Default"
    Title="Content Page" %>

    <asp:Content ID="Content1"
```

(continued)

Listing 1-5 *(continued)*

```
                ContentPlaceHolderID="ContentPlaceHolder1"
                Runat="Server">
            <h2>This is my content.</h2>
        </asp:Content>
```

The two interesting parts of Listing 1-5 are the MasterPageFile attribute and the Content element. The Page directive contains a new attribute named MasterPageFile, specifying the filename of the Master Page to use.

It is evident from Listing 1-5 that content pages don't have HTML markup as normal Web Forms do. Instead, add a Content element that identifies where to place content markup within the Master Page by specifying the id of the contentplaceholder (in the Master Page) to use with the ContentPlaceHolderID attribute. Only the markup in the Content element appears in the Master Page.

Because you don't have normal HTML markup, such as a Head, set the Title attribute of the Page directive with what you would have normally put into the Head title element.

This shows how elegant and simple it is to use Master Pages in ASP.NET v2.0. Supported by VS 2005, it is a much easier solution than ASP.NET v1.1 hacks.

URL Rewriting

URL rewriting is the practice of accepting URLs with meaningful naming conventions and translating them into real query strings. A couple of reasons why you would want meaningful naming conventions include the capability to organize information into a logical hierarchy or to mask query string parameters. In this section I'll show you how URL rewriting can improve your user interface, describe the old and new ways to accomplish URL rewriting, and give you some code that demonstrates the concept.

> **I also add some extra code in this section to demonstrate best practices in n-tier architecture and data handling, rather than cheat with a simpler datasource control.**

Why URL Rewriting?

An example of hierarchical organization could be described by looking at how a blog is organized by time. From the user's perspective, the following query string could be considered very cryptic:

```
http://www.someblogsite.com/username/?y=2005&m=01&d=31
```

The preceding query string returns blog entries for January 1, 2005, which isn't easy to determine. You could modify code so that the parameters were more meaningful, such as this:

```
http://www.someblogsite.com/username/?year=2005&month=01&day=31
```

No matter how query string parameters are presented, average users will have a hard time understanding. Instead, it would be much clearer to write the query string with a nice hierarchical representation, like this:

```
http://www.someblogsite.com/username/2005/January/31
```

For a typical visitor of a site, the preceding URL is not too hard to figure out. For example, if users were to remove the day, they would see all of the entries for the entire month of January.

Even when you don't have a natural hierarchy or just a single parameter, a URL with a meaningful name, rather than a query parameter, is still easier for a consumer to understand.

The ASP.NET v1.1 Hack

One of the best resources available for performing URL rewriting in ASP.NET v1.x is Scott Mitchell's MSDN article titled "URL Rewriting in ASP.NET" at `msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspp/html/urlrewriting.asp`. In this article, Scott describes how to perform URL rewriting with both HTTP Modules and HTTP Handlers, and explains when each is best to use. He also builds a reusable URL rewriting engine that uses regular expressions via a configuration file.

The ASP.NET v2.0 Replacement

Rewriting URLs is supported, somewhat, in ASP.NET v2.0 with the `urlMappings` configuration element. Add a new entry to `web.config`, similar to this for mapping a URL:

```
<urlMappings enabled="true">
  <add url="~/Articles/AspDotNet/UrlRewriting"
mappedUrl="~/Articles.aspx?cat=1&id=16" />
</urlMappings>
```

The `url` attribute is what the user sees and the `mappedUrl` attribute describes the actual requested page. In the preceding `urlMappings` element, imagine that there is an articles page that dynamically returns articles based on category and article identifier. The `url` shows the preferred user interface, but the `mappedUrl` attribute shows the actual page and parameters that will be requested.

Implementing the URL Mapping Capability

I've written an example application to show how to use this capability. It is a variation of the article viewing concept shown previously, but based on year and month. For example, the articles application will allow you to select the year and then the month. Each page is displayed with readable URLs that you can use to navigate the application by simply modifying the page address. Listing 1-6 shows the initial page of the articles application.

Listing 1-6: Main page that uses readable URLs for identifying years: Default.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
```

(continued)

Listing 1-6 *(continued)*

```
<head runat="server">
    <title>.NET Article Archive</title>
</head>
<body>
    <form id="form1" runat="server">
        <h1>.NET Article Archive</h1>
        <p>
            Pick a Year:
        </p>
        <p>
            <asp:HyperLink ID="HyperLink1" runat="server"
                 NavigateUrl="~/2006">2006</asp:HyperLink><br />
            <asp:HyperLink ID="HyperLink2" runat="server"
                 NavigateUrl="~/2005">2005</asp:HyperLink>
        </p>
    </form>
</body>
</html>
```

The `NavigateUrl` attribute of the `HyperLink` elements contain readable URLs that will be rewritten as query string parameters. Similarly, after users select a year, they will see the year page shown in Listing 1-7.

Listing 1-7: Year page with readable URLs: YearView.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="YearView.aspx.cs"
Inherits="YearView" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Articles for the Year</title>
</head>
<body>
    <form id="form1" runat="server">
        <h1>
            <asp:Label ID="lblTitle"
                       runat="server"
                       Text="Articles for the Year #">
            </asp:Label>
        </h1>
        <p>
            <asp:Panel ID="pnlMonths" runat="server" Height="50px" Width="125px">
                <asp:HyperLink ID="hypJanuary"   runat="server"
                    NavigateUrl="~/YEAR/01">January</asp:HyperLink><br />
                <asp:HyperLink ID="hypFebruary"  runat="server"
                    NavigateUrl="~/YEAR/02">February</asp:HyperLink><br />
                <asp:HyperLink ID="hypMarch"     runat="server"
                    NavigateUrl="~/YEAR/03">March</asp:HyperLink><br />
                <asp:HyperLink ID="hypApril"     runat="server"
                    NavigateUrl="~/YEAR/04">April</asp:HyperLink><br />
```

```
                <asp:HyperLink ID="hypMay"       runat="server"
                    NavigateUrl="~/YEAR/05">May</asp:HyperLink><br />
                <asp:HyperLink ID="hypJune"      runat="server"
                    NavigateUrl="~/YEAR/06">June</asp:HyperLink><br />
                <asp:HyperLink ID="hypJuly"      runat="server"
                    NavigateUrl="~/YEAR/07">July</asp:HyperLink><br />
                <asp:HyperLink ID="hypAugust"    runat="server"
                    NavigateUrl="~/YEAR/08">August</asp:HyperLink><br />
                <asp:HyperLink ID="hypSeptember" runat="server"
                    NavigateUrl="~/YEAR/09">September</asp:HyperLink><br />
                <asp:HyperLink ID="hypOctober"   runat="server"
                    NavigateUrl="~/YEAR/10">October</asp:HyperLink><br />
                <asp:HyperLink ID="hypNovember"  runat="server"
                    NavigateUrl="~/YEAR/11">November</asp:HyperLink><br />
                <asp:HyperLink ID="hypDecember"  runat="server"
                    NavigateUrl="~/YEAR/12">December</asp:HyperLink>
            </asp:Panel>
        </p>
    </form>
</body>
</html>
```

Listing 1-7 shows `HyperLink` elements with `NavigateUrl` attributes set to readable URLs. They include a level for each month of the year, where the number corresponds to the order of the month. To process these URLs properly, edit the code file for the YearView Web form as shown in Listing 1-8.

Listing 1-8: Reading the query string parameter of a page called with a parameterized URL: YearView.aspx.cs

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

public partial class YearView : System.Web.UI.Page
{
    string year;  // passed in query string

    protected void Page_Load(object sender, EventArgs e)
    {
        // we'll use this in multiple methods
        year = Request.QueryString["year"];

        // set page title
        SetTitle();

        // configure months to refer to proper page
```

(continued)

Listing 1-8 *(continued)*

```
            SetMonths();
        }

        /// <summary>
        /// configure months to refer to proper page
        /// </summary>
        private void SetMonths()
        {
            foreach (Control ctrl in pnlMonths.Controls)
            {
                HyperLink monthLink = ctrl as HyperLink;

                if (monthLink != null)
                {
                    monthLink.NavigateUrl =
                        monthLink.NavigateUrl.Replace("YEAR", year);
                }
            }
        }

        /// <summary>
        /// set page title
        /// </summary>
        private void SetTitle()
        {
            lblTitle.Text = "Articles for the Year " + year;
        }
    }
```

The Page_Load method of Listing 1-8 extracts the year parameter out of the query string so it can be used in subsequent methods. The SetTitle method uses this value to rewrite the title on the page with the correct year.

Also, notice the `NavigateUrl` properties for the HyperLink controls in Listing 1-8, where they all contain the text `"YEAR"` in the year's position of the URL. This makes the code more dynamic because, depending on the year, these `NavigateUrl` properties must be rewritten. That's what the `SetMonths` method of Listing 1-8 is for. The code in Listing 1-8 deliberately places each HyperLink control into a Panel so we can get to its `Controls` collection in code. Therefore, in the code file, the `SetMonths` method can simply iterate through the Controls collection and set the year with a simple `string.Replace` method call. To get this built-in URL rewriting functionality to work for this application, there is a `web.config` file with a `urlMapping` element, as shown in Listing 1-9.

Listing 1-9: The urlMapping element in a web.config file enables URL rewriting in ASP.NET v2.0

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <urlMappings>
      <add url="~/2006"
           mappedUrl="~/YearView.aspx?year=2006"/>
```

```
        <add url="~/2006/01"
             mappedUrl="~/MonthView.aspx?year=2006&amp;month=01"/>
        <add url="~/2006/02"
             mappedUrl="~/MonthView.aspx?year=2006&amp;month=02"/>
        <add url="~/2005"
             mappedUrl="~/YearView.aspx?year=2005"/>
        <add url="~/2005/01"
             mappedUrl="~/MonthView.aspx?year=2005&amp;month=01"/>
        <add url="~/2005/02"
             mappedUrl="~/MonthView.aspx?year=2005&amp;month=02"/>
      </urlMappings>
      <compilation debug="true"/>
    </system.web>
  </configuration>
```

In each one of the `add` elements of Listing 1-9, the hackable URL translates into a `mappedUrl`, which is the actual address and query string sent to the page. The tilde (~) symbol represents the application directory that each page is relative to. You must use `&` in place of just the & symbol to separate parameters. We've left out entries for some months to shorten the listing.

To see the list of articles, users select the month they are interested in. Listing 1-10 shows how `MonthView.aspx` is implemented. I used the GridView control in Listing 1-10 because I needed to format the output nicely for multiple data rows and I wanted to bind it to the ObjectDataSource control. The GridView control is a new control added to ASP.NET in v2.0 that replaces the DataGrid control.

Listing 1-10: Using the GridView to read query parameters: MonthView.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="MonthView.aspx.cs"
Inherits="MonthView" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Articles for the Month</title>
</head>
<body>
    <form id="form1" runat="server">
        <h1>
            Requested Articles:</h1>
        <br />
        <asp:GridView ID="GridView1" runat="server"
            AutoGenerateColumns="False" DataSourceID="ArticlesODS">
            <Columns>
                <asp:BoundField DataField="Year"
                    HeaderText="Year" SortExpression="Year" />
                <asp:BoundField DataField="Month"
                    HeaderText="Month" SortExpression="Month" />
                <asp:BoundField DataField="Title"
                    HeaderText="Title" SortExpression="Title" />
                <asp:BoundField DataField="Content"
                    HeaderText="Content" SortExpression="Content" />
```

(continued)

Listing 1-10 *(continued)*

```
            </Columns>
        </asp:GridView>
        <asp:ObjectDataSource ID="ArticlesODS" runat="server"
            SelectMethod="GetArticles" TypeName="Articles">
            <SelectParameters>
                <asp:QueryStringParameter DefaultValue="2006"
                    Name="year" QueryStringField="year"
                    Type="String" />
                <asp:QueryStringParameter DefaultValue="01"
                    Name="month" QueryStringField="month"
                    Type="String" />
            </SelectParameters>
        </asp:ObjectDataSource>
    </form>
</body>
</html>
```

Being an advocate of well-designed n-tier architectures, I decided to implement article management with business objects and a formal data access layer. This is why you see the ObjectDataSource control in the `MonthView.aspx` page. It was handy to map query string parameters directly to the GetArticles method of the Articles class, as shown in Listing 1-11.

Listing 1-11: The Articles class contains a list of article objects: articles.cs

```
using System;
using System.Collections.Generic;

/// <summary>
/// List of Articles
/// </summary>
public class Articles : List<Article>
{
    public List<Article> GetArticles(string year, string month)
    {
        ArticleData dal = new ArticleData();
        dal.GetArticles(this, year, month);
        return this;
    }
}
```

The `Articles` class in Listing 1-11 takes advantage of the new *generics* feature of the C# programming language to create a strongly typed collection of `Article` objects. By inheriting List<Article> we get the benefits of generics with the capability to name our type something a bit more understandable. Listing 1-12 shows the data access layer, represented by the ArticleData class.

Listing 1-12: The ArticleData class populates the current list with new articles from the data source: ArticleData.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
```

```
using System.Data;
using System.Web;

/// <summary>
/// Summary description for ArticleData
/// </summary>
public class ArticleData
{
    public void GetArticles(List<Article> articles, string year, string month)
    {
        DataSet dsArticles = new DataSet();
        dsArticles.ReadXml(HttpContext.Current.Server.MapPath("Articles.xml"));
        DataView dvArticles = new DataView(dsArticles.Tables["article"]);
        dvArticles.RowFilter =
            "year = '" + year + "' " +
            "and month = '" + month + "'";

        Article currArticle = null;

        IEnumerator articleRows = dvArticles.GetEnumerator();

        while (articleRows.MoveNext())
        {
            DataRowView articleRow = (DataRowView)articleRows.Current;
            currArticle = new Article(
                (string)articleRow["year"],
                (string)articleRow["month"],
                (string)articleRow["title"],
                (string)articleRow["content"]);

            articles.Add(currArticle);
        }
    }
}
```

This class loads an XML file (see Listing 1-13) into a DataSet. It uses a DataView to filter the results based on the year and month parameters that were passed in. The Article class is the business object that will eventually be bound in the UI layer to a GridView. Of course, the UI layer doesn't need to know that the data came from an XML file or was processed via ADO.NET components. It can be easily changed later to facilitate new requirements. Fortunately, the GridView is able to display business objects in a Generic collection, which is why we pass the data back as List<Article> through the articles parameter.

Listing 1-13: Article data is represented via an XML file: Articles.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<articles>
  <article>
    <year>2005</year>
    <month>01</month>
    <title>Title1</title>
    <content>This is the text of Title1.</content>
  </article>
  <article>
```

(continued)

Listing 1-13 *(continued)*

```
      <year>2005</year>
      <month>02</month>
      <title>Title2</title>
      <content>This is the text of Title2.</content>
    </article>
    <article>
      <year>2005</year>
      <month>02</month>
      <title>Title3</title>
      <content>This is the text of Title3.</content>
    </article>
    <article>
      <year>2006</year>
      <month>01</month>
      <title>Title4</title>
      <content>This is the text of Title4.</content>
    </article>
    <article>
      <year>2006</year>
      <month>01</month>
      <title>Title5</title>
      <content>This is the text of Title5.</content>
    </article>
    <article>
      <year>2006</year>
      <month>02</month>
      <title>Title6</title>
      <content>This is the text of Title6.</content>
    </article>
  </articles>
```

The code in Listing 1-12 uses an XML file (see Listing 1-13) as the datasource for a very simple implementation. The ArticlesData class uses the data to populate Article objects, as shown in Listing 1-14.

Listing 1-14: The Article class is a business object that will be bound to a GridView in the UI layer: Article.cs

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

/// <summary>
/// Represents an Article
/// </summary>
public class Article
{
```

```
    private string m_year;

    public string Year
    {
        get { return m_year; }
        set { m_year = value; }
    }

    private string m_month;

    public string Month
    {
        get { return m_month; }
        set { m_month = value; }
    }


    private string m_title;

    public string Title
    {
        get { return m_title; }
        set { m_title = value; }
    }

    private string m_content;

    public string Content
    {
        get { return m_content; }
        set { m_content = value; }
    }

  public Article(string year, string month, string title, string content)
  {
        Year = year;
        Month = month;
        Title = title;
        Content = content;
  }
 }
```

Data in the Articles class is exposed through properties as its public interface, providing encapsulation. This gives you the ability to change the underlying implementation as necessary, including adding business rules and validation logic.

> **If I had used a datasource control other than the OjectDataSource control in my code, adding business rules would not have been an option. I chose not to for the purpose of simplifying the code in this example, but I still have a choice. Using the Object DataSource control properly, you can design applications so they are flexible and maintainable.**

Using the ASP.NET v2.0 URL Mapping feature appears to be a great capability, but there is one catch: You can't use regular expressions. The articles example in this section could have been better written using regular expressions. Notice that the `urlMappings` elements contain very similar code that only differs by year or month. This could have been implemented better with a single regular expression that mapped the year and month with parameter placement. If you used Scott Mitchell's URL rewriting engine, the configuration would look like this:

```
<RewriterConfig>
   <Rules>
      <!-- Rules for Blog Content Displayer -->
      <RewriterRule>
         <LookFor>~/(\d{4})/(\d{2})/Default\.aspx</LookFor>
         <SendTo><![CDATA[~/YearView.aspx?year=$1&month=$2]]></SendTo>
      </RewriterRule>
      <RewriterRule>
         <LookFor>~/(\d{4})/Default\.aspx</LookFor>
         <SendTo>~/YearView.aspx?year=$1</SendTo>
      </RewriterRule>
   </Rules>
</RewriterConfig>
```

This is a simple, one-time configuration. However, with the Visual Studio 2005 implementation, demonstrated by the `urlMapping` elements of the configuration file in Listing 1-9, you must continually update the file for every year and every month of every year. This is not practical because the file will continually grow and it requires manual intervention. For serious URL rewriting capabilities with regular expressions, I recommend you use the hack in Scott Mitchell's article. Use the URL mapping feature of ASP.NET v2.0 in only the simplest cases.

Wrapping Up

This chapter introduced a few hacks that were invented by pioneers of ASP.NET v1.1. These people and others had such a great influence that Microsoft turned their hacks into major product features in ASP.NET v2.0. Wizard hacks became the Wizard control. Template, inheritance, and user control Master Page hacks became Master Pages in ASP.NET v2.0. URL rewriting hacks influenced URL mapping in ASP.NET v2.0, and are still so good that the URL rewriting hacks will continue to be a preferred method of implementing URL rewriting when regular expressions are desired.