MARUDHAR KESARI JAIN COLLGE FOR WOMEN , VANIYAMBADI

PG DEPARTMENT OF COMPUTER APPLICATIONS

CLASS : II MCA

SUBJECT CODE : GCA43

SUBJECT NAME : WEB APPLICATION USING C#

SYLLABUS

UNIT - III: COMPONENT BASED PROGRAMMING Teaching Hours: 12

Introduction – Creating a Simple Component – Properties and State – Database Components – Consuming the Database Component – Enhancing the Component with Error Handling – Aggregate Information – Data Objects.

Chapter 21: Component-Based Programming

Overview

Component-based programming is a simple, elegant idea. When used properly, it allows your code to be more organized, consistent, and reusable. It's also incredibly easy to implement in a .NET application, because you never need to touch the registry or perform any special configuration.

To create a component, you separate a portion of your program's functionality and compile it to a separate assembly (DLL file). Your web pages (or any other .NET application) can then use this component in the same way that they use ordinary .NET classes. Best of all, your component provides *exactly* the features your code requires, and hides all the other messy details.

When combined with careful organization, component-based programming is the basis of good ASP.NET application design. In this chapter, we'll examine how you can create components (and why you should), and consider examples that show you how to encapsulate database functionality with a well-written business object. We'll also take a detour into the world of COM, and show how you can access legacy components by adding .NET wrappers around them.

Why Use Components?

To master ASP.NET development, you need to become a skilled user of the .NET class library. So far, you've learned how to use .NET components designed for sending mail, reading files, and interacting with databases. Though these class library ingredients are powerful, they aren't customizable, which is both an advantage and a weakness.

For example, it's convenient to be able to access any database in almost exactly the same way, irrespective of what type of information it contains. However, it's much less helpful to have to weave database details (such as SQL queries and connection strings) directly into your web page code. Though you can improve on this situation by storing database constants in a web.config application file (as described in Chapter 5), you still need to retrieve data using specific field names. If the structure of a commonly used database changes even slightly, you could be left with dozens of pages to update and retest. To solve these problems, you need to create an extra layer between your web page code and the database. This "extra layer" takes the form of custom component.

This database scenario is only one of the reasons you might want to create your own components. Component-based programming is really just a logical extension of good code organizing principles, and it offers a long list of advantages:

Increased security For example, you could configure your component to only allow access to certain tables, fields, or rows in a database. This is often easier than setting up complex permissions in the database itself. Because the application has to go through the component, it needs to play by its rules.

Better organization Components move the clutter out of your web page code. It also becomes easier for other programmers to understand your application's logic when it is broken down into separate components.

Easier troubleshooting It's impossible to oversell the advantage of components when testing and debugging an application. Component-based programs are broken down into smaller, tighter blocks of code, making it easier to isolate exactly where a problem is occurring.

More manageable Component-based programs are much easier to enhance and modify because common tasks are coded once (in the component), and used in as many places as required. Without components, commonly used code has to be copied and pasted throughout an application, making it extremely difficult to change and synchronize.

Code reuse Components can be shared with any ASP.NET application that needs the component's functionality. Even better, a component can be used by *any* .NET application, meaning you could create a common "backbone" of logic that's used by a web application and an ordinary Windows application.

Simplicity Components can provide multiple related tasks for a single client request (writing several records to a database, opening and reading a file in one step, or even starting and managing a database transaction). Similarly, components hide details—an application programmer can use a database component without worrying about the database name, location of the server, or the user account needed to connect. Even better, you can perform a search using certain criteria, and the component itself can decide whether to use a dynamically generated SQL statement or stored procedure.

Performance If you need to perform a long, time-consuming operation, you can create an asynchronous component to handle the work for you. That allows you to perform other tasks with the web page code, and return to pick up the result at a later time.

Components in ASP

In ASP programming, components were also a requirement to overcome the limitations of VBScript. Unlike ASP pages, components could be written in Visual Basic 6, which provided a richer syntax, a speedier compiled language, and the ability to perform tasks that just weren't allowed in ordinary script code. ASP.NET, as you've learned, doesn't suffer from any of these limitations, and hence components aren't required to compensate for ASP deficiencies. However, components still make good sense and are required to design a large-scale, well-organized ASP.NET application.

Component Jargon

Component-based programming is sometimes shrouded in a fog of specialized jargon. Understanding these terms helps sort out exactly what a component is supposed to do, and it also allows you to understand MSDN articles about application design. If you are already familiar with the fundamentals of components, feel free to skip ahead.

Three-Tier Design

The idea of three-tier design is that the functionality of most complete applications can be divided into three main levels (see Figure 21-1). The first level is the user interface (or presentation tier), which displays controls, and receives and validates user input. All the event handlers in your web page are in this first level. The second level is the "business" layer, where the application-specific logic takes place. For an e-commerce site, application-specific logic includes rules such as how shipping charges are applied to an order, when certain promotions are valid, and what customer actions should be logged. It doesn't involve generic .NET details such as how to open a file or connect to a database. The third layer is the data layer, where the application's information is stored in files or a database. The third layer contains logic about how to retrieve and update data, such as SQL queries or stored procedures.

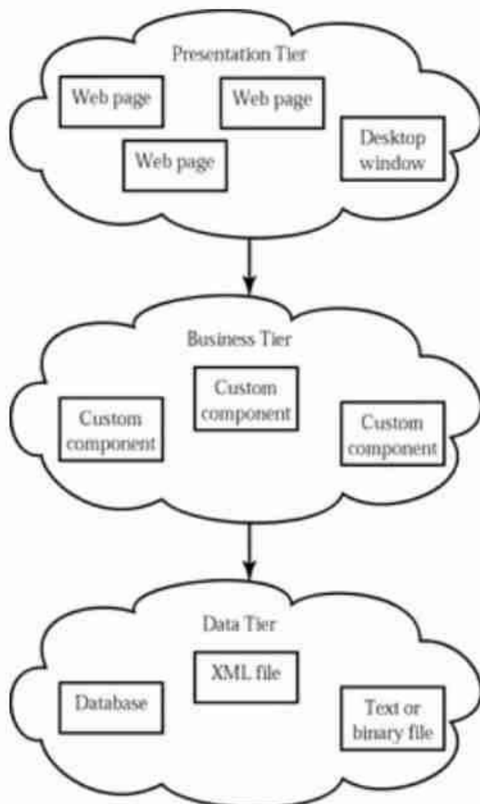

Figure 21-1: Three-tier design

The important detail about three-tier design is that information only travels from one level to an adjacent level. In other words, your user interface code should not try to directly access the database and retrieve information. Instead, it should go through the second layer and then arrive at the database.

This basic organization principle can't always be adhered to, but it's a good ideal. When you create a component, it's almost always used in the second layer to bridge the gap between the data and the user interface. In other words, if you want to fill a list of product categories in a list box, your user interface code would call a component, which would get the list from the database, and then return it to your code. Your web page code is isolated from the database—and if the database structure changes you need to change one concise component, instead of every page on your site.

Encapsulation

If three-tier design is the overall goal of component-based programming, encapsulation is the best rule of thumb. Encapsulation is the principle that you should create your application out of "black boxes" that hide information. For example, if you have a component that logs a purchase on an e-commerce site, that component handles all the details and allows only the essentially variables to be specified.

For example, this component might accept a user ID and an order item ID, and then handle all the other details. The calling code would not need to worry about how the component works or where the data is coming from—it just needs to understand how to use the component. (This principle is described in a lot of picturesque ways. For example, you know how to drive a car because you understand its component interface—the steering wheel and pedals—not because you understand the low-level details about combustion and the engine.)

Data Objects

Data objects are used in a variety of ways. In this book, we'll use the term to mean a custom

object you make that represents a certain grouping of data. For example, you could create a Person class that has properties like Height, Age, and EyeColor. Your code can then create data objects based on this class. You might want to use a data object to pass information from one portion of code to another. (Note that data objects are sometimes used to describe objects that handle data management. This is *not* the use we will follow in this book.)

Business Objects

The term *business object* often means different things to different people. Generally, business objects are the components in the second layer of your application that provide the extra layer between your code and the data source. They are called business objects because they enforce "business rules." For example, if you try to submit a purchase order without any items, the appropriate business object would throw an exception and refuse to continue. In this case, no .NET error has occurred—instead, you've detected the presence of a condition that should not be allowed according to your application's logic.

In our examples, business objects are also going to contain data access code. In an extremely complicated, large, and changeable system, you might want to further subdivide components, and actually have your user interface code talking to a business object which in turn talks to a data object which interacts with the data source. However, for most programmers this extra step is overkill, especially with the increased level of consistency that is provided by ADO.NET.

Creating a Simple Component

Technically, a component is just a collection of one or more classes that are compiled together as a unit. For example, Microsoft's System.Web.dll is a single (but very large) component that provides the objects found in many of the System.Web namespaces.

So far, the code examples in this book have only used a few types of class—mainly custom web page classes that inherit from System.Web.UI.Page and contain mostly event handling procedures. Component classes, on the other hand, won't interact directly with a web page and rarely inherit from anything. They are more similar to the custom Web Service classes described in the fourth part of this book, which collect related features together in a series of utility methods.

Web Services versus Components

Web Services provide some of the same opportunities for code reuse as custom components. However, Web Services are primarily designed as an easy way to share functionality across different computers and platforms. A component, on the other hand, is not nearly as easy to share with the wide world of the Internet, but is far more efficient for sharing *internally* (for example, between different applications in the same company or different web sites on the same server). For that reason, Web Services and components don't directly compete—in fact, a Web Service could even make use of a component (or vice versa). In some cases you might find yourself programming a site with a mixture of the two, putting the code that needs to be reused in-house into components, and the functionality that needs to be made publicly available into Web Services.

The Component Class

You have two options when writing a component.

- You can code it by hand in a .vb text file, and then compile it with the vbc.exe command-line

compiler described in earlier chapters.

- You can use Visual Studio .NET, and start a new Class Library project. When choosing the project directory, you need to specify a real (physical) directory, not the virtual directory name. Figure 21-2 shows Visual Studio .NET's New Project window with a Class Library project.

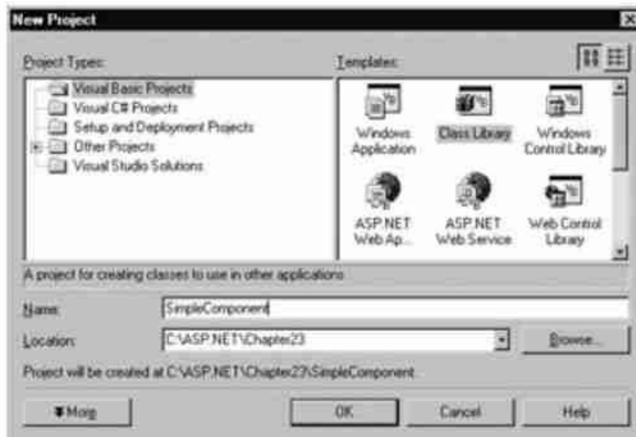

Figure 21-2: Creating a component in Visual Studio .NET

These options are equivalent. In your .vb file, you define the component as a class.

```
Public Class SimpleTest
End Class
```

Remember, a component can contain more than one class. You can create this other classes in the same file, or in separate files, which will be compiled together into one assembly.

```
Public Class SimpleTest
End Class

Public Class SimpleTest2
End Class
```

To add functionality to your class, you add public methods (functions or subroutines). The web page code calls these methods to retrieve information or perform a task. The following example shows one of the simplest possible components, which does nothing more than return a string to the calling code.

```
Public Class SimpleTest
  Public Function GetInfo(param As String) As String
    Return "You invoked SimpleTest.GetInfo() with '" & _
        param & "'"
  End Function
End Class

Public Class SimpleTest2
  Public Function GetInfo(param As String) As String
    Return "You invoked SimpleTest2.GetInfo() with '" & _
        param & "'"
  End Function
End Class
```

Usually, these classes will be organized in a special namespace. You can group classes into a namespace at the file level by using the Namespace block structure. In the following example, the classes will be accessed in other applications as SimpleComponent.SimpleTest and SimpleComponent.SimpleTest2. If needed, you can create multiple levels of nested namespaces.

```
Namespace SimpleComponent

   Public Class SimpleTest
      ' (Class code omitted.)
   End Class

   Public Class SimpleTest2
      ' (Class code omitted.)
   End Class

End Namespace
```

In Visual Studio .NET, your component is automatically placed in a root namespace that has the project name. This code doesn't appear in the .vb file (although you can add additional namespaces there). To change the name of your root namespace, right-click on the project in the Solution Explorer and select Properties. Look under the Common Properties | General group for the root namespace setting (see Figure 21-3). You can also use this window to configure the name that will be given to the compiled assembly file.

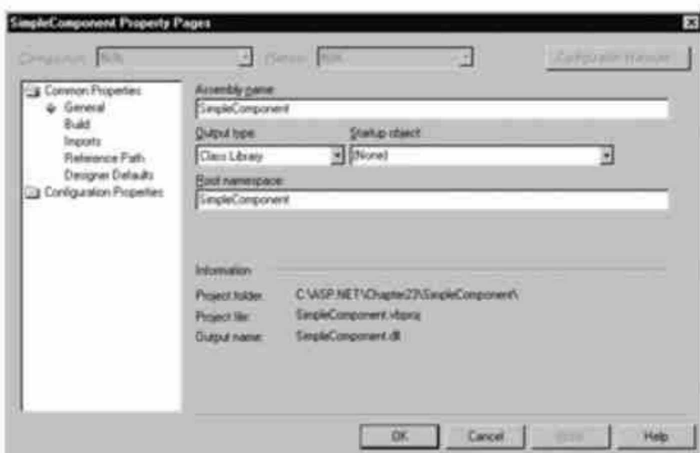

Figure 21-3: Setting the root namespace

Compiling the Component Class

In Visual Studio .NET, you can compile this class just by right-clicking on the project in the Solution Explorer and choosing Build. You can't actually run the project, because it isn't an application, and it doesn't provide any user interface.

If you are writing the class by hand in a text file, you can compile it in more or less the same way as you compile a .aspx code-behind file or .asmx web service.

```
vbc /t:library /r:System.dll /r:System.Web.dll SimpleComponent.vb
```

The /t:library parameter indicates that you want to create a DLL assembly instead of an EXE assembly. The compiled assembly will have the name SimpleComponent.dll. The /r parameter specifies any dependent assemblies you use. The compilation process is the same as that described with web pages in Chapter 5.

All Components Must Be Compiled

Unlike web pages and Web Services, you *must* compile a component before you can use it. Components are not hosted by the ASP.NET service and IIS, and thus cannot be compiled automatically when they are needed.

Using the Component

Using the component in an actual ASP.NET page is easy. If you are working without Visual Studio .NET, all you really need to do is copy the DLL file into the application's bin directory. ASP.NET automatically monitors this directory, and makes all of its classes available to any web page in the application, just as it does with the native .NET types.

In Visual Studio .NET, you need to specify the component that you plan to use as a reference. This allows Visual Studio .NET to provide its usual syntax checking and IntelliSense. Otherwise, it will interpret your attempts to use the class as mistakes and refuse to compile your code.

To link a component, select Project | Add Reference from the menu (you can also right-click on the References group in the Solution Explorer). Select the .NET tab, and click Browse. Find and select the SimpleComponent.dll file we created in the previous example, and choose Open (see Figure 21-4).

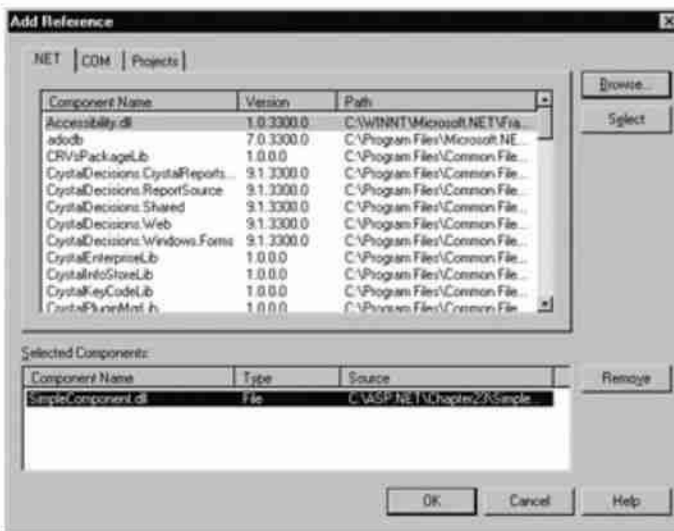

Figure 21-4: Adding a custom component reference

Once you add the reference, the DLL file will be automatically copied to the bin directory of your current project. You can verify this by checking the Full Path property of the SimpleComponent reference, or just browsing to the directory in Windows Explorer. The nice thing is that this file will automatically be overwritten with the most recent compiled version of the assembly every time you run the project.

Project References

You can also create a Visual Studio .NET solution that combines a class library project with an ASP.NET application project. This technique, which we examined with Web Services in Chapter 20, allows you to debug a component while it is in use alongside the application. In this case, you don't add a reference to the DLL file. Instead, you add a special project reference to the class library project in the same solution, using the Projects tab of the Add Reference window. With project references, Visual Studio .NET automatically compiles the component for you when needed.

You can now use the component by creating instances of the SimpleTest or SimpleTest2 class.

```
Public Class TestPage
  Inherits Page
```

```
    Protected lblResult As Label

    Private Sub Page_Load(sender As Object, e As EventArgs) _
     Handles MyBase.Load
        Dim TestComponent As New SimpleComponent.SimpleTest()
        Dim TestComponent2 As New SimpleComponent.SimpleTest2()
        lblResult.Text = TestComponent.GetInfo("Hello") & "<br><br>"
        lblResult.Text &= TestComponent2.GetInfo("Bye")
    End Sub

End Class
```

The output for this page, shown in Figure 21-5, combines the return value from both GetInfo methods.

Figure 21-5: The SimpleTest component output

To make this code slightly simpler, you could have chosen to use shared methods, which don't need a valid instance. A shared GetInfo method would look like this:

```
Public Class SimpleTest
    Public Shared Function GetInfo(param As String) As String
        Return "You invoked SimpleTest.GetInfo() with '" & _
            param & "'"
    End Function
End Class
```

The web page would access the shared GetInfo method through the class name, and would not need to create an object.

```
Private Sub Page_Load(sender As Object, e As EventArgs) _
 Handles MyBase.Load
    lblResult.Text = SimpleComponent.SimpleTest.GetInfo("Hello")
End Sub
```

Properties and State

The SimpleTest classes provide functionality through public methods. If you're familiar with class-based programming (or if you've read Chapter 3), you'll remember that classes can also store information in private member variables, and provide property procedures that allow the calling code to modify this information. For example, a Person class might have a FirstName property.

When you create classes that use property procedures, you are using *stateful* design. In stateful design, the class has the responsibility of maintaining certain pieces of information. In *stateless* design, like that found in our SimpleTest component, no information is retained between method calls. Compare that to a stateful SimpleTest class.

```
Public Class SimpleTest

   Private _Data As String
   Public Property Data() As String
      Get
         Return _Data
      End Get
      Set(Value As String)
         _Data = Value
      End Set
   End Property

   Public Function GetInfo() As String
      Return "You invoked SimpleTest.GetInfo()," & _
           "and _Data is '" & _Data & "'"
   End Function

End Class
```

In the programming world, there have been a number of great debates arguing whether stateful or stateless programming is best. Stateful programming is the most natural, object-oriented approach, but it also has a few disadvantages. In order to accomplish a common task, you might need to set several properties before calling a method. Each of these individual steps adds a little bit of unneeded overhead. A stateless design, on the other hand, often performs all its work in a single method call. However, because no information is retained in state, you may need to specify several parameters, which can make for tedious programming. A good example of stateful versus stateless objects is shown by the FileInfo and File classes, which are described in Chapter 16.

There is no short answer about whether stateful or stateless design is best, and it often depends on the task at hand. However, stateful designs have some important shortcomings that are difficult to circumvent. The most important of these is the fact that properties need to be changed individually, and that can allow classes to be placed in inconsistent or invalid states. Even if this state is temporary, it can lead to a problem if a software or hardware failure occurs. This type of problem is most commonly seen with transactions.

The next example illustrates the difference with two ways to design an Account class.

A Stateful Account Class

Consider a stateful account class that represents a single customers account. Information is read from the database when it is first created in the constructor method, and can be updated using the Update method.

```
Public Class Account

   Private _AccountNumber As Integer
   Private _Balance As Decimal

   Public Property Balance() As Decimal
      Get
         Return _Balance
      End Get
      Set(Value As Decimal)
         _Balance = Value
      End Set
   End Property

   Public Sub New(AccountNumber As Integer)
      ' (Code to read account record from database goes here.)
   End Sub

   Public Sub Update()
      ' (Code to update database record goes here.)
```

```
  End Sub

End Class
```

If you have two Account objects that expose a Balance property, you need to perform two separate steps to transfer money from one account to another. Conceptually, the process works like this:

```
' Create an account object for each account,
' using the account number.
Dim AccountOne As New Bank.CustomerAccount(122415)
Dim AccountTwo As New Bank.CustomerAccount(123447)
Dim Amount As Decimal = 1000

' Withdraw money from one account.
AccountOne.Balance -= Amount

' Deposit money in the other account.
AccountTwo.Balance += Amount

' Update the underlying database records using an Update method.
AccountOne.Update()
AccountTwo.Update()
```

The problem here is that if this task is interrupted halfway through by an error, you'll end up with at least one unhappy customer.

A Stateless AccountUtility Class

A stateless object, on the other hand, might only expose a shared method called FundTransfer, which performs all its work in one method.

```
Public Class AccountUtility

  Public Shared Sub FundTransfer(AccountOne As Integer, _
   AccountTwo As Integer, Amount As Decimal)
    ' (The code here retrieves the two database records,
    '  changes them, and updates them.)
  End Sub

End Class
```

The calling code can't use the same elegant Account objects, but it can be assured that account transfers are protected from error. Because all the database operations are performed at once, they can use a database stored procedure for greater performance, and a transaction to ensure that the withdrawal and deposit either succeed or fail as a whole.

```
' Store the account and transfer details.
Dim Amount As Decimal = 1000
Dim AccountIDOne As Integer = 122415
Dim AccountIDTwo As Integer = 123447

Bank.AccountUtility.FundTransfer(AccountIDOne, AccountIDTwo, _
                    Amount)
```

In a mission-critical system, transactions are often a requirement. For that reason, classes that retain very little state information are often the best design approach, even though they are not quite as elegant or as well organized.

Database Components

Clearly, components are extremely useful. But if you're starting a large programming project, you may not be sure what features are the best candidates for being made into separate components. Learning how to break an application into components and classes is one of the great arts of programming, and it takes a good deal of practice and fine-tuning.

One of the most common types of components is a database component. Database components are an ideal application of component-based programming for several reasons:

- Databases require extraneous details (connection strings, field names, and so on) that can distract from the application logic, and could easily be encapsulated by a well-written component.
- Databases frequently change. Even if the underlying table structure remains constant and additional information is never required (which is far from certain), queries may be replaced by stored procedures, and stored procedures may be redesigned.
- Databases have special connection requirements. You may even need to change the database access code for reasons unrelated to the application. For example, after profiling and testing a database, you might discover that you can replace a single query with two queries or a more efficient stored procedure. In either case, the returned data remains constant, but the data access code is dramatically different.
- Databases are used repetitively in a finite set of ways. In other words, a common database routine should be written once and is certain to be used many times over.

A Simple Database Component

To examine the best way to create a database component, we'll consider a simple application that provides a classifieds page listing items that various individuals have for sale. The database uses two tables: an Item table that lists the description and price of a specific sale item, and a Categories table with possible groupings under which items can be grouped. The relationship is shown in Figure 21-6.

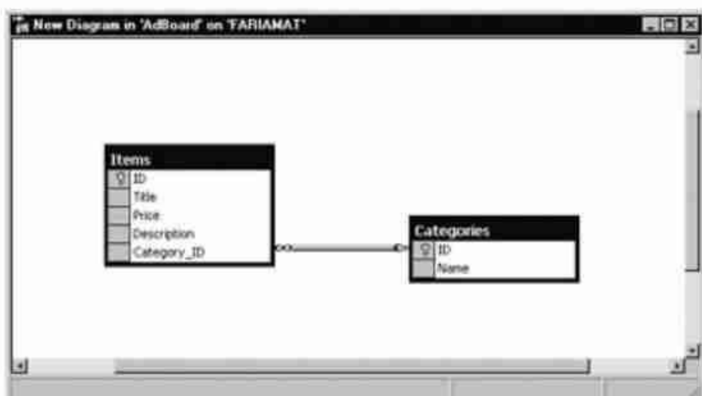

Figure 21-6: The AdBoard database relationships

In our example, we are connecting with an SQL server database using the OLE DB part of the class library. You can create this database yourself, or you can refer to the online samples, which include the SQL code that will generate it automatically. To start, the Categories table is preloaded with a standard set of allowed categories.

The database component is simple. It's an instance class that retains some basic information (such as the connection string to use), but it does not allow the client to change this information, and therefore does not need any property procedures. Instead, it performs most of its work in

methods such as GetCategories and GetItems. These methods return DataSets with the appropriate database records. This type of design creates a fairly thin layer over the database—it handles some details, but the client is still responsible for working with familiar ADO.NET classes such as the DataSet.

```
Imports System.Data
Imports System.Data.OleDb
Imports System.Configuration

Public Class DBUtil
   Private ConnectionString As String

   Public Sub New()
      ConnectionString = ConfigurationSettings.AppSettings( _
                   "ConnectionString")
   End Sub

   Public Function GetCategories() As DataSet
      Dim Query As String = "SELECT * FROM Categories"
      Dim ds As DataSet = FillDataSet(Query, "Categories")
      Return ds
   End Function

   Public Overloads Function GetItems() As DataSet
      Dim Query As String = "SELECT * FROM Categories"
      Dim ds As DataSet = FillDataSet(Query, "Items")
      Return ds
   End Function

   Public Overloads Function GetItems(categoryID As Double) _
    As DataSet
      Dim Query As String = "SELECT * FROM Items "
      Query &= "WHERE Category_ID='" & categoryID & "'"
      Dim ds As DataSet = FillDataSet(Query, "Items")
      Return ds
   End Function

   Public Sub AddCategory(name As String)
      Dim Insert As String = "INSERT INTO Categories "
      Insert &= "(Name) VALUES ('" & name & "')"

      Dim con As New OleDbConnection(ConnectionString)
      Dim cmd As New OleDbCommand(Insert, con)

      con.Open()
      cmd.ExecuteNonQuery()
      con.Close()
   End Sub

   Public Sub AddItem(title As String, description As String, _
              price As Decimal, categoryID As Integer)
      Dim Insert As String = "INSERT INTO Items "
      Insert &= "(Title, Description, Price, Category_ID)"
      Insert &= "VALUES ('" & title & "', '" & description & "', "
      Insert &= price & ", '" & categoryID & "')"

      Dim con As New OleDbConnection(ConnectionString)
      Dim cmd As New OleDbCommand(Insert, con)
      con.Open()
      cmd.ExecuteNonQuery()
      con.Close()
   End Sub

   Private Function FillDataSet(query As String, _
                   tableName As String) As DataSet
      Dim con As New OleDbConnection(ConnectionString)
      Dim cmd As New OleDbCommand(query, con)
```

```
    Dim adapter As New OleDbDataAdapter(cmd)

    Dim ds As New DataSet()
    adapter.Fill(ds, tableName)
    con.Close()

    Return ds
  End Function

End Class
```

Interesting Facts About this Code

- This code automatically retrieves the connection string from the web.config file when the class is created, as described in Chapter 5. This trick enhances encapsulation, but if the client web application does not have the appropriate setting, the component will not work. If you are creating a component that needs to handle ASP.NET and Windows applications, this design wouldn't be suitable.
- This class uses an overloaded GetItems method. This means the client can call GetItems with no parameters to return the full list, or with a parameter indicating the appropriate category. (Chapter 2 provides an introduction to overloaded functions.)
- Each method that accesses the database opens and closes the connection. This is a far better approach than trying to hold a connection open over the lifetime of the class, which is sure to result in performance degradation in multi-user scenarios.
- The code uses its own private FillDataSet function to make the code more concise. This is not made available to clients. Instead, the FillDataSet function is then used by the GetItems and GetCategories methods.

Consuming the Database Component

To use this component in a web application, we first have to make sure that the appropriate connection string is configured in the web.config file, as shown here.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

 <appSettings>
  <add key="ConnectionString"
     value="Provider=SQLOLEDB.1;Data Source=localhost;Initial
          Catalog=AdBoard;Integrated Security=SSPI" />
 </appSettings>

 <system.web>
  <!-- Configuration sections go here. -->
 </system.web>

</configuration>
```

Next, compile and copy the component DLL file, or add a reference to it if you are using Visual Studio .NET. The only remaining task is to add the user interface.

To test out this component, you can create a simple test page. In our example, shown in Figure 21-7, this page allows users to browse the current listing by category and add new items. When the user first visits the page, it prompts the user to select a category.

Figure 21-7: The AdBoard categories

Once a category is chosen, the matching items are displayed, and a panel of controls appears, which allows the user to add a new entry to the ad board under the current category (see Figure 21-8).

Figure 21-8: The AdBoard listing

The page code creates the component when needed, and displays the appropriate database information by binding the DataSet to the drop-down list or DataGrid control.

```
Public Class AdBoard
  Inherits Page

    Protected lstCategories As DropDownList
    Protected WithEvents cmdDisplay As Button
    Protected pnlNew As Panel
    Protected txtDescription As TextBox
    Protected txtPrice As TextBox
```

```
    Protected txtTitle As TextBox
    Protected gridItems As DataGrid
    Protected WithEvents cmdAdd As Button

    Private Sub Page_Load(sender As Object, e As EventArgs) _
      Handles MyBase.Load
        If Me.IsPostBack = False Then
            Dim DB As New SimpleDB.DBUtil()

            lstCategories.DataSource = DB.GetCategories()
            lstCategories.DataTextField = "Name"
            lstCategories.DataValueField = "ID"
            lstCategories.DataBind()
            pnlNew.Visible = False
        End If
    End Sub

    Private Sub cmdDisplay_Click(sender As Object, e As EventArgs) _
      Handles cmdDisplay.Click
        Dim DB As New SimpleDB.DBUtil()

        gridItems.DataSource = DB.GetItems( _
          lstCategories.SelectedItem.Value)
        gridItems.DataBind()
        pnlNew.Visible = True
    End Sub

    Private Sub cmdAdd_Click(sender As Object, e As EventArgs) _
      Handles cmdAdd.Click
        Dim DB As New SimpleDB.DBUtil()

        DB.AddItem(txtTitle.Text, txtDescription.Text, _
                txtPrice.Text, lstCategories.SelectedItem.Value)

        gridItems.DataSource = DB.GetItems(_
          Val(lstCategories.SelectedItem.Value))
        gridItems.DataBind()
    End Sub

End Class
```

Interesting Facts About this Code

- Not all the functionality of the component is used in this page. For example, the page does not use the AddCategory method or the version of GetItems that does not require a category number. This is completely normal. Other pages may use different features from the component.
- The page is free of data access code. It does, however, need to understand how to use a DataSet, and it would need to know specific field names in order to create a more attractive DataGrid with custom templates for layout (instead of automatically generated columns).
- The page could be improved with error handling code or validation controls. As it is, no validation is performed to ensure that the price is numeric, or to even ensure that the required values are supplied.

Enhancing the Component with Error Handling

One way the component could be enhanced is with better support for error reporting. As it is, any database errors that occur will be immediately returned to the calling code. In some cases (for example, if there is a legitimate database problem), this is a reasonable approach, because the component can't handle the problem.

However, there is one common problem that the component fails to handle properly. This problem occurs if the connection string is not found in the web.config file. Though the component tries to read the connection string as soon as it is created, the calling code won't realize there is a problem until it tries to use a database method.

A better approach is to notify the client as soon as the problem is detected, as shown in the following code example.

```
Public Class DBUtil
  Private ConnectionString As String

  Public Sub New()
    ConnectionString = ConfigurationSettings.AppSettings(_
              "ConnectionString")
    If ConnectionString = "" Then
      Throw New ApplicationException(_
       "Missing ConnectionString variable in web.config file.")
    End If
  End Sub

  ' (Other class code omitted.)

End Class
```

You Can Debug Component Code

If you are debugging your code in Visual Studio .NET, you'll find that you can single step from your web page code right into the code for the component, even if it isn't a part of the same solution. The appropriate source code file will be loaded into your editor automatically, as long as it is available.

This code throws an ApplicationException with a custom error message that indicates the problem. To provide even better reporting, you could create your own exception class that inherits from ApplicationException, as described in Chapter 11.

Enhancing the Component with Aggregate Information

The component doesn't have to limit the type of information it provides to DataSets. Other information can also be useful. For example, you might provide a read-only property called ItemFields that returns an array of strings representing the names for fields in the Items table. Or you might add another method that retrieves aggregate information about the entire table, such as the average cost of an item or the total number of currently listed items.

```
Public Class DBUtil
  ' (Other class code omitted.)

  Public Function GetAveragePrice() As Decimal
    Dim Query As String = "SELECT AVG(Price) FROM Items"

    Dim con As New OleDbConnection(ConnectionString)
    Dim cmd As New OleDbCommand(Query, con)

    con.Open()
    Dim Average As Decimal = cmd.ExecuteScalar()
    con.Close()

    Return Average
  End Function
```

```
    Public Function GetTotalItems() As Integer
        Dim Query As String = "SELECT Count(*) FROM Items"

        Dim con As New OleDbConnection(ConnectionString)
        Dim cmd As New OleDbCommand(Query, con)

        con.Open()
        Dim Count As Integer = cmd.ExecuteScalar()
        con.Close()

        Return Count
    End Function

End Class
```

These commands use some customized SQL that may be new to you (namely, the Count and AVG functions). However, these methods are just as easy to use from the client's perspective as GetItems and GetCategories.

```
Dim DB As New SimpleDB.DBUtil()
Dim AveragePrice As Decimal = DB.GetAveragePrice()
Dim TotalItems As Integer = DB.GetTotalItems()
```

Read-Only Properties versus Methods

It may have occurred to you that you could return information such as the total number of items through a read-only property procedure (like TotalItems) instead of a method (in this case, GetTotalItems). Though this would work, property procedures are better left to information that is maintained with the class (in a private variable) or is easy to reconstruct. In this case, it takes a database operation to count the number of rows, and this database operation could cause an unusual problem or slow down performance if used frequently. To help reinforce that fact, a method is used instead of a property.

Enhancing the Component with a Data Object

Sometimes a DataSet is not the best way to return information to the client. One possible reason could be that the field names are unusual, unintuitive, or subject to change. Another reason might be that the client isn't using data binding, and doesn't want to worry about using the ADO.NET objects to extract the appropriate information.

In these cases, you can create a more advanced component that returns information using a custom data class. This approach takes encapsulating one step further, and isolates the calling code from most of the underlying database details. For example, you could create the following classes to represent the important details for items and categories.

```
Public Class Item
    Public ID As Integer
    Public Title As String
    Public Price As Decimal
    Public Description As String
    Public Category As String
End Class

Public Class Category
    Public ID As Integer
    Public Name As String
End Class
```

For the most part, the member variable names match the actual field names, although they don't need to do so. You can add these classes directly to the .vb file that contains the DBUtil component class.

The DBUtil class requires a slight bit of rewriting to use the new classes. To simplify matters, this version only uses the category-specific version of the GetItems method. You'll notice, however, that this method has been enhanced to perform a join query, and return the matching category name (instead of just the ID) for each item row.

```
Public Class DBUtil
  ' (Other class code omitted.)

  Public Function GetCategories() As Collection
    Dim Query As String = "SELECT * FROM Categories"
    Dim ds As DataSet = FillDataSet(Query, "Categories")
    Dim dr As DataRow
    Dim Categories As New Collection()

    For Each dr In ds.Tables("Categories").Rows
      Dim Entry As New Category()
      Entry.ID = dr("ID")
      Entry.Name = dr("Name")
      Categories.Add(Entry)
    Next

    Return Categories
  End Function

  Public Function GetItems(categoryID As Double) As Collection
    Dim Query As String = "SELECT * FROM Items "
    Query &= "INNER JOIN Categories ON "
    Query &= Category_ID=Categories.ID "
    Query &= "WHERE Category_ID='" & categoryID & "'"
    Dim ds As DataSet = FillDataSet(Query, "Items")
    Dim dr As DataRow
    Dim Items As New Collection()

    For Each dr In ds.Tables("Items").Rows
      Dim Entry As New Item()
      Entry.ID = dr("ID")
      Entry.Title = dr("Title")
      Entry.Price = dr("Price")
      Entry.Description = dr("Description")
      Entry.Category = dr("Name")
      Items.Add(Entry)
    Next

    Return Items
  End Function

End Class
```

Now the GetCategories and GetItems methods return a collection of Category or Item objects. Whether this information is drawn from a file, database, or created manually is completely transparent to the calling code.

Unfortunately, this change has broken the original code. The problem is that you can't use data binding with the member variables of an object. Instead, your class needs to provide property procedures. For example, you can rewrite the classes for a quick test, using the following code.

```
Public Class Item
  Public ID As Integer
  Public Price As Decimal
  Public Description As String
```

```
    Private _Title As String
    Public Property Title() As String
        Get
            Return _Title
        End Get
        Set(Value As String)
            _Title = Value
        End Set
    End Property

    Private _Category As String
    Public Property Category() As String
        Get
            Return _Category
        End Get
        Set(Value As String)
            _Category = Value
        End Set
    End Property
End Class

Public Class Category

    Private _ID As Integer
    Public Property ID() As Integer
        Get
            Return _ID
        End Get
        Set(Value As Integer)
            _ID = Value
        End Set
    End Property

    Private _Name As String
    Public Property Name() As String
        Get
            Return _Name
        End Get
        Set(Value As String)
            _Name = Value
        End Set
    End Property

End Class
```

Now data binding will be supported for the ID and Name properties of any Category objects, and the Title and Category properties of all Item objects. The original AdBoard code won't require any changes (unless you have changed the component's namespace or class names), but it will work with one slight difference. Because the DataGrid can only bind to two pieces of information, it will only display two columns (see Figure 21-9).

Figure 21-9: The AdBoard with the new component

Of course, the calling code doesn't need to use data binding. Instead, it can work with the Item and Category objects directly, just as you can work with .NET structures such as DateTime and Font. Here's an example of how you could fill the list box manually.

```
Dim DB As New DataObjectDB.DBUtil()
Dim Item As New DataObjectDB.Category()
Dim Categories As Collection = Components.GetCategories()

For Each Item In Categories
   lstCategories.Items.Add(New ListItem(Item.Name, Item.ID))
Next
```

Data Objects Can Also Add Functionality

The Category and Item objects are really just special structures that group together related data. You can also add methods and other functionality into this class. For example, you could rewrite the AddItem method in the DBUtil component so that it required an Item object parameter. The Item object could provide some basic error checking when you create and set its properties that would notify you of invalid data before you attempt to add it to the database.

This approach can streamline a multi-layered application, but it can also add unneeded complexity, and risk mingling database details deep into your classes and code. For more information, you might want to read a book on three-tier design, or refer to the architectural white papers provided by Microsoft in the MSDN Knowledge Base.

Using COM Components

If you're a longtime ASP developer, you probably have developed your own components at one time or another. You may even have legacy COM components that you need to use in an ASP.NET application. While the ideal approach is to redesign and recode these components as .NET assemblies, time constraints don't always allow for these changes, particularly if you are already involved in a large migration effort to convert existing web pages. There is no way to

directly mingle the unmanaged code in a COM component with .NET code, as this would violate the security and verification services built into the Common Language Runtime.

Fortunately, .NET does provide a method to interact with COM components. It's called a Runtime Callable Wrapper (RCW)—a special .NET proxy class that sits between your code and the COM component (see Figure 21-10). The RCW handles the transition from managed Visual Basic .NET code to the unmanaged COM component. Though this may seem simple, it actually involves marshalling calls across application domains, converting data types, and using traditional COM interfaces. It also involves converting COM events to the completely different .NET event framework.

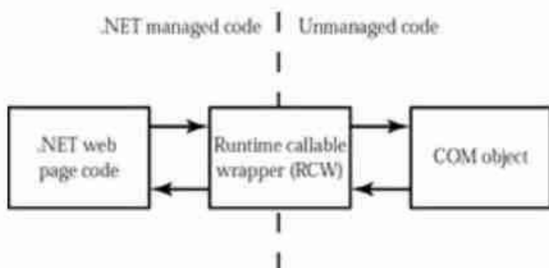

Figure 21-10: Accessing a COM object through an RCW

The RCW presents your code with an interface that mimics the underlying COM object—it's the same idea as the Web Service proxy class, which "pretends" to be the Web Service in order to simplify your coding. Once you have created an RCW, you can use it in the same way that you would use the actual COM component, invoking its methods, using its properties, and receiving its events.

Creating an RCW in Visual Studio .NET

There are two basic ways to generate an RCW. The first method is to use Visual Studio .NET. In this case, select Project | Add Reference from the menu, and then select the COM tab to display a list of currently installed COM objects. Find the COM object you want to use and double-click on it, or browse to it using the Browse button (see Figure 21-11).

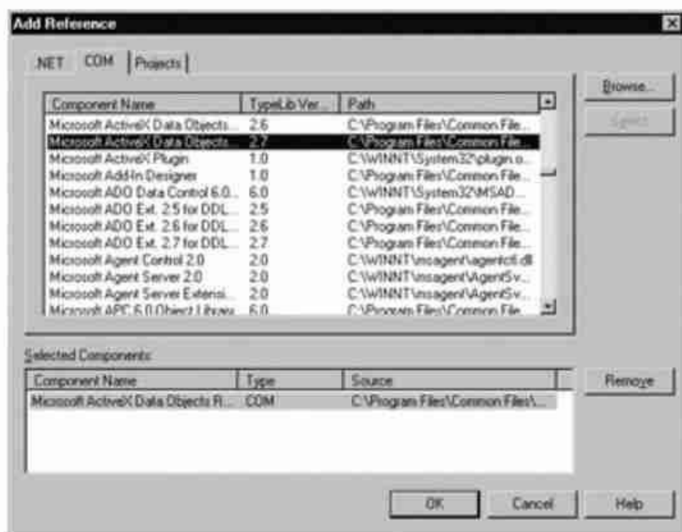

Figure 21-11: Adding a COM reference

When you click OK, .NET will check for a primary Interop assembly, which is an RCW that has been created and signed by the same party that created the original COM component. If it cannot find a primary Interop assembly, it will prompt you with a warning and generate the RCW DLL file automatically. This file will be placed in the bin directory of your application (as are all components), and will automatically be given the name of the COM component's type library.

For example, in Figure 21-11, a reference is being added to the COM-based ADO data access library (the precursor to ADO.NET). The RCW has the name ADODDB.dll and is made available through the namespace ADODB. This primary Interop assembly is provided with the .NET framework, but you can create your own RCW assemblies for other COM components just as easily.

You can find the component's namespace by examining the current references for your project in the Solution Explorer (see Figure 21-12).

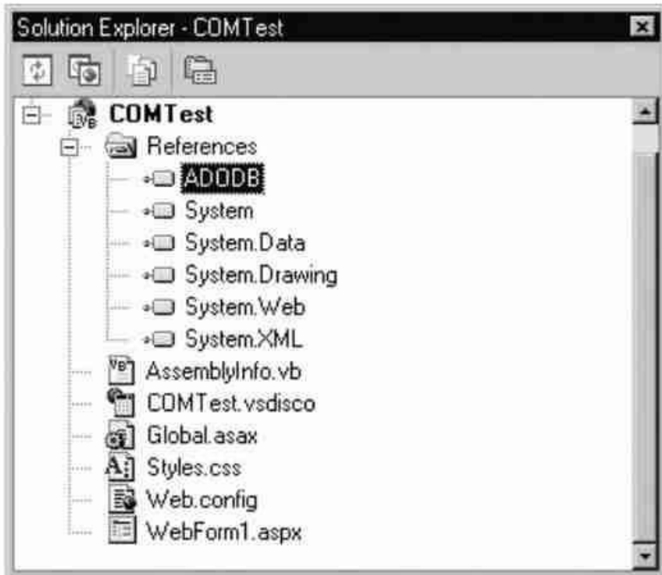

Figure 21-12: The Runtime Callable Wrapper

You can now create classes from the COM component, and use them as though they are native .NET classes (which, of course, they now are). That means you use the .NET syntax to create an object or receive an event using the classes in the RCW (see Figure 21-13).

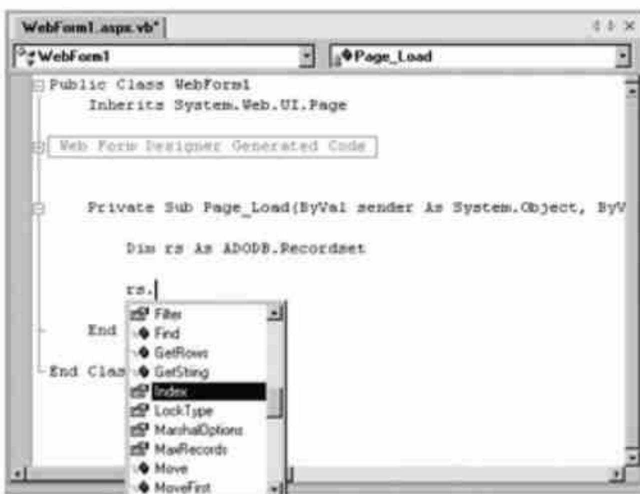

Figure 21-13: Using an RCW class

Creating an RCW with Type Library Import

If you aren't using Visual Studio .NET, you can create a wrapper assembly using a command-line utility that is included with the .NET framework. This utility, which is known as the type library import tool, is quite straightforward. The only piece of information you need is the filename that contains the COM component.

The following statement creates an RCW with the default filename and namespace (as it would be in Visual Studio .NET), assuming that the MyCOMComponent.dll file is in the current directory.

```
tlbimp MyCOMComponent.dll
```

Assuming that the MyCOMComponent has a type library named MyClasses, the generated RCW file will have the name MyClasses.dll and will expose its classes through a namespace named MyClasses. You can also configure these options with command-line parameters, as described in Table 21-1.

Table 21-1: Type Library Import Parameters

Parameter	Description
/out:Filename	Sets the name of the RCW file that will be generated.
/namespace:Namespace	Sets the namespace that will be used for the classes in the RCW assembly.
/asmversion: VersionNumber	Specifies the version number that will be given to the RCW assembly.
/reference:Filename	If you do not specify the /reference option, tlbimp.exe automatically imports any external type libraries that are referenced by the current type library. If you specify the /reference option, tlbimp.exe attempts to resolve these references using types in the .NET assemblies you specify before it imports other type libraries.
/strictref	Does not import a type library if the Tblimp.exe cannot resolve all references in the current assembly or the assemblies specified with the /reference option.

There are also additional options for creating signed assemblies, which is useful if you are a component vendor who needs to distribute a .NET assembly to other clients. These features are described in the MSDN reference. Additionally, you can use .NET classes to manually write your own wrapper class. This process is incredibly painstaking, however, and in the words of Microsoft's own documentation, "seldom performed."