2

Working with Master Pages

It took developers little time to realize that something was missing in the ASP.NET approach to creating Web pages and Web sites. Certainly ASP.NET greatly simplified the process. However, after a few months of real-world experience, many developers recognized that they needed more effective and powerful tools to build useful Web applications with the same ease that they could build simple sites.

Almost all Web sites use a similar graphical layout for all their pages. This doesn't happen by chance—it grows out of accepted guidelines for design and usability. A consistent layout is characteristic of all cutting-edge Web sites, no matter how complex. For some Web sites, the layout consists of the header, body, and footer; for others, it is a more sophisticated aggregation of menus, buttons, and panels that contain and render the actual content.

The question is, how can you effectively build such Web sites? I wouldn't be giving you the best advice if I told you to manually duplicate your code and HTML elements. Making your code automatically reusable represents a better approach, but how do you implement it, *in practice*?

Both classic ASP and ASP.NET 1.x provide good workarounds for this type of issue, but neither tackles such a scenario openly and provides a definitive, optimal solution. ASP.NET 2.0 faces up to the task through a new technology—*master pages*—and basically exploits the ASP.NET Framework's ability to merge a "supertemplate" with user-defined content replacements.

The Rationale Behind Master Pages

With ASP.NET 1.x, you can apply a common layout to all the pages of a Web site by wrapping all the common user interface widgets in user controls and

reuse them in all the pages. With classic ASP, include files offer the best approach. As you'll see in a moment, though, neither of these approaches (either of which you could also use with ASP.NET 2.0) really hits the target.

A better way to build and reuse pages must fulfill three requirements. First, the pages have to be easy to modify. Second, changes shouldn't require deep recompilation and diffuse retouch of the source code. Third, any change must have minimal impact on the overall performance of the application.

Before we look at how ASP.NET 2.0 master pages address these requirements, let's briefly examine what is good and bad about the ASP.NET 1.x and classic ASP approaches so you'll understand how the final architecture of master pages came to be.

User Controls in ASP.NET 1.x

In ASP.NET 1.x, the best approach to authoring pages with a common layout is to employ *user controls*. User controls are aggregates of ASP.NET server controls, literal text, and procedural code. The ASP.NET runtime exposes user controls to the outside world as programmable components. The idea is that you employ user controls to tailor your own user interface components and share them among the pages of the Web site. For example, all the pages that need a navigational menu can reference and configure the user control that provides that feature.

What's Good About User Controls

User controls are like embeddable pages. Turning an existing ASP.NET page into a user control requires only a few minor changes. User controls can be easily linked to any page that needs their services. Furthermore, changes to a user control's implementation do not affect the referencing page and only require you (or the runtime) to recompile the user control into an assembly.

However, the best feature of user controls turns out to be the main drawback as well.

What's Bad About User Controls

If you change the *internal* implementation of the user control, no referencing page will be affected. However, if you alter any aspect of the control's *public* interface (such as the class name, properties, methods, or events), all the pages that reference the control must be updated. This means you must manually retouch all the pages in the application that use the control. Then you must recompile these pages and deploy the assemblies. In addition, the next time a user views each page, the ASP.NET runtime will take a while to respond because the dynamic assembly for the page must be re-created.

Architecturally speaking, the solution based on user controls works just fine. In practice, though, it is not a very manageable model for large-scale applications—its effectiveness decreases as the complexity of the application (the number of pages involved) increases. If your site contains hundreds of pages, handling common elements through user controls can quickly become inefficient and unmanageable.

This model forces you to introduce duplicate code in content pages. In fact, all pages must reference user controls and all of them must be updated (and recompiled) whenever you change something in the design of the pages or in the programming interface of the user controls. Touching hundreds of files is simply out of the question.

Another, subtler problem arises when you use user controls to build a pagewide user interface. User controls are individual components and as such should be self-contained and designed as embeddable pages. When individual components are used to build a pagewide user interface, you are likely to end up splitting HTML elements between different user controls. For example, a <*table*> element might begin in one user control and end in another one. Although this is not strictly a syntax error, it clearly indicates a less-than-optimal design.

Include Files in Classic ASP

Classic ASP offers a smaller set of tools than ASP.NET 1.x, but the best practice that emerged after years of real-world experience with classic ASP is philosophically closer to ASP.NET 2.0 master pages than user controls. With classic ASP, developers reuse common user interface widgets (such as a header, a footer, or menus) by wrapping them in external include files. When the ASP runtime builds the response for the browser, the content stored in include files is merged with the main template of the ASP page.

What's Good About Include Files

A typical include file contains the HTML markup for the portion of the page it represents. An include file can contain either static or dynamic content. Changes to any include files are reflected in the final page but don't affect how the final page is built. No performance hit stems from changes to included files; this is due to the different runtime architecture of classic ASP compared to ASP.NET.

The page served to the browser is constructed by importing external content into the main template of the page. However, each page of the application remains an independent object and is considered the root of a small tree whose leaves are the include files. In other words, there are no points of contact

between the various pages that share a common layout. ASP.NET 2.0 master pages—supertemplates common to all pages sharing a given layout—are simply an enhancement to this approach.

What's Bad About Include Files

Include files are plain containers of relatively static text, and they are merely a cache of HTML markup that is used throughout the application. The markup is integrated with the existing skeleton of the page; it is typically placed in table cells and rows.

This approach has two main drawbacks. First, there's no object orientation, so integrating this approach with the ASP.NET programming model is hard. Second, include files tags opened in one file are frequently closed in another file (either the .asp page or another include file). This situation makes WYSIWYG designer support virtually impossible.

Outline of a Better Approach

The following aspects of master pages make page layout easy to share, simple to maintain, and functional to the application:

- Definition of a supertemplate—the master page—that individual content pages refer to explicitly
- Attachment of a master at various levels in the application scheme—all pages in the Web space, all pages in a directory, and each individual page
- Support for multiple master pages per Web application
- WYSIWYG support for both master pages and content pages

The master page is a single file that defines the template for a set of pages. Similar to an ordinary .aspx page, the master contains replaceable sections that are each marked with a unique ID. Pages in an application that will inherit the structure defined in the master will reference the master page in their *@Page* directive or programmatically. A page based on a master is called a *content page*. One master page can be bound to any number of content pages.

Master pages are completely transparent to end users. When working with an application, a user sees and invokes only the URL of content pages. If a content page is requested, the ASP.NET runtime applies a different compilation algorithm and builds the dynamic class as the merge of the master and the content page. Figure 2-1 illustrates the overall scheme.

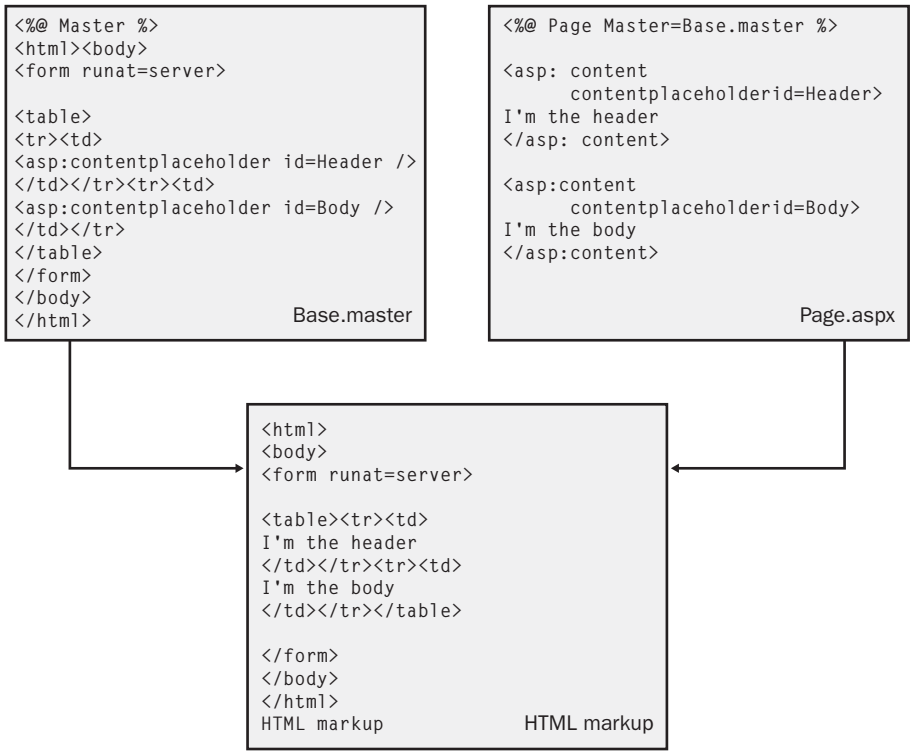

Figure 2-1 The master page includes one or more content placeholders that define regions where replaceable content will appear.

Master pages are among the hottest new features of ASP.NET 2.0 and address one of the hottest threads in many ASP.NET 1.x newsgroups. By using master pages, a developer can create a Web site in which various physical pages share a common layout. You code the shared user interface and functionality in the master page. The master also contains named placeholders for content that the derived page will provide. The shared information is stored in a single place—the master page—instead of being replicated in each page. Since the master page is a Web page, any pagewide construct begins and ends within the same context. Each content page references the master and fleshes out the placeholders with the actual content. The presence of a master page is not revealed on the client side, and no master file is ever downloaded.

The master page is to some extent similar to ASP.NET templated controls, in which the templates are content pages and the outer markup of the control is the master.

> **Note** Master pages in ASP.NET 2.0 offer one way of building Web pages—not the only way or even the preferred way. You should use master pages only if you're using user controls extensively to duplicate portions of your user interface or if your application lends itself to being (re)designed in terms of master and content pages.

What Are Master Pages?

To build Web pages based on a master, you start by creating the master. A master page is a file with a .master extension. The syntax of a master page is not much different from that of a regular .aspx page. A master page has two key characteristics:

- A new *@Master* directive replacing the *@Page* directive
- One or more *ContentPlaceHolder* child controls

Each embedded *ContentPlaceHolder* control identifies a region of markup text whose real content is provided at run time by content pages. The body of a master page can contain any combination of server controls, literal text, images, HTML elements, and managed code. All this, plus the content bound to placeholders, originates the virtual .aspx source from which the dynamic page class for the user is generated. (See Figure 2-1.)

Master pages work in conjunction with content pages. In fact, neither master pages nor content pages are of any use if used independently. If you attempt to request a .master resource, you'll see an error message because .master represents a forbidden type of resource in ASP.NET. (See Figure 2-2.)

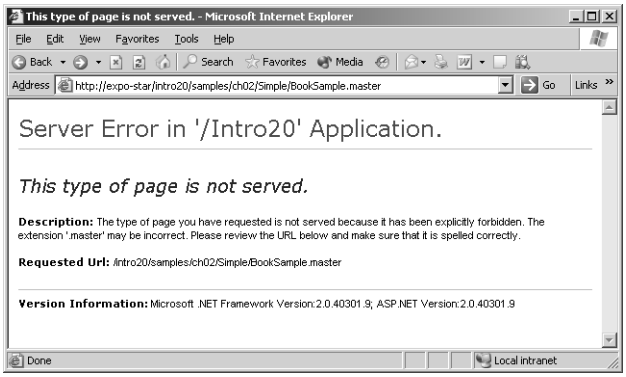

Figure 2-2 Requests for .master resources are received and rejected by the ASP.NET runtime.

Likewise, you see an error message if you request an .aspx resource that's built as a content page but isn't bound to an existing master page. (You'll learn more about content pages in a moment.)

We'll write a sample master/content pair of pages to see how the whole mechanism works and what syntactical elements are involved. Then we'll consider more advanced issues and build a more realistic example of templated pages.

Writing a Master Page

As mentioned, a master page is similar to an ordinary .aspx page except for the top *@Master* directive and the presence of one or more *ContentPlaceHolder* server controls. A master page without content placeholders is technically correct and will be processed correctly by the ASP.NET runtime. However, a placeholderless master fails in its primary goal—to be the supertemplate of multiple pages that look alike. A master page devoid of placeholders works like an ordinary Web page but with the extra burden required to process master pages.

Here's a simple master page named booksample.master:

```
<%@ Master %>
<html>
<head>
    <link rel="Stylesheet" href="styles.css" />
</head>
<form runat="server">

<table border="0" width="100%" bgcolor="beige"
       style="BORDER-BOTTOM:silver 5px solid">
  <tr>
     <td><h2>Introducing ASP.NET 2.0</h2></td>
  </tr>
</table>
<br>
<asp:contentplaceholder runat="server" id="PageBody" />
</form>
</body>
</html>
```

As you can see, the master page looks like a standard ASP.NET page. Aside from the identifying *@Master* directive, the only key difference is the *ContentPlaceHolder* control. A page bound to this master automatically inherits all the contents of the master and has a chance to attach custom markup and server controls to each defined placeholder.

The content placeholder element is fully identified by its *ID* property and normally doesn't require other attributes.

The *@Master* Directive

The *@Master* directive distinguishes master pages from content pages and allows the ASP.NET runtime to properly handle each. The *@Master* directive supports quite a few attributes, including *Language*, *Debug*, *Inherits*, and *ClassName*. These attributes play the same role that they do for ordinary .aspx pages. The *Language* attribute specifies the language used in the master. *Inherits* specifies the base class for the current master, and *ClassName* specifies the name of the actual master class. Finally, *Debug* ensures that debug symbols are added to the compiled page, and it causes the ASP.NET runtime to persist any temporary files (including the source code of the dynamic page class) created during the processing of the page request.

The attributes supported by a *@Master* directive are also the same as those defined on the *@Control* directive for user controls. This is not coincidental. A master page file is compiled to a class that derives from the *MasterPage* class. The *MasterPage* class, in turn, inherits *UserControl*. So, at the end of the day, a master page is treated as a special kind of ASP.NET user control.

> **Note** You can also create master pages programmatically. You build your own class and make it inherit *MasterPage*. Then you create .master files in which the *Inherits* attribute points to the fully qualified name of your class. Rapid application development (RAD) designers such as the one embedded in Visual Studio 2005 use this approach to create master pages.

Table 2-1 details the attributes of the *@Master* directive.

Table 2-1 Attributes of the *@Master* Directive

Attribute	Description
AutoEventWireup	Specifies whether the master page's events are bound to methods with a particular name. The default is *true*.
ClassName	Specifies the name for the class that will be created to render the master page. This value can be any valid class name but should not include a namespace.

Table 2-1 Attributes of the *@Master* Directive

Attribute	Description
CompilerOptions	Specifies a sequence of compiler command-line switches used to compile the master class. The target compiler depends on the language of choice.
Debug	Specifies whether the master page will be compiled with debug symbols. If *true*, the source code of the master will not be deleted and can be retrieved under the Temporary ASP.NET Files folder.
Description	Provides a text description of the master page.
EnableViewState	Specifies whether view state for the controls in the master page is maintained across page requests. The default is *true*.
EnableTheming	Specifies whether themes for the controls in the master page are enabled. The default is *true*.
Explicit	Specifies whether the master page will be compiled using the Visual Basic *Option Explicit* mode. This attribute is ignored by languages other than Visual Basic .NET. It is *false* by default.
Inherits	Specifies a code-behind class for the master page to inherit. This can be any class derived from *MasterPage*.
Language	Specifies the language used throughout the master page.
MasterPageFile	Specifies the name of the master page file that this master refers to. A master can refer to another master through the same mechanisms a page uses to attach to a master. If this attribute is set, you will have nested masters.
Strict	Specifies whether the master page will be compiled using the Visual Basic *Option Strict* mode. This attribute is ignored by languages other than Visual Basic .NET. It is *false* by default.
Src	Specifies the source filename of the code-behind class to dynamically compile when the master page is requested.
WarningLevel	Specifies the compiler warning level at which the compiler will abort compilation of the master page.

Note that the *@Master* directive doesn't override attributes set at the *@Page* directive level. For example, you can have the master set the language to Visual Basic .NET, and one of the content pages can use C#. The language set at the master page level never influences the choice of the language at the content page level.

> **Note** You can use other ASP.NET directives in a master page—for example, *@Import*. However, the scope of these directives is limited to the master file and does not extend to child pages generated from the master. For example, if you import the *System.Data* namespace into a master page, you can call the *DataSet* class within the master. But to call the *DataSet* class from within a content page, you must also import the namespace into the content page.

The *ContentPlaceHolder* Container Control

The *ContentPlaceHolder* control inherits from the *Template* class and is defined in the *System.Web.UI.WebControls* namespace. It acts as a container placed in a master page. It marks places in the master where related pages can insert custom content. A content placeholder is uniquely identified by an ID. Here's an example:

```
<asp:contentplaceholder runat="server" ID="PageBody" />
```

A content page is an .aspx page that contains only *<asp:Content>* server tags. This element corresponds to an instance of the *Content* class that provides the actual content for a particular placeholder in the master. The link between placeholders and content is established through the ID of the placeholder. The content of a particular instance of the *Content* server control is written to the placeholder whose ID matches the value of the *ContentPlaceHolderID* property, as shown here:

```
<asp:Content runat="server" contentplaceholderID="PageBody">
   ⋮
</asp:Content>
```

In a master page, you define as many content placeholders as there are customizable regions in the page. A content page doesn't have to fill all the placeholders defined in the bound master. However, a content page can't do more than just fill placeholders defined in the master.

> **Note** A placeholder can't be bound to more than one content region in a single content page. If you have multiple *<asp:Content>* server tags in a content page, each must point to a distinct placeholder in the master.

Specifying Default Content

A content placeholder can be assigned default content that will show up if the content page fails to provide a replacement. Each *ContentPlaceHolder* control in the master page can contain default content. If a content page does not reference a given placeholder in the master, the default content will be used. The following code snippet shows how to define default content:

```
<asp:contentplaceholder runat="server" ID="PageBody">
   <!-- Use the following markup if no custom
          content is provided by the content page -->
   ⋮
</asp:contentplaceholder>
```

The default content is completely ignored if the content page populates the placeholder. The default content is never merged with the custom markup provided by the content page.

> **Note** A *ContentPlaceHolder* control can be used only in a master page or a (templated) user control. Content placeholders are not valid on .aspx pages. If such a control is found in an ordinary Web page, a parser error occurs.

Writing a Content Page

The master page defines the skeleton of the resulting page. If you need to share layout or a navigational menu among all the pages, placing it in a master page will greatly simplify management of the pages in the application. You create the master and then think of your pages in terms of a delta from the master. The master defines the common parts of a certain group of pages and leaves placeholders for customizable regions. Each content page, in turn, defines what the content of each region has to be for a particular .aspx page.

The *Content* Control

The key part of a content page is the *Content* control. The class is defined in the *System.Web.UI.WebControls* namespace and inherits *Control*. A *Content* control is a container for other controls placed in a content page. The control is used only in conjunction with a corresponding *ContentPlaceHolder* and is not a standalone control.

The master file that we considered earlier defines a single placeholder named *PageBody*. This placeholder represents the body of the page and is

placed right below an HTML table that provides the page's header. The following listing shows a sample content page bound to the booksample.master file:

```
<%@ Page Language="C#" masterpagefile="booksample.master" %>

<script runat="server">
    void OnButtonClick(object sender, EventArgs e)
    {
        msg.Text = "Hello, Master Pages";
    }
</script>

<asp:content runat="server" contentplaceholderID="PageBody">
<div>
   <h1>This is the body of the page</h1>
   <asp:button runat="server" text="Click Me" onclick="OnButtonClick" />
   <asp:label runat="server" id="msg" />
</div>
</asp:content>
```

The content page is the resource that users invoke through the browser. Let's call it withmaster.aspx. When the user points her browser to this page, the output in Figure 2-3 is shown.

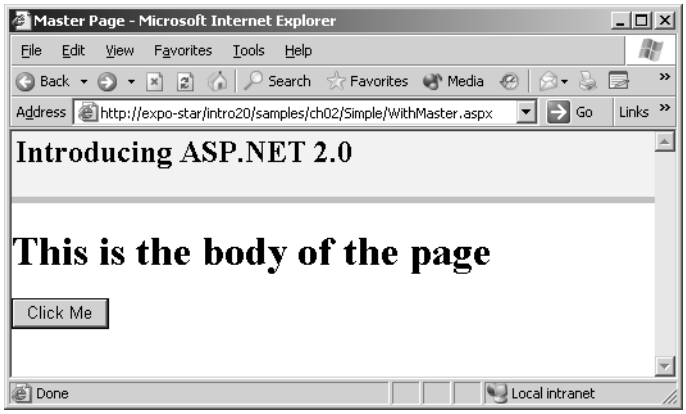

Figure 2-3 The page named withmaster.aspx is obtained by merging the master and the content page.

The replaceable part of the master is filled with the corresponding content section defined in the derived pages. In the previous example, the *<asp:Content>* section for the *PageBody* placeholder contains a button and a label. The server-side code associated to the button is defined in the content page. You should notice that the *Language* attribute points to a different language in the *@Master* directive than it does in the *@Page* directive. In spite of this, the page is created and displayed correctly.

Attaching Pages to a Master

In the previous example, the content page is bound to the master using the new *MasterPageFile* attribute in the *@Page* directive. The attribute points to a string representing the path to the master page. Page-level binding is just one possibility—the most common one.

You can also set the binding between the master and the content at the application or folder level. Application-level binding means that you link all the pages of an application to the same master. You configure this behavior by setting the *Master* attribute in the *<pages>* element of the principal web.config file:

```
<configuration>
   <system.web>
       <pages masterpagefile="WwwContosoCom.master" />
   </system.web>
</configuration>
```

If the same setting is expressed in a child web.config file—a web.config file stored in a site subdirectory—all ASP.NET pages in the folder are bound to a specified master page.

Note that if you use this approach, all the Web pages in the application must have *Content* controls mapped to one or more placeholders in the master page. Application-level binding prevents you from having (or later adding) a page to the site that is not configured as a content page. Any page that contains server controls will throw an exception. A page that is bound to a master is not permitted to host server controls outside of an *<asp:Content>* tag.

Device-Specific Masters

Like all ASP.NET pages and controls, master pages can detect the capabilities of the underlying browser and adapt their output to the specific device in use. The great news about ASP.NET 2.0 is that it makes choosing a device-specific master easier than ever. If you want to control how certain pages of your site appear on a particular browser, you can build them from a common master and design the master to address the specific features of the browser. In other words, you can create multiple versions of the same master, each targeting a different type of browser.

How do you associate a particular version of the master and a particular browser? In the content page, you define multiple bindings using the same *MasterPageFile* attribute, but prefixed with the identifier of the device. For example, suppose you want to provide ad hoc support for Microsoft Internet Explorer and Netscape browsers and use a generic master for any other browsers that users employ to visit the site. You use the following syntax:

```
<%@ Page ie:masterpagefile="ieBase.master"
    netscape:masterpagefile="nsBase.master"
    masterpagefile="Base.master" %>
```

The ieBase.master file will be used for Internet Explorer; the nsBase.master will be used in contrast if the browser belongs to the Netscape family. In any other case, a device-independent master (Base.master) will be used. When the page runs, the ASP.NET runtime automatically determines what browser or device the user is using and selects the corresponding master page.

The prefixes you can use to indicate a particular type of browser are those defined in the *<browserCaps>* section of the machine.config file. It goes without saying that you can distinguish not just between uplevel and downlevel browsers but also between browsers and other devices such as cellular phones and personal digital assistants (PDAs). If you use device-specific masters, you must also indicate a device-independent master.

Underpinnings of Master Pages

The use of master pages slightly changes how pages are processed and compiled. For one thing, a page based on a master has a double dependency—on the .aspx source file (the content page) and on the .master file (the master page). If either of these pages changes, the dynamic page assembly will be recreated. Although the URL that users need is the URL of the content page, the page served to the browser results from the master page fleshed out with any replacement provided by the content page. Let's see in a bit more detail how a content page merges with the master page.

Merging Master and Content Pages

When the user requests an .aspx resource mapped to a content page—that is, a page that references a master—the ASP.NET runtime begins its job by tracking the dependency between the source .aspx file and its master. This information is persisted in a local file created in the ASP.NET temporary files folder. Next, the runtime parses the master page source code and creates a Visual Basic .NET or C# class, depending on the language set in the master page. In the previous example, the booksample.master master page is parsed to a Visual Basic .NET class. (If the *Language* attribute is not specified, Visual Basic .NET is assumed.) The class inherits *MasterPage* and is then compiled to an assembly.

The *MasterPage* Class

The *MasterPage* class is pretty simple—just a small wrapper built around the *UserControl* class:

```
public class MasterPage : UserControl
{
}
```

The dynamic class that the ASP.NET runtime builds from the master page source code extends *MasterPage* by adding any public members defined in line. For example, it adds new properties, methods, and events. In addition, a few protected and private members are added by the framework. In particular, a protected member is added for each *ContentPlaceHolder* server tag found in the .master source. The name of the property matches the ID of the server tag in the source file. Based on the aforementioned simple.master file, the protected property looks like the following snippet:

```
Protected PageBody As System.Web.UI.WebControls.ContentPlaceHolder
```

In addition, a template member is added to represent the content dynamically bound to the placeholder. The property is of type *ITemplate* and is set with actual content provided by the content page for that placeholder.

The overall structure of the source code extracted out of a master page is not much different from that of a classic ASP.NET page. In ASP.NET pages, for each control marked with a *runat* attribute, the runtime generates the code to instantiate and configure the corresponding class. The same occurs with the *ContentPlaceHolder* class; it is instantiated, named, and bound to the matching property on the master page class—the *PageBody* property set above. The final step in this procedure is the instantiation of the template within the placeholder control:

```
// __ctrl is the placeholder control
// Template_PageBody is the internal template member
// representing the dynamically set content of the placeholder
Template_PageBody.InstantiateIn(__ctrl);
```

The templated property is defined but not assigned any value in the master page class. The template is populated while the content page is processed.

> **Note** If multiple .master files are found in the same directory, they are all processed at the same time. Thus a dynamic assembly is generated for any master files found, even if only one of them is used by the ASP.NET page whose request triggered the compilation process. Therefore, don't leave unused master files in your Web space—they will be compiled anyway. Also note that the compilation tax is paid only the first time a content page is accessed within the application. When a user accesses another page that requires the second master, the response is faster because the master is precompiled.

Importing Compiled Templates

Any ASP.NET page bound to a master page must have a certain structure—no server controls or literal text are allowed outside the <*asp:Content*> tag. As a result, the layout of the page looks like a plain collection of content elements, each bound to a particular placeholder in the master. The connection is established through the ID property.

The <*asp:Content*> element works like a control container, much like the *Panel* control of ASP.NET or the HTML <*div*> tag. All the markup text is compiled to a template and associated with the corresponding placeholder property on the master class.

The master page is a special kind of user control. In fact, the ASP.NET Framework calls the *InitializeAsUserControl* method—an internal method on the *UserControl* class—which completes the initialization phase of user controls. The method wires automatic event handlers (such as *Page_Load*, *Page_Unload*) to the control.

The construction of the final page continues with the addition of the filled master page to the control tree of the current instance of the page. No other controls are present in the final page except those brought in by the master. Figure 2-4 shows the skeleton of the final page served to the user.

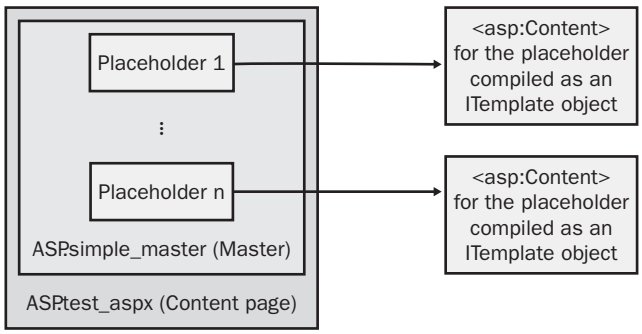

Figure 2-4 The structure of the final page in which the master page and the content page are merged

Nested Masters

So far we've seen a pretty simple relationship between a master and a collection of content pages. However, the topology of the relationship can be made as complex and sophisticated as needed. A master can, in fact, be associated with another master and form a hierarchical, nested structure. Figure 2-5 shows an example.

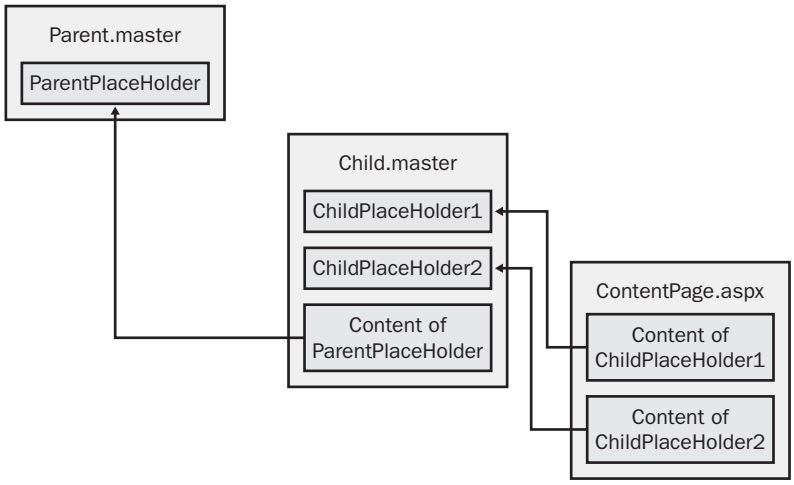

Figure 2-5 Complex sites (such as portals) require the use of interrelated master pages that end up forming a hierarchical structure.

Designing Nested Masters

When nested masters are used, any child master is seen and implemented as a plain content page in which extra *ContentPlaceHolder* controls are defined for an extra level of content pages. Put another way, a child master is a content page that contains a combination of *<asp:Content>* and *<asp:ContentPlaceHolder>* elements. Like any other content page, a child master points to a master page and provides content blocks for its parent's placeholders. At the same time, it makes available new placeholders for its child pages.

There's no architectural limitation in the number of nesting levels you can implement in your Web sites. Performance-wise, the depth of the nesting has a negligible impact on the overall functionality and scalability of the solution. The final page served to the user is always compiled on demand and never modified as long as dependent files are not touched.

Let's extend the previous example so it encompasses a nested structure of master pages.

Building an Example

Many Web sites—most of them portals—have such a complex structure that they can't be rendered using a flat master/content relationship. Suppose you need to have a global menu on top of the page and then a second-level menu whose options vary quite a bit depending on which logical group the page belongs to. In addition, suppose that the second-level menu is not the only visual element to be displayed—a search box or a login text box is also required.

Even from this simple description, it's clear that the portal contains two distinct sets of widgets. Hence, two distinct but interrelated masters are needed to render it in code.

The parent master (BookSample.master) defines the overall layout of the pages—header, body, and footer. The child master (Body.master) expands the body for a certain group of pages, meaning that the Web site will be made of pages belonging to different groups, each with a differently laid out structure. We define a child master in which a toolbar is expected. The content page is responsible for providing the buttons for the toolbar. We'll use a slightly modified version of the BookSample.master page we considered earlier as the parent master in this example. Here's the code:

```
<%@ Master Language="C#" %>
<html>
<head>
    <link rel="Stylesheet" href="/intro20/styles.css" />
    <title>Master Page</title>
</head>
<body style="margin:0;font-family:verdana;">
<form runat="server">
<table width="100%" bgcolor="beige" style="BORDER-BOTTOM:silver 5px solid">
<tr>
    <td><h1>Introducing ASP.NET 2.0</h1></td>
</tr>
</table>
<br>
<table width="100%" style="border:solid 1px black;">
<tr><td>
   <asp:contentplaceholder runat="server" id="Toolbar" /></td></tr>
<tr><td>
   <asp:contentplaceholder runat="server" id="PageBody" /></td></tr>
<tr><td align="center" style="background-color:lightcyan;">
   All rights reserved.</td></tr>
</table>
</form>
</body>
</html>
```

The following code shows the source of the child master—a file named Body.master:

```
<%@ Master Language="VB" MasterPageFile="BookSample.master" %>
<asp:content runat="server" contentplaceholderID="Toolbar">
<table width="100%">
    <tr bgcolor="lightcyan">
       <td width="100%"><h3>Great choice!</h3></td>
```

```
    </tr>
    <tr>
      <td width="100%" style="text-align:center">
        <asp:contentplaceholder runat="server" id="Menu" />
      </td>
    </tr>
</table>
</asp:content>

<asp:content runat="server" contentplaceholderID="PageBody">
<table>
    <tr>
    <td><img src="/Intro20/Images/cover.jpg" align="right"></td>
    <td>
      <h1>Introducing ASP.NET 2.0</h1>
      <h2>Dino Esposito</h2>
      <h2>Microsoft Press, 2004</h2>
      <h2><a href="http://www.microsoft.com/mspress/books/6962.asp">
          Click to learn more</a></h2>
    </td>
    </tr>
</table>
</asp:content>
```

The *@Master* directive contains a new attribute, *MasterPageFile*, that specifies the master page this page is related to. The child master is two things at once. It is a content page with respect to the parent master (and, in fact, it contains a collection of *<asp:Content>* regions). At the same time, it is a master with respect to other content pages in that it features one or more content placeholders—for example, *Menu*.

The following code illustrates a sample content page that originates from the two nested masters. Figure 2-6 shows the final page in the browser.

```
<%@ Page language="C#" masterpagefile="Body.master" %>
<script runat="server">
    void OnBuy(object sender, EventArgs e)
    {…}
    void OnReview(object sender, EventArgs e)
    {…}
    void OnView(object sender, EventArgs e)
    {…}
</script>

<asp:content runat="server" contentplaceholderID="Menu">
<div>
    <asp:button runat="server" text="View TOC"
```

```
            onclick="OnView" width="90px" />
           
        <asp:button runat="server" text="Buy"
            onclick="OnBuy" width="90px" />
        <asp:button runat="server" text="Review"
            onclick="OnReview" width="90px" />
</div>
</asp:content>
```

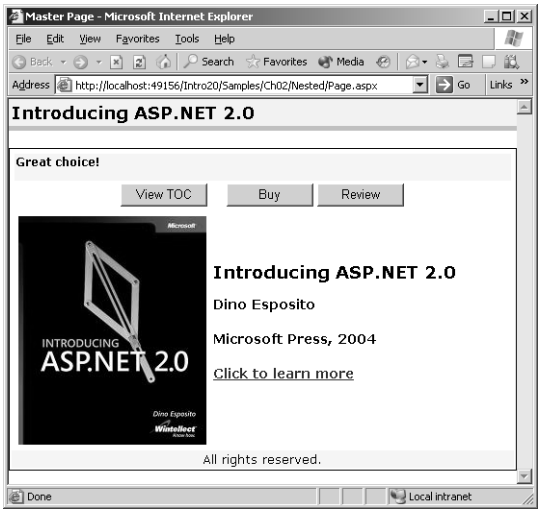

Figure 2-6 The page shown in the browser was created using two interrelated masters—one for the main structure (header, footer, toolbar, and body) and one for the details (subtitle and menu) of the toolbar.

Event Handling

As mentioned earlier, you can configure master pages and content pages to use different languages. In the previous example, in fact, parent and child masters use different languages. But what are languages for? As in ordinary ASP.NET pages, the *Language* attribute specifies the language used to write any procedural code bound to the page. The code can be defined in line or placed in an external file. It typically contains event handlers or public and private methods.

Both master pages and content pages can contain event handlers for embedded controls. Events fired by controls defined in the content page can't be handled at the master-page level. Likewise, controls defined in a master page appear inaccessible to a content page and related events can't be handled.

Page-level events, on the other hand, can be hooked in both the master and the content page. If the same page-level event is handled by both master and content page, handlers are invoked in order, according to the rules set by the page/user control relationship. Once the replaceable regions of a master page are filled, the generated page is hosted within the final page as a user control. (That's why the *MasterPage* class inherits *UserControl*.)

For example, let's consider the order in which the *Page_Init* and *Page_Load* events are captured. Bear in mind that default events are automatically wired up for master pages. If handlers exist in the master and the content page, the order in which they fire will depend on the particular event and will be the same order you'd expect with user controls. As Figure 2-7 shows, the *Init* event reaches the master page before the content page does; for the *Load* event, the order is reversed and the content page is hit first.

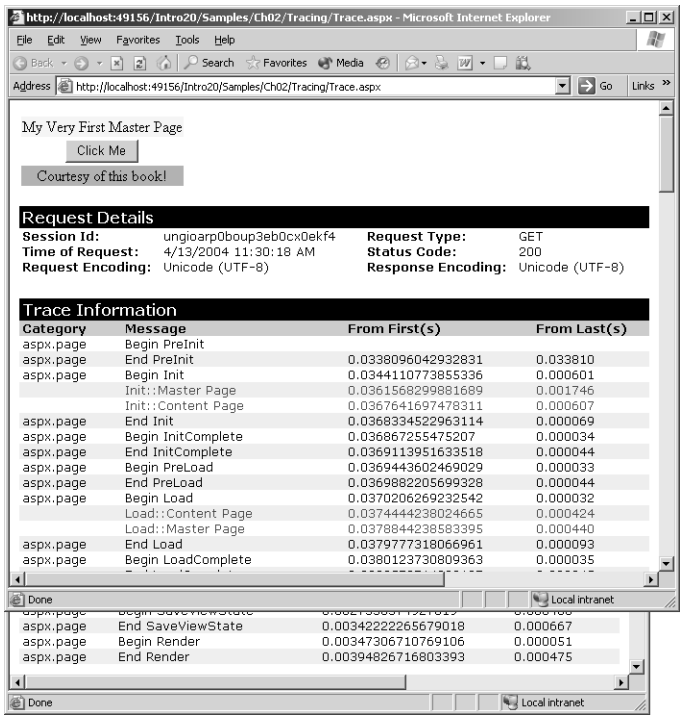

Figure 2-7 Event handlers can be defined in the master page as well as in the content page.

A Realistic Example

Master pages are an incredibly powerful technology that fulfill an important requirement of ASP.NET developers—building similar-looking and similar-working pages quickly and effectively. Former users of the .NET Framework 1.x know about Windows Forms visual inheritance. In brief, it is a Visual Studio .NET feature that allows developers to build new forms from existing ones. There's no magic behind this feature—only pure class inheritance. So why is this feature unavailable to ASP.NET programmers?

I admit that my first guess was laziness or time constraints on the part of the ASP.NET team. The truth is a bit more complex. Master pages are the closest you can get in ASP.NET to visual inheritance *à la* Windows Forms. You'll soon see, though, that they are an equivalent solution.

Master Pages and Visual Inheritance

Let's say it up front. True visual inheritance *à la* Windows Forms is not a goal of ASP.NET 2.0 master pages. Period. The contents of a master page are merged into the content page and dynamically produce a new page class served to the user upon request. The merge process takes place at compile time and only once. In no way do the contents of the master serve as a base class for the content page.

In Windows Forms applications, you can take an existing form—a class derived from *Form*—and use it as the basis for all the new forms you're building. As a result, your new forms inherit any control and any logic built into the base forms. With master and content pages, the final result is pretty much the same, but the underlying mechanism is more complex. In both cases, though, you take a base entity—an existing form or a master page—and build a new entity with the same aspect and capabilities. So why did the ASP.NET team go for master pages instead of true inheritance?

Only a member of the team can provide a definitive answer. However, we can make a few guesses. The team clearly wanted to provide a form of inheritance for Web developers. They also wanted designers (such as the one in Visual Studio 2005) to be able to use such a powerful feature easily. Ideally, a designer would show the master layout when editing a content page and would also restrict users from editing outside the content placeholders. Keeping master and content clearly separate helps to achieve this. Figure 2-8 shows how to edit master pages in Visual Studio.

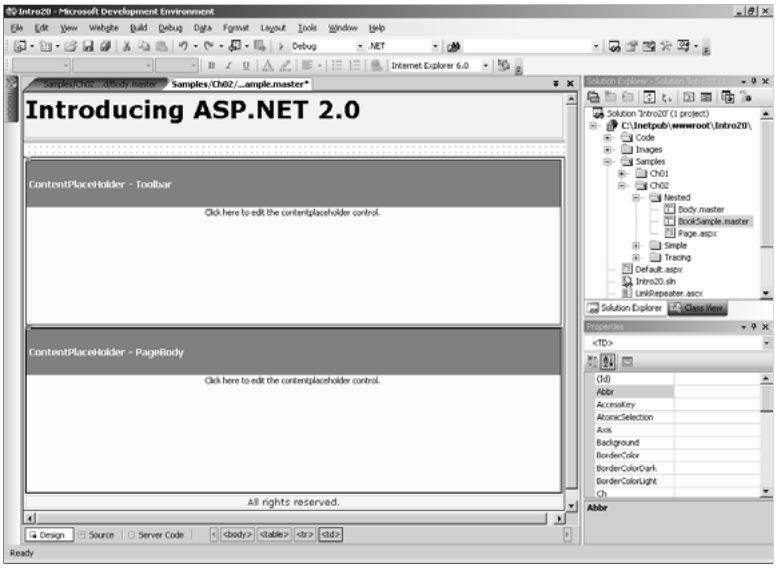

Figure 2-8 A sample master page as displayed by the Visual Studio 2005 designer

Figure 2-9 shows the look and feel of a content page in Visual Studio.

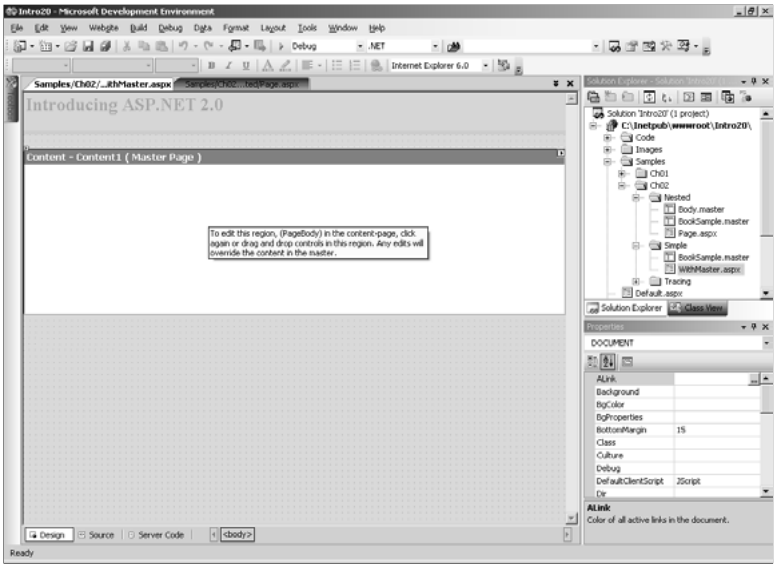

Figure 2-9 A sample content page as displayed by the Visual Studio 2005 designer

The compilation mechanism behind Web pages is too complex to enable a pure inheritance-based approach. A Web form is formed by two indissoluble elements—markup and procedural code. In theory, you could build Web pages using only C# or Visual Basic .NET code. In this case, simply inheriting your Web form from that class instead of *Page* would do the trick—just like in Windows Forms. However, in practice that would force many ASP.NET developers into an unnatural way of working and, more important, would make Web Form editing quite hard. Visual Studio .NET is built to parse markup code; nonempty base pages (those different from *Page*) would either need the ability to convert controls back to markup or a totally different WYSIWYG editor.

In Windows Forms, this works for the simple reason that no markup is involved. A button is a button with a fixed location and attributes. The Visual Studio .NET editor simply renders each control at its own position, using the persisted attributes. New controls are added (or removed) in the same direct way. Simply put, the markup-based nature of server pages (inherited from classic ASP) prevents true visual inheritance in ASP.NET.

Master pages provide the same function—forms inheritance—but by using a totally different set of tools.

> **Note** Visual Studio 2005 doesn't support visual editing of pages with nested masters. If you have two or more nested master pages, neither the child masters nor the content pages can be visually edited. You get an error message that suggests switching to the source view and entering changes codewise.

Layout of the Pages

Figure 2-10 shows a sample Web page based on a master. The master defines five insertion points—that is, content placeholders for developers to add custom code and markup. The figure includes indicators that identify the replaceable regions of the page.

Each content page based on this master would fill one or more regions with custom markup code and any procedural code that is needed to achieve the desired result. The following code shows the master on which the page in Figure 2-10 is based.

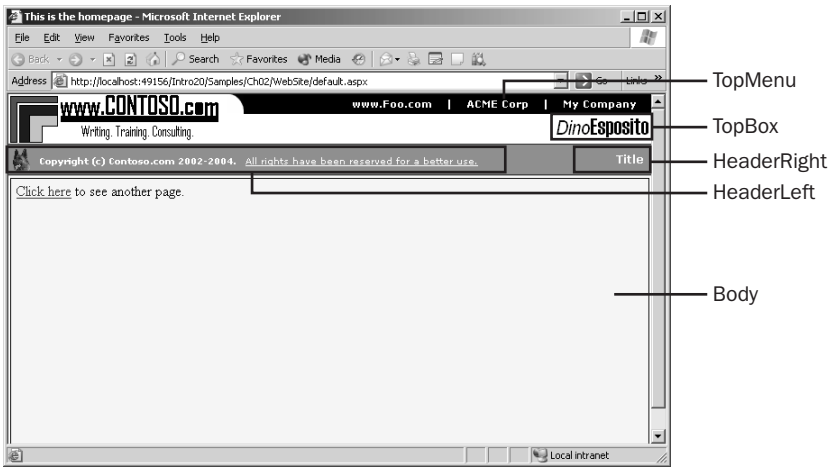

Figure 2-10 A sample Web page based on a master

```
<%@ Master Language="C#" %>
<html>
  <head runat="server">
    <title>Homepage</title>
    <link rel="Stylesheet" href="websitestyles.css" />
  </head>

<body runat="server" style="margin:0">
<form id="Main" runat="server">
  <table cellspacing="0" cellpadding="0" border="0" width="100%">
   <tr>
      <td valign="Top" rowspan="2" style="width:1px;">
        <img src="images/contoso.gif" /></td>
      <td align="Right" valign="Top">
        <img src="images/curve.gif" /></td>
      <td align="Right" class="topMenu">
       <asp:ContentPlaceHolder runat="server" ID="TopMenu" /></td></tr>
   <tr>
      <td align="Right" valign="Top" colspan="3">
       <asp:ContentPlaceHolder runat="server" ID="TopBox" /></td></tr>
  </table>

  <table cellspacing="2" cellpadding="2" border="0" class="headerStrip">
   <tr>
      <td align="Left">
        <asp:ContentPlaceHolder runat="server" ID="HeaderLeft" /></td>
```

```
      <td align="Right">
         <b><span runat="server" id="__titleBar">Title</span> </b>
      </td>
  </tr>
</table>

<div style="margin:2;border:solid 1px black;">
   <asp:ContentPlaceHolder runat="server" ID="Body" />
</div>

</form>
</body>
</html>
```

The master defines the body as well as the single form element of each ASP.NET page. Note that if both the master and the content page define a server-side <*form*> tag, an exception is raised because the final page would end up having multiple forms. The structure of the page is based on a couple of tables—one representing the topmost bar with the menu and one representing the header strip divided in two parts—left (replaceable through a placeholder) and right (modifiable through a property). The topmost table presents two replaceable regions. One is the menu with shortcuts to frequently visited sites; the other is content in the bottom-right corner. This latter area is filled with a bitmap in the sample page shown in Figure 2-10.

Contents of the Pages

The default.aspx page shown earlier in Figure 2-10 replaces the *TopMenu* placeholder with a list of hyperlinks; the *TopBox* placeholder is filled with a bitmap. The header contains two HTML strings with different text alignments. The body of the page can be anything that is appropriate for the application. The following listing shows the source code of the default.aspx content page:

```
<%@ Page Language="C#" MasterPageFile="main.master" %>

<asp:Content runat="server" contentplaceholderID="TopMenu">
   <a href="http://www.foo.com" class="topMenuItem">www.Foo.com</a>
         |   
   <a href="http://www.acme.com" class="topMenuItem">ACME Corp</a>
         |   
   <a href="http://www.mycompany.com" class="topMenuItem">My Company</a>
         
</asp:Content>

<asp:Content runat="server" contentplaceholderID="TopBox">
    <img src="images/dinologo.gif" />
</asp:Content>
```

```
<asp:Content runat="server" contentplaceholderID="HeaderStripLeft">
    <img src="images/jerry.gif"
         alt="Our beloved VP"
         align="absmiddle" width="20" height="25" /> 
    <b>Copyright (c) Contoso.com 2002-2004.  </b>
    <u>All rights have been reserved for a better use.</u>
</asp:Content>

<asp:Content runat="server" contentplaceholderID="Body">
    <asp:panel runat="server" height="300px" backcolor="lightcyan">
        <a href="another.aspx">Click here </a> to see another page.
    </asp:panel>
</asp:Content>
```

Here's the code for a different page designed using the same schema, another.aspx:

```
<%@ Page Language="C#" MasterPageFile="main.master" %>

<asp:Content runat="server" contentplaceholderID="TopMenu">
    <a href="http://mspress.microsoft.com" class="topMenuItem">
    Microsoft Press</a>
          
</asp:Content>

<asp:Content runat="server" contentplaceholderID="TopBox">
    <span class="Normal">Search: </span>
    <asp:textbox runat="server" class="textbox" ID="searchBox" />
    <asp:button runat="server" class="button" Text="Go" />
</asp:Content>

<asp:Content runat="server" contentplaceholderID="HeaderLeft">
    <img src="images/jerry.gif"
         alt="Our beloved VP"
         align="absmiddle" width="20" height="25" /> 
    <b>Copyright (c) Contoso.com 2002-2004.  </b>
    <u>All rights have been reserved for a better use.</u>
</asp:Content>

<asp:Content runat="server" contentplaceholderID="Body">
    <asp:panel runat="server" Height="300px" BackColor="lightcyan">
        <a href="default.aspx">Click here </a> for the home page.
    </asp:panel>
</asp:Content>
```

The new page (shown in Figure 2-11) has a shorter list of hyperlinked Web sites and a different panel below the menu. It also has a search box instead of a bitmap.

Figure 2-11 Another page based on the same master

Pages built on the same master will differ from each other based on what you add in the content sections. The amount of code (and markup) you need to write is comparable to that needed to set up a derived class in a truly object-oriented application.

Programming the Master

You can use code in content pages to reference properties, methods, and controls in the master page, with some restrictions. The rule for properties and methods is that you can reference them if they are declared as public members of the master page. This includes public page-scope variables, public properties, and public methods. Let's consider a simple scenario—setting the title of the final page.

The *Header* property on the *Page* class exposes the content of the <*head*> tag as programmable entities. To set the title of a content page or to add a stylesheet on a per-page basis, you just add some code to the *Page_Load* event:

```
<script runat="server">
void Page_Load(object sender, EventArgs e)
{
  Header.Title = "This is another page";
}
</script>
```

For this code to run, though, the <*head*> tag must be marked with the *runat* attribute. Note that the <*head*> tag can be defined in either the master or the content page. If you define it in the master, you can modify it programmatically in any content page at the price of adding an additional server control—the *HtmlHead* control.

Exposing Master Properties

To give an identity to a control in the master, you simply set the *runat* attribute and give the control an ID. Can you then access the control from within a content page? Not directly. The only way to access the master page object model is through the *Master* property. If a *TextBox* control named *Search* exists in the master, the following code will throw an exception:

```
Master.Search.Text = "ASP.NET 2.0";
```

As you saw earlier, the *Master* property defined on the *Page* class references the master page object for the content page. This means that only public properties and methods defined on the master page class are accessible.

The page shown in Figure 2-10 has a string of text in the rightmost part of the header set to "Title" by default. In the context of the master and related pages, that string indicates the subtitle of the page. So how can you expose, for example, the subtitle of the final page as a programmable property? Two steps are needed. First you render that string through a server-side control; then you create a public wrapper property to make the control (or one of its properties) accessible from the outside. The following code enhances the main.master file and shows how to define a public *SubTitle* property that wraps the *InnerText* property of the *__titleBar* control:

```
<%@ Master ClassName="MyMaster" Language="C#" %>

<script runat="server">
public string SubTitle
{
    get {return __titleBar.InnerText;}
    set {__titleBar.InnerText = value;}
}
</script>

<html>
⋮
<table cellspacing="2" cellpadding="2" border="0" class="headerStrip">
   <tr>
      <td align="Left">
        <asp:ContentPlaceHolder runat="server" ID="HeaderLeft" /></td>
      <td align="Right">
         <b><span runat="server" id="__titleBar">Title</span></b>
      </td>
   </tr>
</table>
⋮
<html>
```

Just as in ASP.NET 1.x, the <*span*> element marked *runat=server* is mapped to an *HtmlGenericControl* object and its text content is exposed through the *InnerText* property. This property is then wrapped by a new public property—*SubTitle*. Reading and writing the *SubTitle* property gets and sets the value of the *InnerText* property on the <*span*> tag.

Invoking Properties on the Master

When you write a content page, you access any public properties defined on the bound master page through the *Master* property. However, the *Master* property is defined as type *MasterPage* and doesn't contain any property or method definition specific of the master you're creating. The following code won't compile because there's no *SubTitle* property defined on the *MasterPage* class:

```
<script runat="server">
void Page_Load(object sender, EventArgs e)
{
    Master.SubTitle = "Welcome!";
}
</script>
```

What's the real type behind the *Master* property? The *Master* property represents the master page object as compiled by the ASP.NET runtime engine. This class follows the same naming convention as regular pages—*ASP.XXX_master*, where *XXX* is the name of the master file. The *ClassName* attribute on the *@Master* directive lets you assign a user-defined name to the master page class. To be able to call custom properties or methods, you must first cast the object returned by the *Master* property to the actual type:

```
((MyMaster)Master).SubTitle = "Welcome!";
```

Using the above code in the content page does the trick.

Changing the Master Page

The *Page* class defines the *MasterPageFile* property that can be used to get and set the master page associated with the current page. The *MasterPageFile* property is a string that points to the name of the master page file. Note that the *Master* property, which represents the current instance of the master page object, is a read-only property and can't be set programmatically. The *Master* property is set by the runtime after it loads the content of the file referenced by either the *MasterPageFile* attribute or *MasterPageFile* property. If both are set, an exception is thrown.

You can use a dynamically changing master page in ASP.NET 2.0—for example, for applications that can present themselves to users through different skins. You should follow two simple rules when you define a dynamic master page:

- Do not set the *MasterPageFile* attribute in the *@Page* directive.
- Make the page's *MasterPageFile* property point to the URL of the desired master page in the *Page_PreInit* event.

The content of the *@Page* directive is processed before the runtime begins working on the request. The *PreInit* event is fired right before the page handler begins working on the page, and this is your last chance to modify parameters at the page level. If you try to set the *MasterPageFile* property in the *Page_Init* or *Page_Load* events, an exception is raised.

Summary

Many Web sites consist of similar-looking pages that use the same header, footer, and perhaps some navigational menus or search forms. What's the recommended approach for reusing code across pages? One possibility is wrapping these user interface elements in user controls and referencing them in each page. Although the model is extremely powerful and produces highly modular code, when you have hundreds of pages to work with, it soon becomes unmanageable.

ASP.NET 2.0 introduces master pages for this purpose. A master page is a distinct file referenced at the application level as well as the page level that contains the static layout of the page. Regions that each derived page can customize are referenced in the master page with a special placeholder control. A derived page is simply a collection of blocks that the runtime uses to fill the holes in the master.

ASP.NET 2.0 is not the first environment to support template formats. Microsoft FrontPage and Macromedia products support templates, too. However, master pages are different, and compatibility with other template systems is not a goal. Likewise, true visual inheritance similar to that of Windows Forms is not a goal. Master pages promote reusability, but through a totally different

mechanism that is not based on class inheritance. Finally, the binding between master and content pages is defined statically and cannot be changed or set programmatically.

ASP.NET 2.0 comes with another mechanism that helps you build a certain type of page quickly and effectively reuse components—Web parts. As you'll see in the next chapter, Web parts provide the infrastructure needed to create Web applications that are modular, content rich, and customizable.