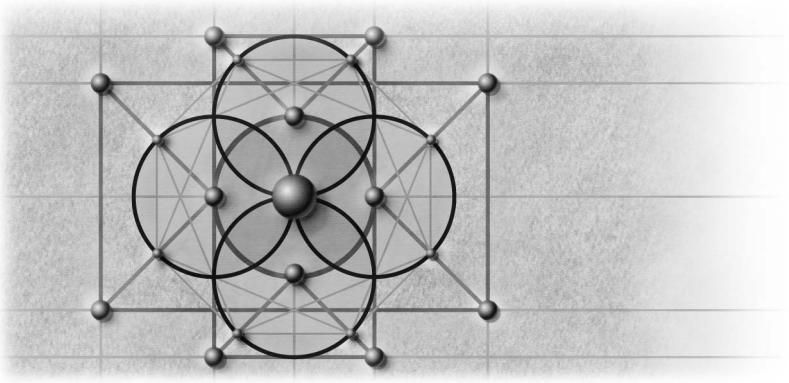

# 17

# Upgrading VB Application Wizard Projects

This chapter looks specifically at upgrading projects created with the VB Application Wizard. Because the wizard generates the same forms and modules for the applications it creates, each project will have the same problems that can be fixed in the same way. Of course, the wizard creates shell projects that are used as a basis for development. Here we'll discuss only the issues common to these shell projects generated by the wizard—not ones that may occur with code you have subsequently added.

When you create a new project in Microsoft Visual Basic 6, the New Project dialog box gives you a number of choices as to the type of project to create. The most common choice is Standard EXE, followed by ActiveX DLL. These create empty projects, to which you have to add menus, toolbars, and the corresponding logic. Since many projects have the same basic components, Visual Basic 6 provides a VB Application Wizard that generates many of these components for you. This wizard starts when you select VB Application Wizard as the project type in the New Project dialog box, as shown in Figure 17-1.

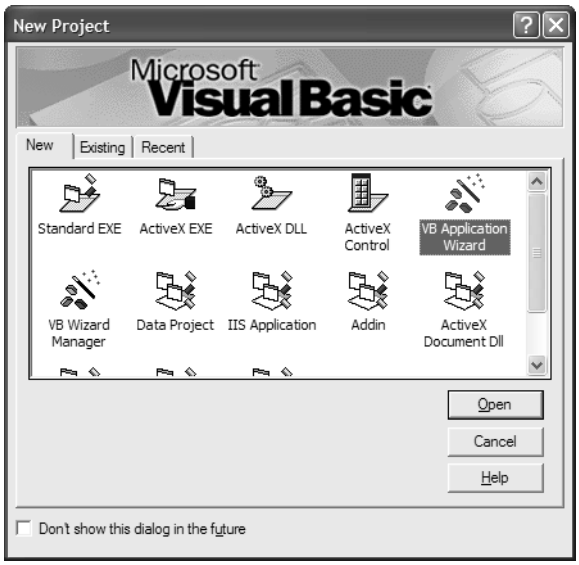


**Figure 17-1** Starting the VB Application Wizard from the New Project dialog box.

The VB Application Wizard asks you a number of questions about the project you want to create, such as the type of interface you want the project to have, the menus and submenus you need, whether strings will be stored in a resource file, and what data access forms the application will have. After you answer these questions, the wizard creates a project that is prepopulated with commonly used forms and classes. This project then becomes a template for your application. You create more forms and classes and add the business logic. Table 17-1 lists the project items the wizard creates.

**Table 17-1 Project Items Created by the VB Application Wizard**

| Project Item | Description |
|---|---|
| frmAbout form | frmAbout is the form that opens when users choose About from the Help menu. |
| frmLogin form | frmLogin is a simple username/password login form that opens when the application first starts. |
| frmMain form | Each project created with the VB Application Wizard has a form called frmMain. This is the primary form of the application. |
| frmSplash form | frmSplash is a splash screen that shows after the frmLogin form but before frmMain opens. |
| frmBrowser | An application for which you specify Internet connectivity has a frmBrowser form. This form contains a WebBrowser ActiveX control. |

**Table 17-1 Project Items Created by the VB Application Wizard**

| Project Item | Description |
|---|---|
| Data forms | Data forms are added to display and edit information from a database. |
| Module1 module | Each project created with the VB Application Wizard has a module called Module1. This module has a *Sub Main* method, the startup object that opens the frmMain form. |
| frmDocument form | Each MDI interface project has a form called frmDocument; this is the MDI child form. |
| frmOptions form | frmOptions is a tabbed dialog form for setting application options. Because this form upgrades without issues, we won't discuss it any further in this chapter. |

Your project created with the VB Application Wizard may have all or some of the project items listed in Table 17-1, depending on the options you selected in the wizard. Each of these project items has its own unique set of upgrade issues. Let's look at these items and see what you have to do to get them working in Visual Basic .NET.

## App.Revision

Although *App.Revision* isn't actually a project item, the VB Application Wizard projects use it in several places, so we'll discuss it right at the beginning. In Visual Basic 6, it is common to concatenate the three *App* object properties—*App.Major*, *App.Minor*, and *App.Revision*—together to create a version number for the application. For example, in frmAbout, you will find the following line of code in *Form_Load*:

```
lblVersion.Caption = "Version " & App.Major & "." & App.Minor & _
   "." & App.Revision
```

Visual Basic .NET does not have an *App* object, so the Upgrade Wizard chooses the most appropriate upgrade for each property. *App.Major* is upgraded to

```
System.Diagnostics.FileVersionInfo.GetVersionInfo( _
   System.Reflection.Assembly.GetExecutingAssembly.Location _
   ).FileMajorPart
```

*App.Minor* is upgraded to

```
System.Diagnostics.FileVersionInfo.GetVersionInfo( _
   System.Reflection.Assembly.GetExecutingAssembly.Location _
   ).FileMinorPart
```

However, there is no equivalent for *App.Revision*, so the Upgrade Wizard leaves it as is, which creates a compile error in Visual Basic .NET. For example, the Visual Basic 6 line

```
lblVersion.Caption = "Version " & App.Major & "." & App.Minor & _
   "." & App.Revision
```

upgrades to

```
lblVersion.Text = "Version " & _
   System.Diagnostics.FileVersionInfo.GetVersionInfo( _
   System.Reflection.Assembly.GetExecutingAssembly.Location _
   ).FileMajorPart _
   & "." & System.Diagnostics.FileVersionInfo.GetVersionInfo( _
   System.Reflection.Assembly.GetExecutingAssembly.Location _
   ).FileMinorPart & "." & App.Revision
```

There are two ways to fix the compile error. The first and easiest is simply to remove the *App.Revision* part of the line:

```
& "." & App.Revision
```

After this change, only the major and minor parts of the version will be shown. Here is what the code just shown looks like after the *App.Revision* part is removed:

```
lblVersion.Text = "Version " & System.Diagnostics.FileVersionInfo.Get-
VersionInfo( _
   System.Reflection.Assembly.GetExecutingAssembly.Location _
   ).FileMajorPart _
   & "." & System.Diagnostics.FileVersionInfo.GetVersionInfo( _
   System.Reflection.Assembly.GetExecutingAssembly.Location _
   ).FileMinorPart
```

While this is a good option, Visual Basic .NET also has a property that returns the entire version number of the application in 0.0.0.0 format. This property is the best way to create a version number in Visual Basic .NET, since it gives the full version information. Here is the replacement code:

```
lblVersion.Text = "Version " & Application.ProductVersion
```

You'll also notice that this line is shorter than the Visual Basic 6 version. You can use this line as a replacement for any code that concatenates together *App.Major*, *App.Minor*, and *App.Revision* to generate a version number.

### What's Up with Versioning in Visual Basic .NET?

Visual Basic 6 has three versioning properties: *App.Major*, *App.Minor*, and *App.Revision*. You can choose to make *App.Revision* auto-increment so that the revision number increases each time you compile the project. Although this is a simple model, it is out of step with Windows versioning, which supports four versioning properties: *Major*, *Minor*, *Revision*, and *Build*. When a Visual Basic 6 project is compiled, *App.Revision* is mapped to the Windows *Build* property and the Windows *Revision* property is left blank. Visual Basic .NET supports the four Windows versioning properties. The version number is set using the *AssemblyVersion* attribute in the AssemblyInfo.vb file. By default, the revision and build are set to be auto-incrementing based on the time of compilation—when combined they are a timestamp giving the date and time of compilation.

## frmAbout Form

The VB Application Wizard adds the frmAbout form to your project when you choose to add an About box standard form to the application. The About box opens when the user chooses About from the Help menu. It shows the version information for the application and has a command button that opens the System Information application. There is one compile error that you need to fix after upgrading. It is in the *Form_Load* event, and it occurs because the wizard uses the *App.Revision* property to generate the version information shown in the box. As we discussed in the previous section, you should replace the code

```
lblVersion.Text = "Version " & _
   System.Diagnostics.FileVersionInfo.GetVersionInfo( _
   System.Reflection.Assembly.GetExecutingAssembly.Location _
   ).FileMajorPart _
   & "." & System.Diagnostics.FileVersionInfo.GetVersionInfo( _
   System.Reflection.Assembly.GetExecutingAssembly.Location _
   ).FileMinorPart & "." & App.Revision
```

with the following:

```
lblVersion.Text = "Version " & Application.ProductVersion
```

In the *StartSysInfo* procedure, you'll also find a warning that *Dir* has a new behavior. This behavior difference applies to the *Dir* method when it is used to return a list of directories. In this case, since *Dir* is being used to determine whether the file Msinfo32.exe exists, the warning can be ignored, since it applies only to directories.

# frmLogin Form

The frmLogin form is added to your project when you choose the option to add a login dialog box to the application. This form works perfectly after upgrading.

# frmMain Form

The application's primary form is frmMain. For multiple document interface (MDI) applications, it is an MDIForm; for single document interface (SDI) and Explorer applications, it is a standard form. Most of the modifications you have to complete for upgraded VB Application Wizard projects involve this form.

## API *Declare* Statements

All projects generated by the VB Application Wizard contain an API *Declare* statement for OSWinHelp. MDI applications also contain a *Declare* statement for the SendMessage API. Here are the two *Declare* statements in Visual Basic 6:

```
Private Declare Function SendMessage Lib "user32" _
   Alias "SendMessageA" _
   (ByVal hwnd As Long, ByVal wMsg As Long, ByVal wParam As Long, _
   ByVal lParam As Any) As Long

Private Declare Function OSWinHelp% Lib "user32" Alias "WinHelpA" _
   (ByVal hwnd&, ByVal HelpFile$, ByVal wCommand%, dwData As Any)
```

After upgrading, both of these *Declare* statements cause compile errors, since each of them passes variables as *As Any*. Here are the upgraded *Declare* statements:

```
Private Declare Function SendMessage Lib "user32"  Alias _
   "SendMessageA"(ByVal hwnd As Integer, ByVal wMsg As Integer, _
   ByVal wParam As Integer, ByVal lParam As Any) As Integer

Private Declare Function OSWinHelp Lib "user32"  Alias "WinHelpA"( _
   ByVal hwnd As Integer, ByVal HelpFile As String, _
   ByVal wCommand As Short, ByRef dwData As Any) As Short
```

The SendMessage API is not used in VB Application Wizard projects, so you can simply remove the declaration for SendMessage. The *As Any* parameter for OSWinHelp should be changed to *String*, since it is used to pass the name of the Help file. After removing the SendMessage API and changing the OSWin-Help API, the declare statement for the API looks like this:

```
Private Declare Function OSWinHelp Lib "user32" Alias "WinHelpA" ( _
   ByVal hwnd As Integer, ByVal HelpFile As String, _
   ByVal wCommand As Short, ByRef dwData As String) As Short
```

**Note** Although not discussed here, the correct Visual Basic .NET declaration for SendMessage is:

```
Private Declare Function SendMessage Lib "user32" Alias _
   "SendMessageA" (ByVal hwnd As Integer, _
    ByVal wMsg As Integer, _
    ByVal wParam As Integer, ByVal lParam As Integer) _
    As Integer
```

## *mnuHelpAbout_Click* Event Procedure

If you haven't added an About box to the application, the event for the About menu item, *mnuHelpAbout_Click*, has code that opens a message box with the application version:

```
MsgBox "Version " & App.Major & "." & App.Minor & "." & App.Revision
```

After upgrading, this code causes an error because it uses *App.Revision* to generate the application version. It upgrades to

```
MsgBox("Version " & _
   System.Diagnostics.FileVersionInfo.GetVersionInfo( _
   System.Reflection.Assembly.GetExecutingAssembly.Location _
   ).FileMajorPart _
   & "." & System.Diagnostics.FileVersionInfo.GetVersionInfo( _
   System.Reflection.Assembly.GetExecutingAssembly.Location _
   ).FileMinorPart & "." & App.Revision)
```

As we discussed in the section on *App.Revision*, you should replace this line with the following:

```
MsgBox("Version " & Application.ProductVersion)
```

## *App.HelpFile*

*App.HelpFile* is used in several places in the frmMain form. Windows Forms has a new system for displaying Help that is not compatible with Visual Basic 6. Because of this incompatibility, the *App.HelpFile* property is not upgraded and causes a compile error after upgrading. If your application uses context-sensitive Help, you will have to reimplement it in Visual Basic .NET. If your application simply shows Help contents and a Help index, you can modify your application to show them. Simply replace the instances of *App.HelpFile* with the name of the Help file.

For example, in the *mnuHelpSearchForHelpOn_Click* event handler, the line

```
nRet = OSWinHelp(Me.hwnd, App.HelpFile, 261, 0)
```

is upgraded to

```
nRet = OSWinHelp(Me.Handle.ToInt32, App.HelpFile, 261, 0)
```

*App.HelpFile* generates a compile error. Suppose that your application's Help file is called C:\MyProject\MyProject.hlp. You would modify the line to read as follows:

```
nRet = OSWinHelp(Me.Handle.ToInt32, "c:\MyProject.hlp", 261, 0)
```

If your application doesn't contain a Help file, you can replace all instances of *App.HelpFile* with an empty string. For example, you would modify the line just shown to the following:

```
nRet = OSWinHelp(Me.Handle.ToInt32, "", 261, 0)
```

An easy way to make this change is to use the Find And Replace dialog box to replace all instances of "App.HelpFile" with "", the empty string.

> **More Info** Implementing context-sensitive help in Visual Basic .NET applications is outside the scope of this book. If you want to learn more, search for the following topics in the Visual Basic .NET help: "Help and User Assistance in Windows Applications" and "Introduction to the Windows Forms HelpProvider Control."

## *ActiveMdiChild* in MDI Projects

In MDI applications, code in event procedures in the frmMain form, such as *tbToolBar_ButtonClick*, often uses soft binding to access controls on the active MDI child form. Visual Basic .NET detects and generates compile errors for controls that are accessed in this way.

Before discussing the solution, let's look a little closer at the reason that soft binding is used. When a user clicks a toolbar button or chooses a menu command from the MDI parent form, the code often acts upon the currently selected MDI child form. If an MDI application has many child forms, the active

form is determined only at run time; therefore, it is common to use the *ActiveForm* object to access controls on the current form. For example, the following extract from the *tbToolBar_ButtonClick* event procedure shows how to change the text alignment of the selected text inside a control called rtfText on the active form:

```
Select Case Button.Key
   Case "Align Right"
      ActiveForm.rtfText.SelAlignment = rtfRight
```

For MDI applications, the Visual Basic .NET equivalent of Visual Basic 6's *ActiveForm* is *ActiveMdiForm*. The code just shown upgrades to the following:

```
Select Case eventArgs.button.Key
   Case "Align Right"
      'UPGRADE_WARNING: Control rtfText could not be resolved because
      'it was within the generic namespace ActiveMdiChild.
      ActiveMdiChild.rtfText.SelAlignment = _
         RichTextLib.SelAlignmentConstants.rtfRight
```

Notice that the Upgrade Wizard has inserted an EWI because it cannot resolve rtfText. Because the active form is defined dynamically at run time, the wizard has no way of knowing what control rtfText actually refers to. Visual Basic .NET has stronger type checking than Visual Basic 6, so it generates a compile error for this line of code. All the compiler knows is that *ActiveMdiChild* is a *Form* object, and rtfText is not a valid control or property of a standard *Form* object. To fix this error, you should either strongly type the form or force the code to use late binding. When possible, you should strongly type the form because doing so gives you IntelliSense and type checking at compile time. Here is how to modify the code to use strong type checking.

First of all, add a property called *GetActiveMdiChild* to the form:

```
ReadOnly Property GetActiveMdiChild() As frmDocument
   Get
      Return Me.ActiveMdiChild
   End Get
End Property
```

Then change all occurrences of *ActiveMdiChild* to use this property:

```
Select Case eventArgs.button.Key
   Case "Align Right"
      GetActiveMdiChild.rtfText.SelAlignment = _
         RichTextLib.SelAlignmentConstants.rtfRight
```

Notice that since our property is of type *frmDocument* (the MDI child form), code can access properties, methods, and controls of the MDI child form. The compiler knows that rtfText is a control of *frmDocument*, so this code compiles and runs.

You will have to make this change for every procedure that uses controls of *ActiveMdiChild*. The easiest way to do this is to use the Find And Replace dialog box to replace all instances of "ActiveMdiChild" with "GetActiveMdi-Child". When you're doing the find and replace, be careful not to rename the newly added *GetActiveMdiChild* property.

What about MDI applications that have several different form types? What if the active child form could be one of a number of different forms? In most cases, the different child forms won't all have controls with the same name, but you can still fix the code to work. Defining *GetActiveMdiChild* as type *Object* will force the code to use late binding, and the control will be resolved only at run time. To do this, change the property to the following:

```
ReadOnly Property GetActiveMdiChild() As Object
   Get
      Return Me.ActiveMdiChild
   End Get
End Property
```

Where possible, however, you should avoid late binding and instead use the strongly typed version of the property. When the property is strongly typed, you get IntelliSense and type checking, and the code executes quicker.

## Forms Collection in *frmMain_Closed*

In the *Form_Unload* event of SDI and Explorer applications, the VB Application Wizard generates code that unloads each open form:

```
'Close all sub forms
For i = Forms.Count - 1 To 1 Step -1
   Unload Forms(i)
Next
```

This code upgrades to the following. Not shown here is that *Form_Unload* is renamed *frmMain_Closed* during the upgrade.

```
'Close all sub forms
'UPGRADE_ISSUE: Forms collection was not upgraded.
'UPGRADE_WARNING: Couldn't resolve default property of object
Forms.Count.
For i = Forms.Count - 1 To 1 Step -1
   'UPGRADE_ISSUE: Forms collection was not upgraded.
   'UPGRADE_ISSUE: Unload Forms(i) was not upgraded.
   Unload(Forms(i))
Next
```

As we discussed in Chapter 15, the forms collection cannot be upgraded automatically. Fixing this problem is simple; you can remove this entire section of code, since Visual Basic .NET applications automatically unload all their child forms when the main form closes.

## Clipboard in MDI Projects

MDI applications have three event procedures for the *Clipboard* object: Cut, Copy, and Paste. They are created as follows:

```
Private Sub mnuEditCut_Click()
    On Error Resume Next
    Clipboard.SetText ActiveForm.rtfText.SelRTF
    ActiveForm.rtfText.SelText = vbNullString
End Sub

Private Sub mnuEditCopy_Click()
    On Error Resume Next
    Clipboard.SetText ActiveForm.rtfText.SelRTF
End Sub

Private Sub mnuEditPaste_Click()
    On Error Resume Next
    ActiveForm.rtfText.SelRTF = Clipboard.GetText
End Sub
```

To make looking at the upgraded Clipboard code simpler, let's focus on the two lines of Clipboard code that set and get the Clipboard text, respectively:

```
Clipboard.SetText ActiveForm.rtfText.SelRTF

ActiveForm.rtfText.SelRTF = Clipboard.GetText
```

After the upgrade, these two lines cause errors in Visual Basic .NET, since the *Clipboard* object cannot be upgraded automatically (see Chapter 15).

```
'UPGRADE_ISSUE: Clipboard method Clipboard.SetText was not upgraded.
Clipboard.SetText(ActiveMdiChild.rtfText.SelRTF)
'UPGRADE_ISSUE: Clipboard method Clipboard.GetText was not upgraded.
ActiveMdiChild.rtfText.SelRTF = Clipboard.GetText
```

You can fix the problem by adding two helper methods to Module1. These helper methods get and set the Clipboard text:

```
Function ClipboardSetText(ByVal newText As String)
   Try
      Clipboard.SetDataObject(newText, True)
   Catch ex As Exception
      MsgBox("Exception setting clipboard text: " & ex.Message)
   End Try
End Function
```

*(continued)*

```
Function ClipboardGetText()
   Try
      Dim sClipText As String
      Dim stringData As IDataObject
      Dim getString As New DataObject(DataFormats.StringFormat)
      stringData = Clipboard.GetDataObject()
      If stringData.GetDataPresent(DataFormats.Text) Then
         sClipText = stringData.GetData(DataFormats.Text, True)
         Return sClipText
      End If
   Catch ex As Exception
      MsgBox("Exception getting clipboard text: " & ex.Message)
   End Try
End Function
```

Now you just need to modify the set and get code to call these functions:

```
ClipboardSetText(GetActiveMdiChild.rtfText.SelRTF)

GetActiveMdiChild.rtfText.SelRTF = ClipboardGetText()
```

Notice that we've also changed *ActiveMdiChild* to *GetActiveMdiChild*, as we recommended earlier.

## frmSplash Form

The form frmSplash is a splash screen. It is added to the project if you specify a splash screen at application startup in the VB Application Wizard. After being upgraded, this form has two problems. First, as the frmAbout form does, frmSplash inserts the application version into a label. As we discussed in the earlier section on *App.Revision*, you should replace the version code in *Form_Load*. The following Visual Basic 6 code

```
lblVersion.Caption = "Version " & App.Major & "." & App.Minor & "." _
   & App.Revision
```

upgrades to

```
lblVersion.Text = "Version " & _
   System.Diagnostics.FileVersionInfo.GetVersionInfo( _
   System.Reflection.Assembly.GetExecutingAssembly.Location _
   ).FileMajorPart _
   & "." & System.Diagnostics.FileVersionInfo.GetVersionInfo( _
   System.Reflection.Assembly.GetExecutingAssembly.Location _
   ).FileMinorPart & "." & App.Revision
```

Replace this code with the following:

```
lblVersion.Text = "Version " & Application.ProductVersion
```

The second problem is something new. Applications with a splash screen perform a series of steps in *Sub Main* to

1. Show the frmSplash form.
2. Refresh the frmSplash form (forcing the form to be fully painted).
3. Load the frmMain form.
4. Unload the frmSplash form.
5. And finally, show the frmMain form.

Here is the code that performs this series of steps:

```
Sub Main()
   frmSplash.Show
   frmSplash.Refresh
   Set fMainForm = New frmMain
   Load fMainForm
   Unload frmSplash
   fMainForm.Show
End Sub
```

After upgrading, this code looks like the following:

```
Public Sub Main()
   frmSplash.DefInstance.Show()
   frmSplash.DefInstance.Refresh()
   fMainForm.DefInstance = New frmMain
   'UPGRADE_ISSUE: Load statement is not supported.
   Load(fMainForm)
   frmSplash.DefInstance.Close()
   System.Windows.Forms.Application.Run(fMainForm.DefInstance)
End Sub
```

This code causes a compile error because the *Load* statement is not supported in Visual Basic .NET. To fix the error, remove the *Load(fMainForm)* statement; the application will work normally. The updated *Sub Main* event will look like this:

```
Public Sub Main()
   frmSplash.DefInstance.Show()
   frmSplash.DefInstance.Refresh()
   fMainForm.DefInstance = New frmMain
   frmSplash.DefInstance.Close()
   System.Windows.Forms.Application.Run(fMainForm.DefInstance)
End Sub
```

## frmBrowser Form

When creating a project with the VB Application Wizard, if you indicate that you want your users to be able to access the Internet from your application, the wizard adds the frmBrowser form to the application. Choosing Web Browser from the View menu then opens the form. The code generated in the *mnuViewWebBrowser_Click* event in frmMain looks like this:

```
Private Sub mnuViewWebBrowser_Click()
   Dim frmB As New frmBrowser
   frmB.StartingAddress = "http://www.microsoft.com"
   frmB.Show
End Sub
```

This code upgrades to the following:

```
Public Sub mnuViewWebBrowser_Click(ByVal eventSender As _
   System.Object, _
   ByVal eventArgs As System.EventArgs) _
   Handles mnuViewWeb Browser.Click
   Dim frmB As frmBrowser = New frmBrowser
   frmB.StartingAddress = "http://www.microsoft.com"
   frmB.Show()
End Sub
```

If this is an MDI project, you need to adjust the behavior of the frmBrowser form. Open the code window for frmBrowser and search for the line *Me.Show*. This statement is in the *Sub New* method within the "Windows Form Designer generated code" hidden region. Remove the *Me.Show* line.

The reason for removing the line is to avoid an event ordering issue. This line causes the *Form_Load* event to fire before the *mnuViewWebBrowser* method has assigned the *StartingAddress* property of the form. After you remove the *Me.Show* line, the frmBrowser form will navigate to the default Web site when it is first opened.

## Data Forms

The VB Application Wizard has an option to add data access forms to the project. Each data access form allows users to add, update, and delete data in a database table. The forms use ADO data binding and can be set up to use binding with the ADO Data control, code, or a data class.

The Upgrade Wizard can upgrade only projects with ADO Data control data binding. Forms that accomplish data binding via code will not work in Visual Basic .NET because the controls do not support the *DataSource* property. Data binding by using a data class cannot be upgraded for two reasons:

the controls do not support the *DataSource* property, and the *DataSource* behavior of classes cannot be upgraded automatically. If your application uses either of these two types of data binding, you will need to reimplement the data binding, using ADO or ADO.NET.

Forms with ADO Data control data binding are upgraded to Visual Basic .NET. Data binding has its own set of issues, and these are covered in Chapter 14.

# Module1 Module

The VB Application Wizard adds a module called Module1 to your project. This module contains the *Sub Main* startup method and will also contain the *LoadResStrings* method if you chose to store the project's strings in a resource file.

## *LoadResStrings* Method

If you choose Yes when the VB Application Wizard asks if you would like to use a resource file for the strings in your application, the wizard stores strings (such as text for menu items) in a resource file. The wizard adds a method named *LoadResStrings* to the application to load the strings at run time. This procedure loads the strings for menu captions, ToolTips, control captions, and fonts. After the upgrade, this procedure causes 19 compile errors, mainly because it uses soft binding. The Upgrade Wizard adds an EWI to the method, advising you to replace the function with the following code supplied in Help:

```
Sub LoadResStrings(ByRef frm As System.Windows.Forms.Form)
   On Error Resume Next
   Dim ctl As System.Windows.Forms.Control
   Dim obj As Object
   Dim fnt As System.Drawing.Font
   Dim sCtlType As String
   Dim nVal As Short

   ' Set the form's caption.
   frm.Text = VB6.LoadResString(CShort(frm.Tag))

   ' Set the font.
   Dim FontName As String, FontSize As Double
   FontName = VB6.LoadResString(20)
   FontSize = CShort(VB6.LoadResString(21))
   frm.Font = New System.Drawing.Font(FontName, FontSize)

   ' Set the controls' captions using the Caption
   ' property for menu items and the Tag property
   ' for all other controls.
   For Each ctl In frm.Controls
```

*(continued)*

```
            ctl.Font = fnt
            sCtlType = TypeName(ctl)
            If sCtlType = "Label" Then
               ctl.Text = VB6.LoadResString(CShort(ctl.Tag))
            ElseIf sCtlType = "AxTabStrip" Then
               For Each obj In CObj(ctl).Tabs
                  obj.Caption = VB6.LoadResString(CShort(obj.Tag))
                  obj.ToolTipText = VB6.LoadResString _
                     (CShort(obj.ToolTipText))
               Next obj
            ElseIf sCtlType = "AxToolbar" Then
               For Each obj In CObj(ctl).Buttons
                  obj.ToolTipText = VB6.LoadResString _
                     (CShort(obj.ToolTipText))
               Next obj
            ElseIf sCtlType = "AxListView" Then
               For Each obj In CObj(ctl).ColumnHeaders
                  obj.Text = VB6.LoadResString(CShort(obj.Tag))
               Next obj
            Else
               nVal = 0
               nVal = Val(ctl.Tag)
               If nVal > 0 Then ctl.Text = VB6.LoadResString(nVal)
               nVal = 0
               nVal = Val(CObj(frm).ToolTip1.GetToolTip(ctl))
               If nVal > 0 Then
                  CObj(frm).ToolTip1.SetToolTip(ctl, VB6.Load-
                     ResString(nVal))
               End If
            End If
         Next ctl

         Dim mnu As System.Windows.Forms.MainMenu = CObj(frm).MainMenu1
         If Not mnu Is Nothing Then
            LoadMenuResStrings(mnu)
         End If
      End Sub

      Public Sub LoadMenuResStrings(ByVal mnu As System.Windows.Forms.Menu)
         Dim mnuItem As System.Windows.Forms.MenuItem
         For Each mnuItem In mnu.MenuItems
            On Error Resume Next
            mnuItem.Text = VB6.LoadResString(CInt(mnuItem.Text))
            On Error Goto 0
            If mnuItem.MenuItems.Count > 0 Then
               LoadMenuResStrings(mnuItem)
            End If
         Next
      End Sub
```

This code is a replacement for the standard *LoadResStrings* method generated by the VB Application Wizard. If you haven't modified the method, simply copy and paste this procedure into your application, replacing the existing *LoadResStrings* method.

## Conclusion

This chapter has focused on upgrading projects created with the VB Application Wizard. It's important to note that the applications generated by this wizard are not complete. They are a starting point from which you continue to add your own business logic and project items. We've examined how to fix problems with the initial project items and code as they are generated by the wizard. Obviously, there will be other issues that occur as a result of code you add to these base projects. You'll find solutions to these general upgrade issues in other chapters of this book.