Building Client/Server Applications with VB .NET: An Example-Driven Approach

JEFF LEVINSON

Apress™

Building Client/Server Applications with VB .NET: An Example-Driven Approach
Copyright © 2003 by Jeff Levinson

ISBN (pbk): 1-59059-070-8

Printed and bound in the United States of America 12345678910

Technical Reviewers: Mary Romero Sweeney, Eric Mashlan

Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Simon Hayes, Karen Watterson, John Zukowski

Managing Editor: Grace Wong

Project Manager: Tracy Brown Collins

Project Editor: Janet Vail

Copy Editor: Kim Wimpsett

Compositor: Susan Glinert

Artist and Cover Designer: Kurt Krames

Indexer: Kevin Broccoli

Production Manager: Kari Brooks

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.

Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 10

Using Reflection

REFLECTION IS THE ABILITY of code to examine itself. The fact that code has the ability to examine itself should come as no surprise because this ability is needed for even such mundane tasks as figuring out the particular address of a method that needs to be called. But Microsoft has taken the mundane and made it spectacular. Microsoft has given you the ability to examine code. The fact that code has knowledge of itself is nothing new, but just because the code knew about itself did not mean you could get that information. And then Microsoft introduced one more awesome ability: You can create custom attributes with which to tag your code, and you can read these attributes using reflection. This is something unique to the .NET Framework.

NOTE To be fair, Java has had reflection since its inception, but Java does not give the developer the ability to create custom attributes. So you can examine the code all you want to determine things about it, but you cannot say anything about it.

In this chapter you will examine reflection in the context of two practical examples. The first example demonstrates reading from attributes to determine how to load a listview without knowing anything about the object that you are taking data from and without knowing anything about the data itself. This project is a small, independent demonstration. For the second example you will incorporate attribute classes into your application in the RegionDescription class. This example shows you how to turn the business rules that you have created into custom attributes and make the class truly self-aware.

TIP After developing this method of implementing business rules, my team and I were able to save approximately 30,000 lines of code in a recent project. We were able to quantify this by extrapolating out the amount of code we saved after converting just a few classes to this method. This also made maintenance of the application much simpler!

Generating Code Dynamically

One other ability of reflection that I wanted to mention (but that I will not cover in this book) is the ability to dynamically create code. The namespace that contains the classes necessary to do this is the Reflection.Emit namespace. Dynamic generation of code can get very complicated, so you should be careful about using this ability, but you can do some incredible things with it. Imagine if you have an application that performs complex calculations, but you do not necessarily know what all of those calculations are beforehand. Say you have given the users the ability to create calculations later and specify how the application processes the calculation. To do this you might take a calculation and dynamically generate the code needed to process the calculation, and then you gain the ability for an application to be expanded without any additional coding by a developer!

I do not expect this ability to catch on overnight because it is a highly complex area of development. To develop code using Reflection.Emit, you need to know a great deal about the Microsoft Intermediate Language (MSIL). An excellent book on the subject is *Compiling for the .NET Common Language Runtime* by John Gough (Prentice Hall, 2001).

Understanding Attribute Classes

The root of all reflection is the System.Attribute class. An attribute class provides information about a coding construct. So what is an attribute class? An *attribute class* is a class you can create and, for lack of a better description, "attach" to anything. You could attach them to a class, property, method, structure, enum, and so on. You can designate properties that only allow an attribute class to be attached to certain types of code structures or to everything—it is completely up to you. And how does this help you? It allows you to describe, or give additional properties to, a specific piece of code. Using reflection, you can examine these classes to learn information about your code elements.

NOTE Attribute classes are passive. That is, they are compiled into the code, and they cannot *react* to changes in data—they can only examine the data after the fact. So, if you need to stop a property from being changed unless it follows certain rules (as opposed to changing the value and marking it as a broken rule), you need to use a combination of attribute classes and business rule checking as shown in earlier chapters.

Used properly, reflection can make classes more flexible and more reusable. It can also give you the ability to dynamically generate information based on your classes. The example you will see first demonstrates that ability. After you have worked through this example, you will probably find some creative ways to use this unique and awesome ability of the .NET Framework.

Setting Up the Scenario

Let's say, for the sake of argument, that you have a bunch of classes you want to display in forms that contain listviews. And let's say that you only want to write the load routine once so that it can be used on all of the list forms in such a way that each developer does not have to come up with their own code to create and fill that list form. Does this sound familiar? If you have worked through the first nine chapters of this book, you will understand this scenario. You had to code each of the load listview routines. Now, expand that out by 20 or 30 forms.... Extrapolating out what you have created so far, each load list routine minus comments and empty lines, is approximately 30 lines of code. If you have 20 forms in an application that do the same thing, that is 600 lines of code that can be removed from the application if you implement this using reflection! And let me say that this is only the tip of the iceberg.

CAUTION After seeing what you can do with this ability, you might be encouraged to start doing everything with reflection—do not. Reflection is great for certain tasks, but for other tasks it is a great deal more work than it is worth. Also, planning these types of classes correctly takes time. So, before you decide to start implementing attribute classes all over the place, think about the complexity and maintainability of the application. In general, reflection allows you to consume classes by using generic routines that do not have to be customized for each implementation.

The way this attribute class works is the following: You will attach an attribute class to each property you want to display in the listview. The listview load method looks in your class and determines which properties to add to the listview as columns. It creates those columns and then adds the values of the object to the listview. Once you have created this method the first time, it is a cinch to reuse it.

Setting Up the Project

Create a new Windows application and call it *ClassAttributeDemo*. Add a new class to the project and call it *ListAttributes*. On the default form that is created, add the controls and set the properties as shown in Table 10-1.

Table 10-1. Form Controls for the ClassAttributeDemo Application

Control	**Name**	**Property**	**Value**
Form1	frmList	Text	Class Attribute Demo
Listview	lvwList	View	Details
Button	btnComputers	Text	Computer List
Button	btnBooks	Text	Book List

When you are done, the form should look like Figure 10-1.

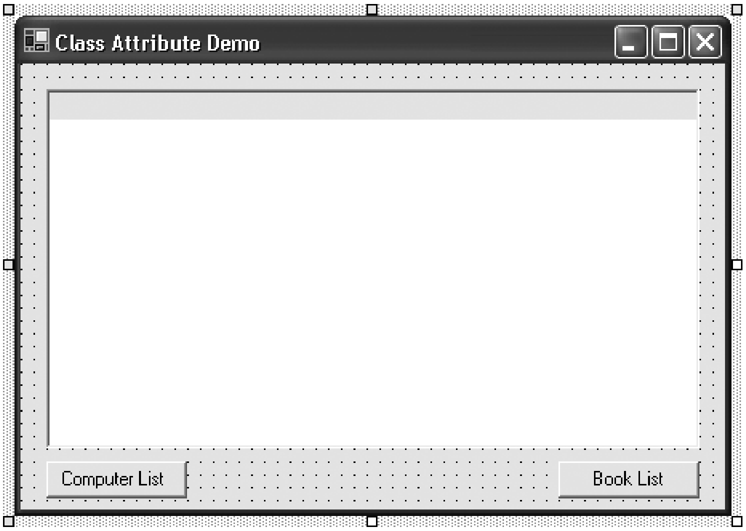


Figure 10-1. The ClassAttributeDemo application

Creating the Attribute Class

Open up the ListAttribute code module and alter the class signature so that it reads as follows:

```
<AttributeUsage(AttributeTargets.Property)> _
Public Class ListAttribute
     Inherits System.Attribute

End Class
```

The AttributeUsage tag turns a class into an attribute class. The enumerated value, AttributeTargets, allows you to specify what type of code block can be the target of this class. The valid values are as follows:

All	Interface
Assembly	Method
Class	Module
Constructor	Parameter
Delegate	Property
Enum	ReturnValue
Event	Struct
Field	

In this case, you are saying that this class can only be applied to a property. Notice also that your class inherits from the System.Attribute class. This marks your class as an attribute in .NET.

NOTE By convention, all attribute classes should have a suffix of *Attribute*, but when you associate them with a method, you do not have to specify the word *Attribute*. You will see an example of this in Listing 10-2.

So, what properties would you need to do what you want to do? Well, you really only need two properties: one to hold the header text and the other to hold the order in which the properties get added to the listview. So, to do this, let's add the following properties and constructor to the ListAttribute class:

```
Public Heading As String
Public Column As Integer

Public Sub New(ByVal Header As String, ByVal Col As Integer)
    Heading = Header
    Column = Col
End Sub
```

That was pretty easy—you are done with your ListAttribute class. You can see that these classes can be easy to create. Or, as with an attribute class that you have already seen—the SerializableAttribute class used for your BusinessErrors class—they can be very complex.

Creating the ComputerList Class

Add a new class to the project and call it *ComputerList*. To keep this simple, use the code in Listing 10-1 for the class.

Listing 10-1. The ComputerList Class

```
Public Class ComputerList
    Private mstrName As String
    Private mstrProc As String
    Private mdblSpeed As Double
    Private mdblPrice As Double
    Private mstrManuf As String

    Public ReadOnly Property Proc() As String
        Get
            Return mstrProc
        End Get
    End Property

    Public ReadOnly Property Speed() As Double
        Get
            Return mdblSpeed
        End Get
    End Property

    Public ReadOnly Property Cname() As String
        Get
            Return mstrName
        End Get
    End Property
    Public ReadOnly Property Price() As Double
        Get
            Return mdblPrice
        End Get
    End Property

    Public ReadOnly Property Manufacturer() As String
        Get
            Return mstrManuf
        End Get
    End Property

    Public Sub New(ByVal Name As String, ByVal Process As String, _
    ByVal Sp As Double, ByVal Pr As Double, ByVal Man As String)
        mstrName = Name
        mstrProc = Process
        mdblSpeed = Sp
        mdblPrice = Pr
        mstrManuf = Man
    End Sub
End Class
```

This simple class has five private variables and five public read-only variables. The variables all get set in the constructor. Next, you are going to tag three of the properties with your ListAttribute class so that only those three properties show up in the list. Add a List tag in front of the properties Cname, Process, and Speed so that each property looks like that in Listing 10-2.

Listing 10-2. Three Properties with the ListAttribute Applied

```
<List("Processor", 1)> Public ReadOnly Property Proc() As String
     Get
          Return mstrProc
     End Get
End Property

<List("Speed", 2)> Public ReadOnly Property Speed() As Double
     Get
          Return mdblSpeed
     End Get
End Property
```

```
<List("Computer Name", 0)> Public ReadOnly Property Cname() As String
     Get
          Return mstrName
     End Get
End Property
```

You will note that when you open the attribute tag (<) only the word *List* appears in the list of available attributes not the whole class name, *ListAttribute*. As mentioned previously, the word *Attribute* is dropped from the end of the attribute class. That is all you need to do to set up the ComputerList class. Now you need to create a collection class to hold a couple of values.

Create a class (in the same code module as the ComputerList class) called *ComputerListMgr* that inherits from the CollectionBase class. Use the code in Listing 10-3.

Listing 10-3. The ComputerListMgr Class

```
Public Class ComputerListMgr
    Inherits System.Collections.CollectionBase

    Public Sub Add(ByVal obj As ComputerList)
        list.Add(obj)
    End Sub

    Public Sub Remove(ByVal Index As Integer)
        list.RemoveAt(Index)
    End Sub

    Public Function Item(ByVal Index As Integer) As ComputerList
        Return CType(list.Item(Index), ComputerList)
    End Function
End Class
```

Examining Property Attributes in Code

Before getting into examining property attributes in code, you need to be aware that you cannot set Option Strict to On.

CAUTION To elaborate, you cannot set Option Strict to On in the code module where you process the property attributes. This is because the type of reflection you are performing requires late binding. In general, this is not a good practice, but it is acceptable in this situation.

Now, having said that, let's start coding. Go into the form code module and import the System.Reflection namespace. Next, add a module-level variable for the ComputerListMgr as follows:

```
Private mobjCLMgr As ComputerListMgr
```

Add the code in Listing 10-4 to the frmList class.

Listing 10-4. The btnComputers_Click Method

```
Private Sub btnComputers_Click(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles btnComputers.Click
     Dim t As Type = GetType(ComputerList)
     mobjCLMgr = New ComputerListMgr()

     mobjCLMgr.Add(New ComputerList("Lightning", "P4", 1.4, 699.0, "Dell"))
     mobjCLMgr.Add(New ComputerList("Thunder", "P4", 1.7, 799.0, "Dell"))
     LoadList(t, CType(mobjCLMgr, CollectionBase))
End Sub
```

The first line of this code gets the type of the ComputerList class and stores it in a type variable. Next you instantiate the computer list manager and add two items to the collection. Finally you call the LoadList method (which you will code next) and pass in your object type and your collection. Notice, however, that you are passing your manager in as a CollectionBase object. In a real application, you would probably want to overload this method to accept virtually any type of collection. The reason why you are passing in generic values is so that the LoadList method remains as flexible as possible. Listing 10-5 contains the code for the LoadList method. Do not panic! You will get an explanation of everything line by line after the listing.

Listing 10-5. The LoadList Method

```
Private Sub LoadList(ByVal t As Type, ByRef col As CollectionBase)
     Dim p As PropertyInfo()
     Dim i, j As Integer
     Dim SortedL As New Collections.SortedList()

     lvwList.Clear

     p = t.GetProperties(BindingFlags.Public Or BindingFlags.Instance)

     For i = 0 To p.Length - 1
          Dim a As Object()
          a = p(i).GetCustomAttributes(False)
          If a.Length > 0 Then
               For j = 0 To a.Length - 1
                    If a(j).GetType Is GetType(ListAttribute) Then
                         Dim la As ListAttribute = CType(a(j), ListAttribute)
                         SortedL.Add(la.Column, p(i))
                    End If
               Next
          End If
     Next

     For i = 0 To SortedL.Count - 1
          Dim pi As PropertyInfo = CType(SortedL.Item(i), PropertyInfo)
          Dim a As Object = pi.GetCustomAttributes(False)
          For j = 0 To a.Length - 1
               If a(j).GetType Is GetType(ListAttribute) Then
                    Dim la As ListAttribute = CType(a(j), ListAttribute)
                    lvwList.Columns.Add(la.Heading, _
                    lvwList.Width / SortedL.Count - 2, _
                    HorizontalAlignment.Left)
                    Exit For
               End If
          Next
     Next

     Dim obj As Object
     Dim myObject() As Object
     For Each obj In col
          Dim cl As Object = Convert.ChangeType(obj, t)
          Dim k As Integer
          Dim lst As New ListViewItem()
          For i = 0 To SortedL.Count - 1
               Dim pr As PropertyInfo = CType(SortedL.Item(i), PropertyInfo)
               Dim strValue As String = ""
               strValue = Convert.ToString(t.InvokeMember(pr.Name, _
               BindingFlags.GetProperty, Nothing, cl, myObject))
               If i = 0 Then
                    lst.SubItems(i).Text = strValue
               Else
                    lst.SubItems.Add(strValue)
               End If
          Next
          lvwList.Items.Add(lst)
     Next
End Sub
```

So now that you think you may be lost, let's try to straighten everything out and explain what is going on here. The first line declares an array of PropertyInfo variables. The PropertyInfo type holds information about—you guessed it—properties. The i and j variables are just counter variables. The SortedL variable stores the properties in the order you have specified they be displayed in (by way of your attribute settings in the ComputerList class). You will see this in action in a minute. Then you clear the listview of all of its current contents—headers and all:

```
Dim p As PropertyInfo()
Dim i, j As Integer
Dim SortedL As New Collections.SortedList()

lvwList.Clear
```

This next line calls the GetProperties method on your type variable. So this line reads, "Get all of the properties of the type (in this case, the ComputerList class) that are public or instance properties." This method returns an array of PropertyInfo types:

```
p = t.GetProperties(BindingFlags.Public Or BindingFlags.Instance)
```

The next block of code continues your process of discovering information about the properties. First, you start by looping through the array of PropertyInfo values:

```
For i = 0 To p.Length - 1
```

The variable a is an object array to hold all of the custom attributes on the specific property:

```
Dim a As Object()
```

The GetCustomAttributes returns an object array because there may be several types of attributes associated with the property you are examining. The False parameter indicates that you do not want to look at any other property values in the inheritance chain for this class:

```
a = p(i).GetCustomAttributes(False)
```

Now you check the length of the array to see if there were any custom attributes associated with the property. Remember, for your class there are only three: the Cname, Proc, and Speed properties:

```
If a.Length > 0 Then
```

If it does find at least one custom attribute, you loop through the array of custom attributes:

```
For j = 0 To a.Length - 1
```

Here you check the type of custom attribute. This is the only known type in the entire method:

```
If a(j).GetType Is GetType(ListAttribute) Then
```

If the custom attribute is of type ListAttribute, then you convert that custom attribute into a value that you can manipulate easily by performing a ctype on it:

```
Dim la As ListAttribute = CType(a(j), ListAttribute)
```

Finally, you add the column number as the key in the sorted list, and you add the PropertyInfo variable as the object in the sorted list so you can reference it later:

```
SortedL.Add(la.Column, p(i))
```

The next block of code adds the column headers to the listview. You start by looping through the sorted list collection and retrieving the PropertyInfo objects. Then you get the custom attributes of the property. Next, you again loop through the custom attributes looking for the ListAttribute. When you find it, you convert it into a ListAttribute variable and extract the heading name. When this block of code finishes executing, the column headers will have been added to the listview:

```
For i = 0 To SortedL.Count - 1
     Dim pi As PropertyInfo = CType(SortedL.Item(i), PropertyInfo)
     Dim a As Object = pi.GetCustomAttributes(False)
     For j = 0 To a.Length - 1
          If a(j).GetType Is GetType(ListAttribute) Then
               Dim la As ListAttribute = CType(a(j), ListAttribute)
               lvwList.Columns.Add(la.Heading, _
               lvwList.Width / SortedL.Count - 2, _
               HorizontalAlignment.Left)
               Exit For
          End If
     Next
Next
```

The obj variable helps you iterate through the ComputerListMgr collection. Because you only know that this is a collection, you cannot use a For Next loop to iterate through the collection. You can only use the For Each enumeration. And because you do not know what type of object is returned to you by the collection (remember, this is a wholly generic routine, so you cannot declare a variable of type ComputerList anywhere), you need to use an object variable. The myObject object array is used as a parameter to the InvokeMethod call. It is a throwaway variable:

```
Dim obj As Object
Dim myObject() As Object
```

Finally, you get to the block of code that adds the values from the collection into the listview. Before you start examining this block of code, think about what it is doing. You are taking a collection that you know nothing about, that stores objects you know nothing about, and that has properties you know nothing about and extracting that data and placing it in a listview! This block of code, in a nutshell, shows exactly how powerful the .NET Framework can be when used to its fullest potential.

Let's now look at what is happening here. The For Each statement, as mentioned earlier, is the only way to iterate through your collection object:

```
For Each obj In col
```

The cl variable is an object that you are converting to the type you have passed in to the method—in this case, the ComputerList type. You do this using the ChangeType method of the Convert class. This is an example of late binding and the chief reason you cannot use Option Strict On in this code module:

```
Dim cl As Object = Convert.ChangeType(obj, t)
```

The k variable is just a counter variable, and lst is the listviewitem you will be adding to the listview:

```
Dim k As Integer
Dim lst As New ListViewItem()
```

Next you loop through the sorted list collection (yet again) to get the properties for which you need to retrieve the values:

```
For i = 0 To SortedL.Count - 1
```

This line retrieves the PropertyInfo from the sorted list collection:

```
Dim pr As PropertyInfo = CType(SortedL.Item(i), PropertyInfo)
```

StrValue holds the value you retrieve from whatever property you are calling. It is initialized to an empty string because you may have a property that was not set and this would leave strValue with a value of nothing, which you absolutely do not want:

```
Dim strValue As String = ""
```

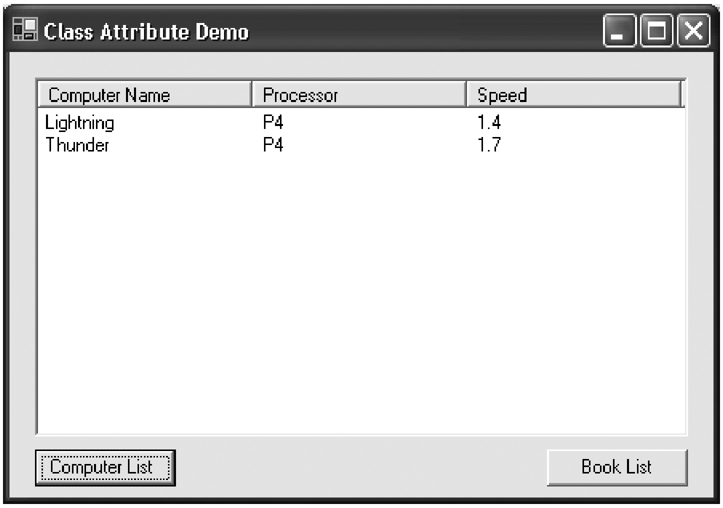
This next line is the workhorse of this method. This line says the following: "Call the method whose name is returned by PropertyInfo variable (pr). Look for this method in the class's collection of properties using the default binder (do not worry about what this is right now, for more information check the MSDN documentation). Call this method on the given object (cl) with the parameters given in myObject and store the return value in the string variable strValue." That was a handful to say the least. The myObject array would, if you were calling a method that required parameters to be passed to it, contain a list of values to pass in to the method:

```
strValue = Convert.ToString(t.InvokeMember(pr.Name, _
BindingFlags.GetProperty, Nothing, cl, myObject))
```

If this is the first time through the loop, assign the value to item 0 of the sub-items collection (because you already instantiated the lst variable previously); otherwise, add a new subitem to the listviewitem. Finally, add the listviewitem to the listview:

```
     If i = 0 Then
          lst.SubItems(i).Text = strValue
     Else
          lst.SubItems.Add(strValue)
     End If
Next
lvwList.Items.Add(lst)
```

Now, if you have not done so yet, run the application and click the Computer List button. The result should look something like Figure 10-2.



```
Computer Name    Processor    Speed
Lightning        P4           1.4
Thunder          P4           1.7
```

Figure 10-2. List of computers displayed by the LoadList method

Now, as a test, edit the ComputerList class and change the order you would like things to display on the screen (by changing the numeric value in the List Attribute tag) and run the application again. Pretty neat, huh?

Listing 10-6 contains the code for the BookList class and the BookListMgr class. They are the same as what you have just done, but they have different properties.

Listing 10-6. The BookList and BookListMgr Classes

```
Public Class BookList
    Private mstrTitle As String
    Private mstrAuthor As String
    Private mdblPrice As Double
    Private mstrPublisher As String

    Public ReadOnly Property Price() As Double
        Get
            Return mdblPrice
        End Get
    End Property

    Public ReadOnly Property Publisher() As String
        Get
            Return mstrPublisher
        End Get
    End Property

    <List("Author", 1)> Public ReadOnly Property Author() As String
        Get
            Return mstrAuthor
        End Get
    End Property

    <List("Book Title", 0)> Public ReadOnly Property Title() As String
        Get
            Return mstrTitle
        End Get
    End Property

    Public Sub New(ByVal sTitle As String, ByVal sAuthor As String, _
    ByVal dPrice As Double, ByVal sPub As String)
        mstrTitle = sTitle
        mstrAuthor = sAuthor
        mdblPrice = dPrice
        mstrPublisher = sPub
    End Sub

End Class
```

```
Public Class BookListMgr
    Inherits System.Collections.CollectionBase

    Public Sub Add(ByVal obj As BookList)
        list.Add(obj)
    End Sub

    Public Sub Remove(ByVal Index As Integer)
        list.RemoveAt(Index)
    End Sub

    Public Function Item(ByVal Index As Integer) As BookList
        Return CType(list.Item(Index), BookList)
    End Function
End Class
```

Next, add the following module-level declaration in frmList:

```
Private mobjBKMgr As BookListMgr
```

Finally, Listing 10-7 shows the code for the btnBooks_Click method.

Listing 10-7. The btnBooks_Click Method

```
Private Sub btnBooks_Click(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles btnBooks.Click
     Dim t As Type = GetType(BookList)
     mobjBKMgr = New BookListMgr()

     mobjBKMgr.Add(New BookList("Life With .NET", "Anonymous", 49.95, _
     "Apress"))
     mobjBKMgr.Add(New BookList("Life With Java", "Unknown", 19.95, _
     "ABC Publishing"))
     LoadList(t, CType(mobjBKMgr, CollectionBase))
End Sub
```

Now try running the application and clicking either button. Try changing the methods with which the custom attributes are associated. No matter what you do, this code will work.

Implementing Business Rules Using Custom Attributes

To work through this example, you will need to complete the coding up through the first part of Chapter 8, “Reusing Code.” When you are done with this example you will have created a set of classes that you can reuse in your own projects to implement business rules.

NOTE I found the basis for this code online at the Newtelligence AG company (`http://www.newtelligence.com`), which built this code in C# as the basis for a Web security interface application. I converted this code to Visual Basic and enhanced it to fit within the framework of the application you have been creating in this book. The code from AG New Intelligencer was developed under a BSD-style license and is used here with the author’s permission (Clemens F. Vasters, who can be reached at `clemensv@newtelligence.com`). Although the code presented here is different from the original code, the implementation of this idea came from the original code.

You can extend this small amount of code, which needs to be written only once, to fit virtually any type of business rule that you may need to create.

NOTE When I originally came across this code, I was thinking of having a class of business rules usable by an entire organization. In this manner, no one in an organization would ever need to build the basic set of business rules ever again, and everyone would have access to these rules.

Creating the BusinessRules Project

You are going to create a separate project to hold all of the business rule attributes and the validation routines. In this way you can distribute the rules to other applications. This project will be another shared project that must exist on both the client and the server. The reason for this is that the class attributes you create will be used in both the data-centric and user-centric objects.

To begin, open the current Northwind solution, add a new Class Library project to the solution, and call it *BusinessRules*. Rename the Class1.vb file that is created by default to *Attributes.vb*. Then delete the default class definition that was created in this code module. You will create the business rule attributes and the necessary interface in this code module. You will eventually create another set of classes to check the business rules specified by the attributes.

Add the following code to the Attributes code module:

```
Option Explicit On
Option Strict On

Imports System.Reflection

Namespace Attributes

End Namespace
```

The interface and all of the classes you create will be created in the Attributes namespace. Before you begin adding classes, let's review the business rules in place in the RegionDC class:

- RegionDescription cannot be null.
- RegionDescription cannot be a zero-length string.
- RegionDescription cannot be more than 50 characters in length.

This gives you the basis for creating your first set of class attributes.

NOTE These are the only attributes you will be creating for this project; however, in the code available for download, there are a considerable number of additional business rule attribute classes.

Going by this list of rules, you need to create three attribute classes that check for the following: a null value, an empty length string, and a maximum number of characters.

Creating the ITest Interface

Before creating the classes, you need to create an interface that all of your classes will support.

NOTE You need the interface because these are all generic classes. When you code the routines that check the rules, you will see that you do not care what the attribute class is, only that it is a rule and that you need to check the rule. In this way, you can continue to add additional business rules without once having to change the way in which you check the rules.

Add the code for the ITest interface as shown in Listing 10-8 to the Attributes namespace in the Attributes code module.

Listing 10-8. The ITest Interface

```
Public Interface ITest
     Function TestCondition(ByVal Value As Object, ByRef cls As Object) As Boolean
     Function GetRule() As String
End Interface
```

The TestCondition method actually determines if the value has broken the specific business rule. It accepts the value stored in the field or property and the object in which the property resides. This is enough information for a method to determine everything about a given class. It returns a value of True if the rule has been broken and a value of False if the rule has not been broken. The GetRule method simply returns a string that describes the rule in plain English. This will be used (in conjunction with another method) to eliminate the need for all of the code in the GetBusinessRules method.

Creating the NotNullAttribute Class

Now you can create the first business rule attribute class: NotNullAttribute. Add the code for the NotNullAttribute class as shown in Listing 10-9.

Listing 10-9. The NotNullAttribute Class

```
<AttributeUsage(AttributeTargets.Field Or AttributeTargets.Property)> _
Public Class NotNullAttribute
     Inherits System.Attribute

     Implements ITest
     Public Function TestCondition(ByVal Value As Object, ByRef cls As Object) _
     As Boolean Implements ITest.TestCondition
          If Value Is Nothing Then
               Return True
          Else
               If IsNumeric(Value) Then
                    If Convert.ToDecimal(Value) = 0 Then
                         Return True
                    End If
               End If
               Return False
          End If
     End Function

     Public Function GetRule() As String Implements ITest.GetRule
          Return "Value cannot be null."
     End Function
End Class
```

Let's examine this code to determine exactly what is happening. The signature tells you that this class can only be applied to a field or property within a class.

```
<AttributeUsage(AttributeTargets.Field Or AttributeTargets.Property)> _
Public Class NotNullAttribute
```

As before, all classes that are attribute classes must inherit from the System.Attribute class. Next, your class implements the ITest interface as will all of your attribute classes. Now you come to the TestCondition method, which does the real work of the class. This first check just tests to see if the value is null; if it is, it returns True and the method ends:

```
If Value Is Nothing Then
     Return True
Else
```

The second part of this routine may or may not be controversial. Because numbers are not nullable, when they are instantiated they are initialized with a value of zero. If a numeric value can be a zero, you should not apply this attribute to it because this attribute is supposed to deal with nulls and is named accordingly, but for simplicity it is useful to keep it in this class. You can always create a separate class called *ValueNotZeroAttribute* and add this code into it—the choice is yours. This code checks to see if the value is numeric, and if it is, it checks to see if the value is equal to zero:

```
If IsNumeric(Value) Then
     If Convert.ToDecimal(Value) = 0 Then
          Return True
     End If
End If
Return False
```

The last method in the class, the GetRule method, simply returns what the rule for the property is.

Creating the DisplayNameAttribute Class

Now, you have one small problem here—how do you show the property to the user in a way that looks nice to the user? If you go by just the name of the property, it is going to look ugly because there are no spaces and sometimes property names do not reflect what the user sees on the screen. To overcome this you are going to add another class called *DisplayNameAttribute* that will store the name for the property you want to show the user.

Add the DisplayNameAttribute class as shown in Listing 10-10.

Listing 10-10. The DisplayNameAttribute Class

```
<AttributeUsage(AttributeTargets.Field Or AttributeTargets.Property)> _
Public Class DisplayNameAttribute
     Inherits System.Attribute

     Private _strValue As String

     Public Sub New(ByVal Value As String)
          _strValue = Value
     End Sub

     Public ReadOnly Property Name() As String
          Get
               Return _strValue
          End Get
     End Property
End Class
```

Creating the NotEmptyAttribute Class

Now you will create the rule that will check to make sure that a string value is not empty. Listing 10-11 shows the code for this class.

Listing 10-11. The NotEmptyAttribute Class

```
<AttributeUsage(AttributeTargets.Field Or AttributeTargets.Property)> _
Public Class NotEmptyAttribute
     Inherits System.Attribute

     Implements ITest

     Public Function TestCondition(ByVal Value As Object, ByRef cls As Object) _
     As Boolean Implements ITest.TestCondition
          If Value Is Nothing Then
               Return True
          Else
               Dim str As String = CType(Value, String)
               If str.Trim.Length = 0 Then
                    Return True
               Else
                    Return False
               End If
          End If
     End Function

     Public Function GetRule() As String Implements ITest.GetRule
          Return "Value cannot be a zero length string."
     End Function
End Class
```

Everything that is occurring in this class should be straightforward except for the check to see if the value is nothing. This check must be made in some form or another in every class that checks a property. After all, how can you check the value of something if the value is nothing? Notice also how similar this is to the first attribute class you created. The beauty of creating rules this way is that the code is compact, easy to understand, and even easier to debug. And once you get it right here, you never need to check it again or write code to perform the same type of validation.

Creating the MaxLengthAttribute Class

This last attribute class is substantially identical to the previous two business rule attributes that you created. Listing 10-12 presents the code for this class.

Listing 10-12. The MaxLengthAttribute Class

```
<AttributeUsage(AttributeTargets.Field Or AttributeTargets.Property)> _
Public Class MaxLengthAttribute
    Inherits System.Attribute

    Implements ITest

    Private _intValue As Integer

    Public Sub New(ByVal Value As Integer)
        _intValue = Value
    End Sub

    Public Function TestCondition(ByVal Value As Object, ByRef cls As Object) _
    As Boolean Implements ITest.TestCondition
        Dim strValue As String = Convert.ToString(Value)

        If strValue.Length > _intValue Then
            Return True
        Else
            Return False
        End If
    End Function

    Public Function GetRule() As String Implements ITest.GetRule
        Return "Value cannot be longer than " & _intValue & " characters."
    End Function
End Class
```

This class simply checks the length of a string value to determine if it has more characters than allowed. Notice that your GetRule method now incorporates the value that you set into the string that is returned. In this case, the RegionDescription property would return a rule that said, “Region Description cannot be longer than 50 characters.”

Now that you have all of the business rules in place, you can apply them to your object.

Assigning Data-Centric Business Rule Attributes

To begin, right-click the NorthwindDC references node and select Add Reference. From the Projects tab, select the BusinessRules project by double-clicking it and then click OK. Next, switch to the RegionDC class, expand the Public Attributes region, and delete the public RegionDescription property. In a single move you have eliminated 19 lines of code from your project (yes, it is at the expense of adding all of the code for the business rule attributes, but think about it, you never need to add them again and this is not all of the code you will delete). Expand the Private Attributes region and change the mstrRegionDescription variable to the following:

```
Public RegionDescription As String
```

Next, go to the Save method and change the mstrRegionDescription variable to *RegionDescription*. Before you apply the attributes, you have to import the BusinessRules.Attributes namespace; once that is done you can start applying attributes. So, add the line to perform this to the top of the RegionDC module.

Now you need to apply the attributes. Change the RegionDescription declaration line so that it reads as follows:

```
<DisplayName("Region Description"), NotNull(), NotEmpty(), MaxLength(50)> _
Public RegionDescription As String
```

It may be anticlimactic, but in reality an easy-to-maintain system does not throw many complicated surprises at you! These three tags tell your class what the value cannot be. The DisplayName attribute tells you the human readable name you will display to the user.

Retrieving the List of Business Rules

Before you get into retrieving the business rules, you need to make one change to your application. You need to move your BusinessErrors class and your structErrors structure to the BusinessRules project. The reason you need to do this is so that your new attribute class is modular and can be used by other applications. Follow these steps to accomplish this:

1. Add a new class module to the BusinessRules project called *Errors.vb*.

2. Delete the default class that is created in this code module.

3. Add a namespace in the Errors code module called *Errors* (this will now be referenced by using BusinessRules.Errors).

4. Cut the BusinessErrors class from the NorthwindShared Errors code module and paste it into the Errors namespace in the BusinessRules project.

5. Cut the structErrors structure from the NorthwindShared Structures code module and paste it into the Errors namespace in the BusinessRules project.

6. Add a reference to the BusinessRules project in the NorthwindUC, NorthwindShared, and NorthwindTraders projects.

7. In each edit form code module, all the data-centric and all the user-centric code modules, as well as the NorthwindShared Interfaces code module and the frmBusinessRules module, replace northwindshared.errors and northwindtraders.northwindshared.errors with businessrules.errors.

As you perform steps 4 and 5 of this list, you will see several errors in the Task List. Not to worry, though—once you are through with the last step, you will not have any errors. That was the hardest part. The next step is to add a new class code module to the BusinessRules project called *Validate*. Once you have done that, delete the default class and add the following code to the Validate class module:

```
Option Explicit On
Option Strict On

Imports BusinessRules.Attributes
Imports BusinessRules.Errors
Imports System.Reflection

Namespace Validate
    Public Class Validation

    End Class
End Namespace
```

The Validation class is the only class you are going to create in this namespace. Now, add the GetDisplayName method to the Validation class as shown in Listing 10-13.

Listing 10-13. The GetDisplayName Method

```
Private Shared Function GetDisplayName(ByVal member As MemberInfo) As String
     Dim obj() As Object = member.GetCustomAttributes(True)
     Dim i As Integer

     If obj.Length > 0 Then
          For i = 0 To obj.Length - 1
               If TypeOf (obj(i)) Is DisplayNameAttribute Then
                    Dim objDisplayNameAttribute As DisplayNameAttribute = _
                   CType(obj(i), DisplayNameAttribute)
                    Return objDisplayNameAttribute.Name
               End If
          Next i
     End If
     Return member.Name
End Function
```

Let's examine what this code does, line by line. One important thing to note is the method signature. This is a shared method because it will be called by a shared method. In fact, all of the methods in this class will be shared methods so this class never has to be instantiated. This provides you with immense speed benefits (especially when you move to the user-centric classes), and because this class maintains no state at all, it is OK to do.

The first line retrieves a list of all of the custom attributes associated with the class member. The True parameter tells the code to retrieve all of the custom attributes along the entire inheritance chain:

```
Dim obj() As Object = member.GetCustomAttributes(True)
```

Next, you check to see if there are any custom attributes:

```
If obj.Length > 0 Then
```

If there are, you loop through them looking for an attribute that is a DisplayNameAttribute:

```
If TypeOf (obj(i)) Is DisplayNameAttribute Then
```

If we find one, you convert it to a true DisplayNameAttribute object and you return the value of the Name property:

```
Dim objDisplayNameAttribute As DisplayNameAttribute = CType(obj(i), _
DisplayNameAttribute)
Return objDisplayNameAttribute.Name
```

Finally, if there was no Display Name attribute found, you simply return the name of the property. Now that you can return the display name, it is time to be able to return the rules.

Add the code for the GetBusinessRules method to the Validate class as shown in Listing 10-14.

Listing 10-14. The GetBusinessRules Method

```
Public Shared Function GetBusinessRules(ByVal cls As Object) As BusinessErrors
     Dim t As Type = cls.GetType
     Dim m As MemberInfo() = t.GetMembers
     Dim i As Integer

     Dim objBusErr As New BusinessErrors

     For i = 0 To m.Length - 1
          Dim obj() As Object = m(i).GetCustomAttributes(True)
          If obj.Length > 0 Then
               Dim j As Integer
               For j = 0 To obj.Length - 1
                    If TypeOf obj(j) Is ITest Then
                         Dim objI As ITest = CType(obj(j), ITest)
                         objBusErr.Add(GetDisplayName(m(i)), objI.GetRule)
                    End If
               Next
          End If
     Next

     Return objBusErr
End Function
```

This is the first routine where you access the ITest interface, so you will see the workings of this method line by line. The method accepts an object, which is the class you want to get the business rules from, and it returns a BusinessErrors object. The first line retrieves all of the type information about the class. The second line retrieves all of the members of the class and stores them in an array of MemberInfo objects:

```
Public Shared Function GetBusinessRules(ByVal cls As Object) As BusinessErrors
     Dim t As Type = cls.GetType
     Dim m As MemberInfo() = t.GetMembers
```

Next you loop through all of the members of the class and you retrieve the custom attributes of each member (you retrieve all of the custom attributes along the inheritance chain). This allows your inherited classes to use the business rules of any base classes. There may be a point at which this is not desirable, though, so you may need to modify this code to suit your particular needs. Then you check to see if there were in fact any custom attributes retrieved from the member:

```
For i = 0 To m.Length - 1
     Dim obj() As Object = m(i).GetCustomAttributes(True)
     If obj.Length > 0 Then
```

Finally, you loop through all of the custom attributes associated with the member. You check to see if the custom attribute implements the ITest interface, and, if it does, you call the GetRule method on it to retrieve the rule. You also extract the Display Name from the member and add them both to the BusinessErrors object:

```
For j = 0 To obj.Length - 1
     If TypeOf obj(j) Is ITest Then
          Dim objI As ITest = CType(obj(j), ITest)
          objBusErr.Add(GetDisplayName(m(i)), objI.GetRule)
     End If
Next
```

Now that you have added this method, let's implement it. In the RegionDC class, add the following declaration:

```
Private mobjVal As BusinessRules.Validate.Validation
```

Alter the GetBusinessRules method so that it now reads as follows:

```
Public Function GetBusinessRules() As BusinessErrors _
Implements IRegion.GetBusinessRules
        Return mobjVal.GetBusinessRules(Me)
End Function
```

Now, build the application, but when you go to deploy the remote assemblies, be sure to deploy all three assemblies: NorthwindDC, NorthwindShared, and BusinessRules. Then run the application, go to Maintenance ➤ Regions, and select one of the existing regions to edit (or select the Add button). Then click the Rules button, and you should see the rules screen with your three business rules. Any changes you make to the business rules will be automatically reflected the next time the application is compiled and run, and you never need to change the GetBusinessRules method again.

Checking Business Rules with Custom Attributes

Now that you can retrieve the business rules, it is time to put them to their real use—constraining data. You will eventually end up writing two different methods to perform this task: one for the data-centric classes and one for the user-centric classes. They are different methods because they do the task in slightly different ways. However, you are only going to worry about the data-centric classes right now. You are going to add a new method to the Validation class (shown in Listing 10-15). This method is a little more involved than the GetBusinessRules method because you have to take into account the differences between fields and properties; but for the most part there are not a lot of differences between this method and the GetBusinessRules method.

Listing 10-15. The Validate Method

```
Public Shared Function Validate(ByVal cls As Object) As BusinessErrors
     Dim t As Type = cls.GetType
     Dim i, j As Integer
     Dim bln As Boolean

     Dim objBusErr As New BusinessErrors

     Dim m As MemberInfo() = t.GetMembers

     For i = 0 To m.Length - 1
          Dim objAttrib() As Object = m(i).GetCustomAttributes(True)
          For j = 0 To objAttrib.Length - 1
               If TypeOf objAttrib(j) Is ITest Then
                    Dim objI As ITest = CType(objAttrib(j), ITest)

                    If TypeOf m(i) Is FieldInfo Then
                         Dim fld As FieldInfo = CType(m(i), FieldInfo)
                         bln = objI.TestCondition(fld.GetValue(cls), cls)
                    End If
                    If TypeOf m(i) Is PropertyInfo Then
                        Dim pro As PropertyInfo = CType(m(i), PropertyInfo)
                        bln = objI.TestCondition(pro.GetValue(cls, Nothing), _
                        cls)
                    End If

                    If bln Then
                        objBusErr.Add(m(i).Name, objI.GetRule())
                    End If
                End If
            Next
        Next

        Return objBusErr
    End Function
```

The real difference in this listing is the test to determine if the member is a field or a property. The reason for this test is that the methods for retrieving the instance values are different for each type. This is because a property is a method, so it does not just retrieve a value; it actually invokes the method to return a value. You then call the TestCondition method, and if the value breaks the rule you add it to the BusinessError method.

One thing to note about this method of validating business rules is that *all* of the rules for each property will be checked as opposed to what you had before. Before only one rule at a time was being checked and thrown as an error. So this method provides you with a little more robust business rule handling and reporting.

Code Reduction Metrics

Many companies today are trying to take a cost-conscious approach to coding—so the first question asked when faced with a new technology is, "How much effort (read: money) will this save and how much easier is it to maintain?" If anyone was looking for a justification to use the .NET Framework, this is it.

On average, you can assume that you will save six lines of code for every business rule that is checked via a custom attribute as opposed to the previous method you were using. This should be able to help you extrapolate out the cost in savings by using custom attributes. In your RegionDC class, for example, you originally had 20 lines of code for the public RegionDescription property, one line for the private RegionDescription field, and nine lines of code for the GetBusinessRules. Now we have one line of code for the public RegionDescription field and three lines of code for the GetBusinessRules method. Thirty lines down to four is a big improvement.

A larger example is your EmployeeDC class. It has 18 properties, which have approximately 518 lines of code devoted to the public attributes plus another 26 lines of code devoted to the GetBusinessRules method. Using reflection, you can knock the number of lines of code down to 21—18 for the properties and three for the GetBusinessRules method. That is 544 lines of code knocked down to just 21 lines of code.

Furthermore, your objects are now truly self-describing. Any changes you make to your business rules are now instantly reflected when you retrieve the business rules from the class. The code reduction plus the self-describing class means that your maintenance costs will go through the floor. No longer do developers have to hunt through the code looking for the rule—they just have to check the attribute tag. Also, this reduces the number of code defects caused by bad business rule checks. Because your business rules are encapsulated, if they are wrong in one place, they are wrong in every place, and it will be much easier to capture these defects and correct them.

With all of the wonderful things that reflection provides, you may be asking yourself at this point why you saw the original method for handling business rules at all. After all, what is the point because this is so much easier and provides so many more advantages? The reason is that this is not a one-size-fits-all solution. On several occasions I have had to create systems that use a rules database because the business rules changed so quickly. In cases such as this, the objects generally need to open up a connection to a database to read the rule information. This is a lot of overhead and in the few tests that I have run is not well served by the reflection model. The reason for this is that the attributes cannot be dynamically changed at runtime by reading from a database. So, it is best to know both methods and apply them as necessary.

There is one last change you need to make to the RegionDC class—it is a change to the Save method. Currently, the first part of your Save method looks like the following:

```
mobjBusErr = New BusinessErrors

With sRegion
     Me.mintRegionID = .RegionID
     Me.RegionDescription = .RegionDescription
End With
```

The change you need to make is simple. Delete the first line from the previous code, and add the following line of code below the With block:

```
mobjBusErr = mobjVal.validate(Me)
```

Now, after all of your properties are assigned, you call the Validate method, retrieve the business errors, and continue as before.

Implement User-Centric Business Rule Attribute Classes

Checking business rules with custom attributes is a little different on the user-centric side. The reason for this is that you check the rules one property at a time. Not only do you still need to throw an exception when an error occurs, but you also need to add an entry to the BrokenRules object. That is a lot more work than you had to do in the data-centric class. Specifically, you cannot get rid of the public properties in the user-centric class like you did in the data-centric class. However, your job is made much easier by the presence of the BusinessBase class.

Before you modify your user-centric classes, let's add a new method to the Validation class. This method throws an exception on the first broken rule it encounters. Add the method shown in Listing 10-16 to the Validation class.

Listing 10-16. The ValidateAndThrow Method

```
Public Shared Sub ValidateAndThrow(ByVal cls As Object, ByVal field As String)
     Dim t As Type = cls.GetType
     Dim m As MemberInfo() = t.GetMember(field)
     Dim i As Integer
     Dim bln As Boolean
     Dim obj() As Object = m(0).GetCustomAttributes(True)

     If obj.Length > 0 Then
         For i = 0 To obj.Length - 1
               If TypeOf obj(i) Is ITest Then
                    Dim objI As ITest = CType(obj(i), ITest)

                    If TypeOf m(0) Is FieldInfo Then
                         Dim fld As FieldInfo = CType(m(0), FieldInfo)
                         bln = objI.TestCondition(fld.GetValue(cls), cls)
                    End If

                    If TypeOf m(0) Is PropertyInfo Then
                         Dim pro As PropertyInfo = CType(m(0), PropertyInfo)
                         bln = objI.TestCondition(pro.GetValue(cls, Nothing), cls)
                    End If
                    If bln Then
                         Throw New Exception(objI.GetRule())
                    End If
               End If
          Next
     End If
End Sub
```

This code is similar to what you have seen before, with the exception that when a broken rule is encountered, an exception is thrown. Notice that it does not specify the property that the exception is thrown on—you know what it is because you had to pass the property into the method. Notice also at the top of the method that you are only retrieving the information for the one property or field that you specified, not for the whole class. That is the extent of this method; now you can implement it in the BusinessBase class.

To begin with, modify the BusinessBase class by adding the following declaration:

```
Protected mobjVal As BusinessRules.Validate.Validation
```

Next you need to add a method that will call the ValidateAndThrow method and will process the results appropriately. Listing 10-17 shows the method, which should be added to the BusinessBase class.

Listing 10-17. The Validate Method of the BusinessBase Class

```
Protected Sub Validate(ByVal strProperty As String)
     Try
          mblnDirty = True
          mobjVal.ValidateAndThrow(Me, strProperty)
          mobjRules.BrokenRule(strProperty, False)
     Catch exc As Exception
          mobjRules.BrokenRule(strProperty, True)
          Throw exc
     End Try
End Sub
```

This method is simple—it takes a property name and calls the ValidateAndThrow method. If no exceptions are thrown, the property is set to not broken; if there is an exception, the property is marked as broken and the exception is rethrown.

Next you need to modify the Region class; specifically, you need to modify the public RegionDescription property. Before you do anything else, you need to add the following Imports line to the Region.vb code module:

```
Imports BusinessRules.Attributes
```

Then you need to tag the RegionDescription property with your business rule attributes. Change the property signature to read as follows:

```
<DisplayName("Region Description"), NotNull(), NotEmpty(), MaxLength(50)> _
Public Property RegionDescription() As String
```

Note that technically you do not need the DisplayName tag here, but it cannot hurt to have it—the choice is yours. Now that you have modified the tag, you need to alter the Set part of the method to read as follows:

```
Set(ByVal Value As String)
     If mstrRegionDescription.Trim <> Value Then
          mstrRegionDescription = Value
          If Not Loading Then
               Me.Validate("RegionDescription")
          End If
     End If
End Set
```

All of the functionality that had been in this method is now encapsulated in either the ValidateAndThrow method or the Validate method of the BusinessBase class. In either case, this property was originally 28 lines of code and it is now 13 lines of code—and that is just for one property!

Summary

This chapter showed you one of the most powerful abilities of the .NET framework: reflection. This is the ability of the Framework to examine itself and invoke things it knows nothing about. There are virtually an unlimited number of things you can do with this ability. You created a sample application to dynamically load a listview from an unknown object with unknown column headers and unknown information. Most importantly, you created a reusable business rule project that will save you countless hours in development time, lines of code, and maintenance costs. You implemented these rules on both the data-centric and user-centric classes and you made a truly self-describing class.

In the next chapter you will move on to one of the hottest topics in Information Technology today: Web services. You will learn a little bit about what they are and you will turn part of your NorthwindTraders application into a Web service. Then you will see how to consume the Web service and publish it using Microsoft's .NET Server 2003 Universal Description, Discovery, and Integration (UDDI).