ActiveX controls and wrapper classes

VB Migration Partner supports all the controls included in the Visual Basic 6 package, with the only exception of the OLE and Repeater controls. When migrating a form that contains unrecognized 3rd-party (non Microsoft) ActiveX controls, such controls are transformed into "placeholder" controls that appear on the form as red rectangles. For each unrecognized ActiveX control a warning is also generated in the migrated project.

VB Migration Partner can recognize and correctly convert a 3rd-party ActiveX control only if a wrapper class exists for it. To create such a wrapper class you use the **AxWrapperGen** utility.

NOTE*: you don't need to run AxWrapperGen if the ActiveX control is written in VB6 and you have its source code. In such case you can just convert the ActiveX Control project with VB Migration Partner as usual and then deploy the resulting DLL as explained in the "Deploying the wrapper class" section below. If the ActiveX control is used only by a few client projects you might also group them in a VB6 project group (*.vbg) and convert all of them in a single operation.*

Contents

Running AxWrapperGen

AxWrapperGen is a tool that is part of the VB Migration Partner package. (You can find it in the main installation directory, which by default is C:\Program Files\Code Architects\VB Migration Partner.) It is a command-line utility, therefore you must open a command window and run AxWrapperGen from there. You can read the complete AxWrapperGen reference in chapter 4 of our online manual.

We also recommend that you read the A smart approach to 3rd-party ActiveX control conversion whitepaper to become familiar with the concept of wrapper classes, ActiveX control migration and ActiveX control replacement.

***IMPORTANT NOTE**: You can run AxWrapperGen only on ActiveX controls for which you own the design-time license.*

AxWrapperGen takes one argument, that is, the path of the OCX file that contains the ActiveX control to wrap. For example, the following command creates the wrapper class for the Microsoft Calendar control (and assumes that such control is installed in the c:\windows\system32 folder):

```
AxWrapperGen c:\windows\system32\mscal.ocx
```

(Notice that in this example we use the MSCAL.OCX control only for illustration purposes, as VB Migration Partner already supports this control.) If the file name contains spaces, you must enclose the name inside double quotes. AxWrapperGen is able to convert multiple ActiveX controls in one shot

```
AxWrapperGen c:\windows\system32\mscal.ocx
   "c:\windows\system32\threed32.ocx"
```

By default, AxWrapperGen generates a new project and solution named **AxWrapper** in current directory. You can change these default settings by means of the **/out** switch (to indicate an alternative output directory) and **/project** switch (to set the name of the new project and solution):

```
AxWrapperGen c:\windows\system32\mscal.ocx /out:c:\myapp /project:NetCalendar
```

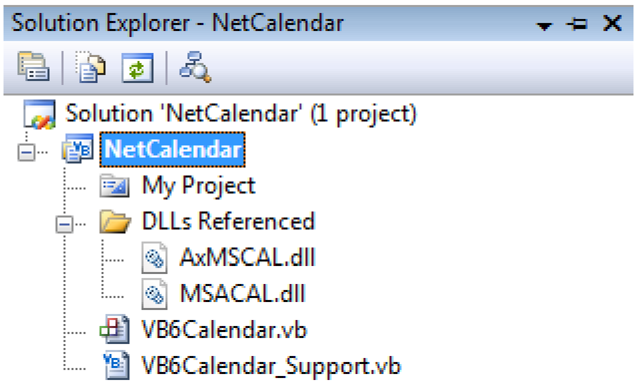
By default, AxWrapperGen generates VS2008 projects. You can generate VS2005 or VS2010 projects by adding a **/version** option:

```
AxWrapperGen c:\windows\system32\mscal.ocx /version:2010
```

AxWrapperGen runs the AxImp tool – which is part of the .NET Framework SDK – behind the scenes, to generate two DLLs that work as the RCW (Runtime Component Wrapper) for the selected control. For example, the mscal.ocx control generates the following two files: msacal.dll and axmsacal.dll. You can specify the following five options, which AxWrapperGen passes to AxImp: **/keyfile, /keycontainer, /publickey, /delaysign**, and **/source**. For more information, read .NET Framework SDK documentation.

At the end of generation process, AxWrapperGen runs Microsoft Visual Studio and loads a solution that contains one VB.NET class library (DLL) project, containing one pair of classes for each ActiveX control:

- The first class of each pair works as a wrapper for the original ActiveX control.
- The second class of each pair provides support during the migration process.



In our example, the MSCAL.OCX file contains only one ActiveX control, therefore only two classes will be created: the class named **VB6Calendar** inherits from AxMSACAL.AxCalendar and wraps the ActiveX control; the second class is named **VB6Calendar_Support** and inherits from VB6ControlSupportBase. This *_Support class is instantiated and used by VB Migration Partner during the migration process, but is never used at runtime during execution.

NOTE: *if the ActiveX control in question uses databinding to display database data using DAO and the "classic" Data control, converting it with AxWrapperGen is pretty useless, because the resulting .NET control won't find any Data control on the migrated form. Unfortunately, DAO databinding relies on a number of undocumented, proprietary Microsoft interfaces that cannot be implemented under .NET. This is the reason why VB Migration Partner doesn't (and can't) support the* ***DBGrid32****,* ***DBCombo*** *and* ***DBList*** *ActiveX controls, nor any other control whose main purpose is being bound to a DAO Data control. The same reasoning apply to controls that bind to the RDC Data control.*

Deploying the wrapper class

In most cases AxWrapperGen generates a wrapper class – or a set of wrapper classes - that compiles with no errors to a .NET DLL (NetCalendar.dll in our example). This DLL depends on *at least* the two DLLs that were generated by AxImp (AxMSCAL.dll and MSACAL.dll in our example), plus other files that AxImp may generate. ***You should consider all these DLLs as a single deployment unit and to always move all of them together*** when advised to do so.

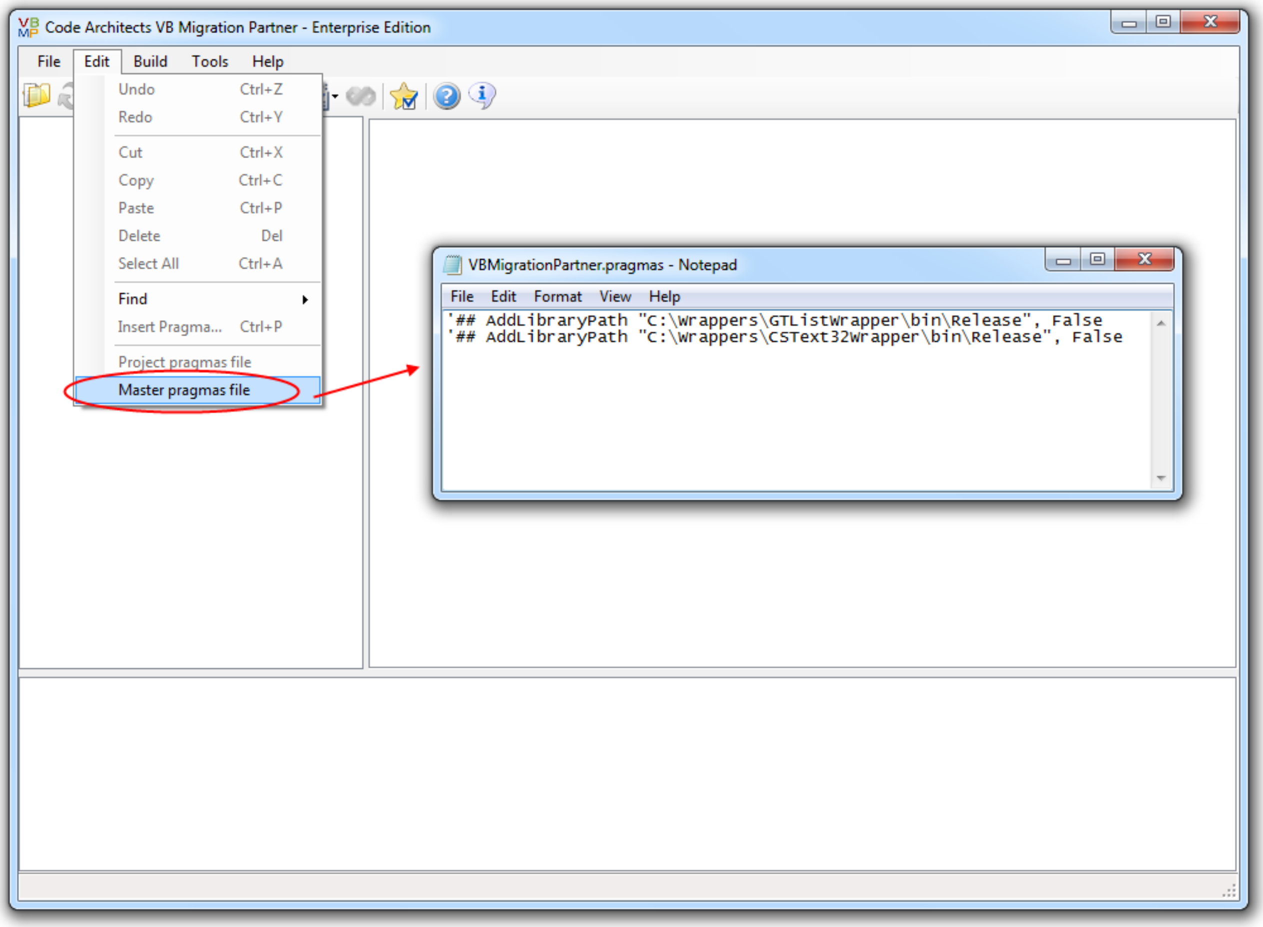
If the AxWrapperGen-generated project compiles correctly, the next step is to deploy the set of DLLs to a folder where VB Migration Partner can find it.

The first candidate for the deployment is VB Migration Partner's own setup folder. This approach is quick and simple, but has two serious shortcomings:

a. If you later fix and recompile the wrapper class project you have to manually copy again the wrapper class to VB Migration Partner's folder. It's easy to mistakenly omit this important step, in which case VB Migration Partner continues to use the previous (presumably bugged) DLL version without showing any error message.

b. As noted above, you must remember to also copy all dependency DLLs. At times this set of DLLs includes one or more files that are already present in VB Migration Partner's folder, either because they were installed with VB Migration Partner itself – for example, the ADODB.dll – or because they were deployed there when migrating another ActiveX control. If you overwrite these DLLs many strange and unexpected behaviors may result.

A better approach for deploying wrapper classes is storing them in a separate folder tree – say, c:\wrappers\<controlname> – and then use an **AddLibraryPath** pragma that points to that folder. This pragma can be only stored inside a file, which you can edit from inside VB Migration Partner's user interface:



```
'## AddLibraryPath "C:\Wrappers\GTListWrapper\bin\Release", False
'## AddLibraryPath "C:\Wrappers\CSText32Wrapper\bin\Release", False
```

Remember that the pragma file must be stored in the same folder as the VB6 project you are converting to .NET.

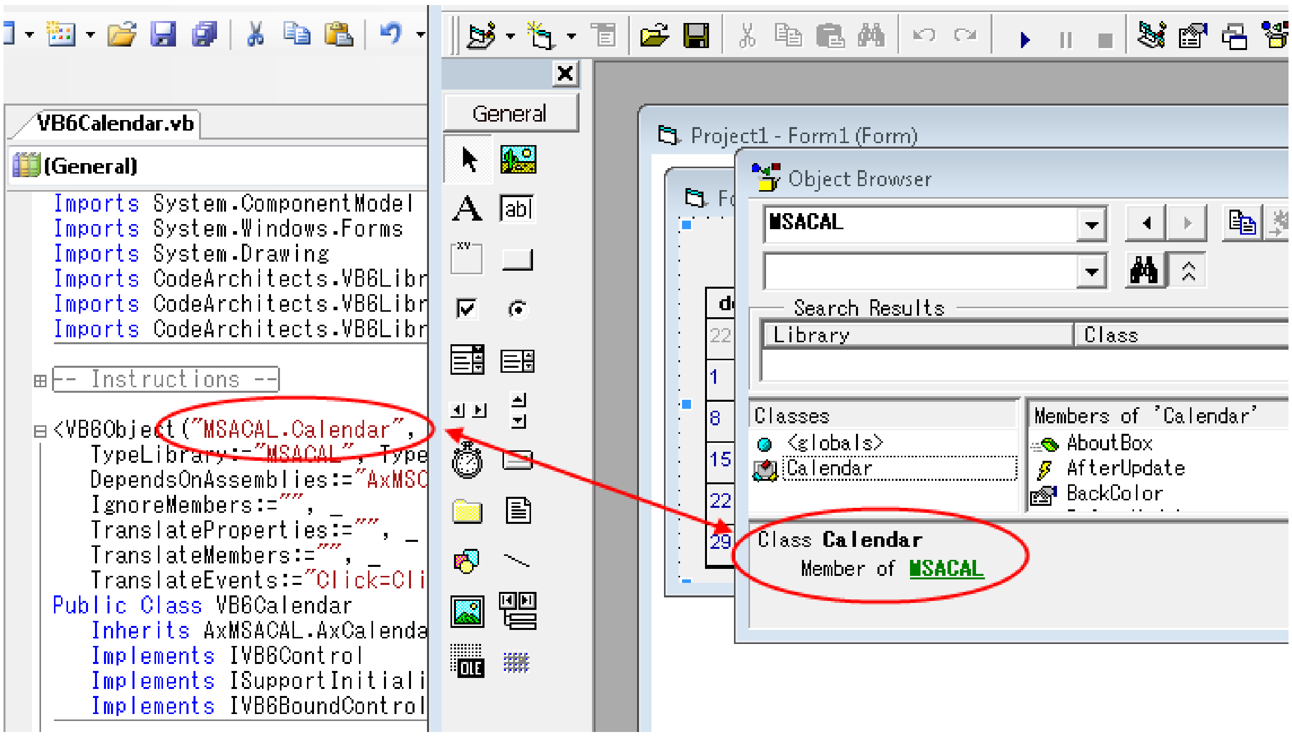
Fixing and polishing the wrapper class

In some cases, the wrapper class doesn't compile correctly and you have to manually fix these problems before continuing. Even if the wrapper class compiles correctly, however, it is a good idea to revising and polishing it before proceeding.

NOTE: *all the common actions that need to be performed on a wrapper class are summarized in the "Instructions" remark regions at the top of the wrapper class. This section expands on the recommendations described in that section.*

1. ActiveX control name

The first and most important step is checking that AxWrapperGen generated the correct ProgID for the ActiveX control, thus you need to compare the first argument of the VB6Object attribute (at the top of the wrapper class) with the control name as it appears in VB6 Object Browser:



```
Imports System.ComponentModel
Imports System.Windows.Forms
Imports System.Drawing
Imports CodeArchitects.VB6Libr
Imports CodeArchitects.VB6Libr
Imports CodeArchitects.VB6Libr

-- Instructions --

<VB6Object("MSACAL.Calendar",
   TypeLibrary:="MSACAL", Type
   DependsOnAssemblies:="AxMSC
   IgnoreMembers:="", _
   TranslateProperties:="", _
   TranslateMembers:="", _
   TranslateEvents:="Click=Cli
Public Class VB6Calendar
   Inherits AxMSACAL.AxCalenda
   Implements IVB6Control
   Implements ISupportInitiali
   Implements IVB6BoundControl
```

If these strings don't match exactly, VB Migration Partner fails to replace the original ActiveX control with this wrapper class during the migration step. After receiving and solving tons of support queries from our customers, we concluded that ***name mismatch is by far the most common cause of malfunctioning***.

2. ActiveX control “alternate” name

Some VB6 ActiveX controls have an “alternate” name, which comes into play when the control is dynamically added to a VB6 form by means of the **Controls.Add** method. For example, the standard name for the VB6 TreeView is “MSComCtlLib.TreeView”, yet you must use its alternate name “MSComCtlLib.TreeCtrl.2” when adding it dynamically:

```
Me.Controls.Add "MSComCtlLib.TreeCtrl.2", "MyTreeView"
```

To support this feature also for the ActiveX control wrapped by the class you must assign the AlternateName property of the **VB6Object** attribute. This property doesn’t appear in the AxWrapperGen-generated source code and must be added manually.

For example, if a fictitious ActiveX control named “XLib.XGrid” had an alternate name of “XLib.XGridControl.2” then you should edit its VB6Object attribute as follows:

```
<VB6Object("XLib.XGrid", _
   AlternateName:="XLib.XGridControl.2", _
   TypeLibrary:="XLib", TypeLibraryNamespace:="XLibCtrl",  _
   DependsOnAssemblies:="XLib.dll,AxXLib.dll", _
   IgnoreMembers:="")> _
Public Class VB6XGrid
```

3. The IgnoreMember property

Not all the properties of the original ActiveX control can or should be migrated to .NET. For example, properties related to docking (e.g. Align), transparent background, and color palettes don’t work under .NET; they can (and should) be ignored when an instance of the control is converted from a VB6 form to a .NET form.

During the migration of a VB6 application, VB Migration Partner looks for the IgnoreMember property of the VB6Object attribute and ignores all the properties listed there. This property is expected to contain the pipe-delimited list of members to be ignored. For example, this is how this property might be set for the fictitious XGrid control:

```
<VB6Object("XLib.XGrid", _
   AlternateName:="XLib.XGridControl.2", _
   TypeLibrary:="XLib", TypeLibraryNamespace:="XLibCtrl",  _
   DependsOnAssemblies:="XLib.dll,AxXLib.dll", _
   IgnoreMembers:="Negotiate|MaskColor|Palette|UsePalette")>  _
Public Class VB6XGrid
```

4. The TranslateProperties property

When VB Migration Partner converts a form and all the controls it hosts from VB6 to .NET, it reads the design-time value for all the control's property from the .frm file, more precisely from the topmost portion of this file (that doesn't appear when you edit the form from inside the VB6 code editor).

The vast majority of properties appear in the .frm file with the same name you can use to read or assign the property from code, but in some cases the two names differ. If the properties differ you must instruct VB Migration Partner about the correspondence between the name in the .frm file and the name in the object's public interface. This information is conveyed in the **TranslateProperties** property of the VB6Object attribute, as a comma-delimited series of *frmname=codename* pairs.

If you notice that one or more properties of the ActiveX control aren't converted correctly by VB Migration Partner when it migrates a form, then you should open the .frm file with Notepad and check whether the properties in question have a name different from the name you see in the object browser.

Let's suppose that you realize that VB Migration Partner fails to correctly convert the design-time value of two properties – named HotTrack and Rows – of the fictitious XGrid control. In the VB6 form these properties are assign the values True and 12, respectively, but these values are lost in the migration.

Open the .frm file with Notepad and look for properties with similar name (and same assigned value) in the Begin… End block dedicated to this control. You might find something like:

```
…
Begin XLib.XGrid XGrid1
…
    HotTracking = -1    ' True
    RowCount = 12
    …
End
```

It is apparent that the HotTrack property is stored in the .frm with the name HotTracking, and that the Rows property is stored as RowCount. You now have all the necessary info to modify the VB6Object attribute to let VB Migration Partner know about this detail:

```
<VB6Object("XLib.XGrid", _
   TypeLibrary:="XLib", TypeLibraryNamespace:="XLibCtrl", _
   DependsOnAssemblies:="XLib.dll,AxXLib.dll", _
   TranslateProperties:="HotTracking=HotTrack,RowCount=Rows", _
   IgnoreMembers:="Negotiate|MaskColor|Palette|UsePalette")> _
Public Class VB6XGrid
```

5. Data-bound controls

By default, AxWrapperGen assumes that the ActiveX control supports ADODB databinding and generates a region of code named **"-- Data-binding support –"**, containing a basic implementation of all the members of the IVB6Control interface. This interface is defined inside CodeArchitects.VBLibrary.dll and is used by VB Migration Partner's support library to simulate data binding in the converted VB.NET form:

```
Public Class VB6XGrid
   Inherits AxXLib.AxXGrid
   Implements IVB6Control
   Implements ISupportInitialize
   Implements IVB6BoundControl


   ...

#Region "-- Data-binding support"

   ' TODO: remove this region if control doesn't support databinding

   ' << the implementation of DataBindings, DataChanged, DataField, DataFormat,
   '    DataMember, DataSource, RealDataSource, BoundValue, DataSourceOpen,
   '    and DataSourceClose members follows  >>

#End Region
```

***NOTE:** these code elements aren't generated if the ActiveX control exposes its own DataXxxx members.*

If you know for sure that the control does not support data-binding – or if you know that you never use data-binding features in your client forms – then it is a good idea to remove both the Implements statement and the code region.

If the ActiveX control does support data-binding and you want to leverage this feature even in migrated programs, then you should ensure that the BoundValue property (in the IVB6BoundControl interface) takes and returns the bound property. In most cases AxWrapperGen makes a correct guess about the name of the bound property, but in some cases you need to correct it. For example, let's suppose that the bound property for your control is named Value (Integer), whereas AxWrapperGen mistakenly assumed it was the Text string property:

```
' TODO: Change the name and type of bound property as necessary
Private Property BoundValue() As Object Implements IVB6BoundControl.BoundValue
    Get
        Return Me.Text
    End Get
    Set(ByVal value As Object)
        Me.Text = VB6Utils.ValueToBoundProperty(Of String)(value)
    End Set
End Property
```

Fixing this code is simple:

```
Private Property BoundValue() As Object Implements IVB6BoundControl.BoundValue
    Get
        Return Me.Value
    End Get
    Set(ByVal value As Object)
        Me.Value = VB6Utils.ValueToBoundProperty(Of Integer)(value)
    End Set
End Property
```

6. Support for "classic" drag-and-drop

By default, AxWrapperGen assumes that the ActiveX control supports "classic" (VB4-style) drag-and-drop and generates a basic implementation of all the members that can implement it, such as the DragIcon property and the Drag method):

```
Public Class VB6XGrid
   Inherits AxXLib.AxXGrid
   Implements ISupportInitialize
   Implements IClassicDragDrop

   ...

#Region "-- Drag-and-drop support"

   ' <<  implementation of DragIcon, DragMode properties, Drag method, and
   '     DragDrop, DragOver events follows

   ...

#End Region
```

If the original control doesn't support "classic" drag-and-drop – or if it does, but your client application doesn't use this feature – then you should remove both the Implements statement and the code region.

7. Rename events to avoid name collisions

It is legal for an ActiveX control to expose an event that has the same name as a property or a method. A common case is an ActiveX control that exposes a Format property and a Format event. Different members with same name aren't valid under .NET, thus you need to rename the event to avoid this collision.

Another reason for renaming an event is if its name matches a VB.NET keyword.

Regardless of the reason, you ***must*** rename the event, else the wrapper class can't compile. The naming convention that we recommend – and that we have used in our VBLibrary – is to append a "6" suffix to the original name to avoid the name collision.

For example, let's assume that the ActiveX control exposes an event named "Error", which is a reserved VB.NET keyword:

```
Public Shadows Event Error As VB6BeforePageBreakEventHandler

Private Sub Control_Error(ByVal sender As Object, ByVal e _
   As AxXLib._IAxGridEvents_ErrorEvent) Handles MyBase.Error
       VB6ErrorEventDispatcher.Raise(Me, "Error", e.errorCode, e.showMsgBox)
End Sub
```

The first thing to do to rename the event is changing the event name:

```
Public Shadows Event Error6 As VB6BeforePageBreakEventHandler

Private Sub Control_Error(ByVal sender As Object, _
    ByVal e As AxXLib._IAxGridEvents_ErrorEvent) Handles MyBase.Error
    VB6ErrorEventDispatcher.Raise(Me, "Error6", e.errorCode, e.showMsgBox)
End Sub
```

Next, you have to make VB Migration Partner aware that the Error event has now a different name. You do so by editing the **TranslateEvents** property of the VB6Object attribute:

```
<VB6Object("XLib.XGrid", _
   TypeLibrary:="XLib", TypeLibraryNamespace:="XLibCtrl", _
   DependsOnAssemblies:="XLib.dll,AxXLib.dll", _
   TranslateEvents:="Error=Error6", _
   IgnoreMembers:="Negotiate|MaskColor|Palette|UsePalette")> _
Public Class VB6XGrid
```

If you rename more than one event, you should use a comma-delimited list of value pairs, as in:

```
TranslateEvents:="Error=Error6,Format=Format6"
```

8. Properties with multiple overloads

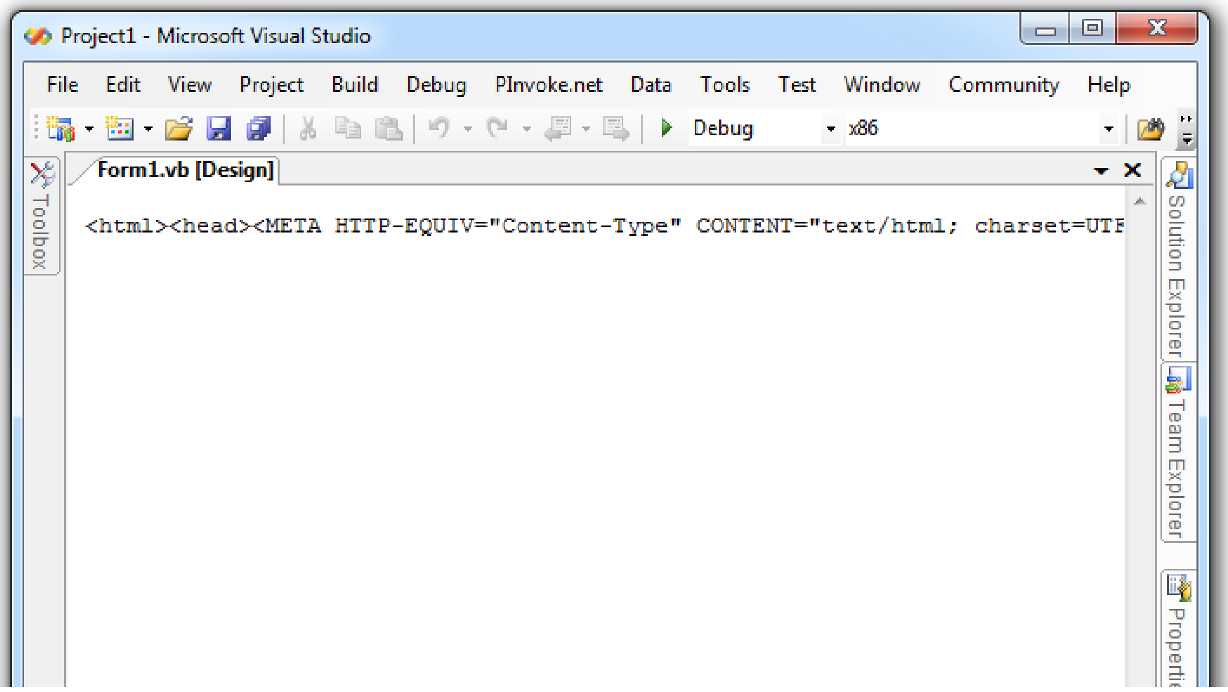
If the ActiveX control exposes a property with optional parameters, then the wrapper class will expose two or more overloads of that property. If you later use the wrapper class to convert forms containing the ActiveX control, the .NET form can't be edited inside Visual Studio. The solution is to remove the overloads by manually merging them back to a property with optional parameters.

Let's see a practical example. When you run AxWrapperGen on the VideoSoft VSFlexGrid control, the generated wrapper class contains this code:

```
Public ReadOnly Property ArchiveInfo(ByVal arcFileName As String, _
      ByVal infoType As VSFlex8L.ArchiveInfoSettings) As Object
    Get
        Return MyBase.get_ArchiveInfo(arcFileName, infoType)
    End Get
End Property

Public ReadOnly Property ArchiveInfo(ByVal arcFileName As String, _
   ByVal infoType As VSFlex8L.ArchiveInfoSettings, ByVal index As Object) As Object
    Get
        Return MyBase.get_ArchiveInfo(arcFileName, infoType, index)
    End Get
End Property
```

The wrapper class compiles correctly and you can use it to migrate VB6 forms that contain the VSFlexGrid control; the generated form runs and behaves correctly and everything seems to be working fine. However, if you later try to modify the .NET form inside Visual Studio you see an error message.


Project1 - Microsoft Visual Studio
File Edit View Project Build Debug PInvoke.net Data Tools Test Window Community Help
Debug
x86
Form1.vb [Design]
Toolbox
Solution Explorer
Team Explorer
Propertie

```
<html><head><META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=UTF
```

The solution is merging the two code blocks in a single property by making the 3rd argument Optional and writing the code that delegates to either get_ArchiveInfo method in the base class, depending on whether the optional argument has been omitted:

```
Public ReadOnly Property ArchiveInfo(ByVal arcFileName As String, _
      ByVal infoType As VSFlex8L.ArchiveInfoSettings, _
      Optional ByVal index As Object = Nothing) As Object
```

```
        Get
            If index Is Nothing Then
                Return MyBase.get_ArchiveInfo(arcFileName, infoType)
            Else
                Return MyBase.get_ArchiveInfo(arcFileName, infoType, index)
            End If
        End Get
    End Property
```

9. Wrapper class members with names not found in original ActiveX control

As explained, AxWrapperGen runs AxImp.exe behind the scenes. AxImp.exe may change the name of a member if the AxHost base class exposes a member with same name. For example, a property named EditMode might be converted into CtlEditMode, and Text might be converted into CtlText.

If you notice that one or more members in the wrapper class have a name that can't be found in the original ActiveX control, then it is necessary to edit the TranslateMembers property of the VB6Object attribute accordingly. This property is then used by VB Migration Partner during the conversion process to detect which member names must be modified when generating the client form's source code. The value of the TranslateMembers attribute is a comma-delimited list of *oldname=newname* pairs, as in:

```
    <VB6Object("XLib.XGrid", _
       TypeLibrary:="XLib", TypeLibraryNamespace:="XLibCtrl", _
       DependsOnAssemblies:="XLib.dll,AxXLib.dll", _
       TranslateMembers:="Text=CtlText,EditMode=CtlEditMode", _
       IgnoreMembers:="Negotiate|MaskColor|Palette|UsePalette")> _
    Public Class VB6XGrid
```

Quite conveniently, AxWrapperGen creates a list of members that are candidate for this treatment. You can find it in section #9 of the "Instructions" region:

```
    '       Candidate members for this class are: CtlText, CtlEditMode
```

10. ScaleMode-related properties

Properties and members that are related to position and size values require minor manual adjustments. For example, our fictitious XGrid control might expose a property named RowHeight and a method named SetTitleBar:

```
    Public Property RowHeight() As Single
        Get
            Return MyBase.RowHeight
        End Get
        Set(ByVal value As Single)
```

```
            MyBase.RowHeight = value
        End Set
    End Property

    Public Sub SetTitleBar(ByVal caption As String, ByVal left As Single, _
          ByVal top As Single, ByVal width As Single, ByVal height As Single)
        MyBase.SetTitleBar(caption, left, top, width, height)
    End Sub
```

The problem with position- and size-related members such as these is that – in most cases – the ActiveX controls expects values expressed in pixels, whereas the client application uses values that are expressed in the current ScaleMode setting. By default, these values are in twips, but the client app might have set a different ScaleMode, including user-defined modes.

You understand that a property requires to be scaled from pixels to a different measure unit if you see that the control stretches or shrinks abnormally after the migration. For example, if a grid column is about 15x larger than it should, then the problem is likely to be a missed conversion from twips to pixels.

The **VB6Utils** helper class – defined in CodeArchitects.VBLibrary.dll – exposes four methods that you can use to convert between pixels and the current ScaleMode setting: **FromPixelX**, **FromPixelY**, **ToPixelX**, and **ToPixelY**. The first argument is the current control, the second argument is the value to be converted, and the third argument must be False if you are converting a position value or True if you are converting a size value. The FromPixelY and ToPixelY methods take a fourth, optional argument that must be True if the height of the menu on the parent form should be ignored:

Here's how the RowHeight and SetTitleBar members look like after the fixes.

```
    Public Property RowHeight() As Single
        Get
            Return VB6Utils.FromPixelY(Me, CInt(MyBase.RowHeight), True, True)
        End Get
        Set(ByVal value As Single)
            MyBase.RowHeight = VB6Utils.ToPixelY(Me, value, True, True)
        End Set
    End Property

    Public Sub SetTitleBar(ByVal caption As String, ByVal left As Single, _
       ByVal top As Single, ByVal width As Single, ByVal height As Single)
        MyBase.SetTitleBar(caption, VB6Utils.ToPixelX(Me, left, False), _
           VB6Utils.ToPixelY(Me, top, False, True), _
           VB6Utils.ToPixelX(Me, width, True), _
           VB6Utils.ToPixelY(Me, height, True, True))
    End Sub
```

11. Readonly, non-browsable, and transient properties

By default, all the public properties defined in the wrapper class appear in Visual Studio's Property window and are stored in the form's code-behind section (i.e. the *.Designer.vb file that Visual Studio creates for each form). You need to take a few additional steps if you want to hide a property in the property browser and/or prevent that its value be saved in the code-behind section (*transient* property).

Hiding a property in Visual Studio's Property window is as simple as marking the property with a **Browsable** attribute. Likewise, preventing a property value from being saved in the code-behind section requires that you mark the property with a **DesignerSerializationVisibility** attribute, as in this example:

```
<Browsable(False)> _
DesignerSerializationVisibility(DesignerSerializationVisibility.Hidden)>  _
Public ReadOnly Property VisibleRowCount() As Single
    Get
        Return MyBase.VisibleRowCount
    End Get
End Property
```

***NOTE:** good candidates for the Browsable attribute are all readonly properties, unless they return an object or a collection.*

In some cases you also need to mark the property with the **Shadows** keyword, as in:

```
Public Shadows ReadOnly Property VisibleRowCount() As Single
    ...
End Property
```

12. Properties with ByRef arguments

Under COM and VB6 you can define properties that take a ByRef argument, whereas.NET requires that properties only take ByVal arguments. Conveniently, AxWrapperGen flags all properties with one or more ByRef arguments with a special TODO comment:

```
' TODO: .NET properties cannot have ByRef params. Manually fix code as necessary.
Public ReadOnly Property RowStatus(ByRef index As Integer) As Integer
    Get
        Return MyBase.RowStatus(index)
    End Get
End Property
```

You can quickly find all such comments in Visual Studio's Task List window.

Basically, there are two ways to fix this problem. First, you should read the ActiveX documentation and check that – even if passed by-reference – the control never modifies the argument being passed. If this is the case you can safely modify the ByRef keyword into ByVal.

```
Public ReadOnly Property RowStatus(ByVal index As Integer) As Integer
    Get
        Return MyBase.RowStatus(index)
    End Get
End Property
```

In other cases the ByRef semantics must be preserved, therefore you must split the property into two separate **get_*propertyname*** and **set_*propertyname*** methods, which don't suffer from the ByRef restriction:

```
Public Function get_RowStatus(ByRef index As Integer) As Integer
    Return MyBase.get_RowStatus(index)
End Property

Public Sub set_RowStatus(ByRef index As Integer, ByVal value As Integer)
    MyBase.set_RowStatus(index, value)
End Sub
```

In this latter case, be prepared to modify the code in the client form to account for these two new methods.

13. Size properties that aren't migrated correctly

You might experience the following problem: the ActiveX control is migrated correctly and appears to work as expected, except its size is different from the original or its size isn't preserved when the .NET form is modified inside Visual Studio's form designer.

In this case, you should uncomment two lines in the support class that AxWrapperGen has created for the ActiveX control:

```
' TODO: uncomment following statements if control doesn't retain the Width/Height
' properties set size properties which will be serialized in the ocx state

control.Width = CInt(comp.GetPropertyValue("Width", 0) / _
   Microsoft.VisualBasic.Compatibility.VB6.TwipsPerPixelX)
control.Height = CInt(comp.GetPropertyValue("Height", 0) / _
   Microsoft.VisualBasic.Compatibility.VB6.TwipsPerPixelY)
```

Troubleshooting

This section summarizes the most common causes for malfunctioning. Most of them have been already described in depth elsewhere in this document.

The first thing to do if anything doesn't work correctly

In some cases, an ActiveX control simply can't be translated to .NET and the blame isn't on AxWrapperGen. When you find out that anything isn't working as described in this document, the first thing to do is attempting to add a reference to the ActiveX control from Visual Studio and – if successful – drop an instance of the control onto a .NET form.

If any of these operations fail, then you just can't create a .NET wrapper class for that control using AxImp or AxWrapperGen. The only solution is to create a wrapper class manually, by inheriting from a .NET control. (This approach is described later in this document.)

More in general, whenever you see that the AxWrapperGen-generated wrapper class doesn't work as expected, your first attempt should be trying to use the same feature on an ActiveX control that you manually dropped on a .NET form inside Visual Studio. If you get the same or similar error, then the problem is likely to be in Microsoft COM Interop technology.

VB Migration Partner doesn't correctly convert the ActiveX control

There are a few common causes for this problem:

- If VB Migration Partner still fails to recognize the compiled DLL that contains the wrapper class, odds are that the control name specified in the VB6Object attribute doesn't match the ActiveX control name as shown by VB6 object browser. (See paragraph 1 in previous section.)
- If the names match, then the next likely cause for this problem is that you failed to correctly deploy it in a place where VB Migration Partner can find it. For more details, double-check the "Deploying the wrapper class" section.
- Another possible cause for this problem is that your application consists of multiple project and they use different versions of the ActiveX control. Read this KB article for more information.
- The path of the ActiveX control specified in the project .vbp file isn't correct. To ensure that the contents of the VBP file is up-to-date, please load the project in the VB6 IDE and then save it again.

VB Migration Partner works errantly after deploying the wrapper class

This problem usually occurs if you deployed the wrapper class DLLs to VB Migration Partner's setup folder. Please notice that we recommend that you instead deploy the DLLs to a folder pointed to by an AddLibraryPath pragma, as explained in the "Deploying the wrapper class" section.

The ActiveX control is recognized but its design-time properties aren't initialized correctly

ActiveX controls store their properties in the code-behind section of a form – that is, the *.Designer.vb file – like standard .NET controls, except they use to pack all property values in a special binary property named **OcxState**. During the conversion process, VB Migration Partner generates the value for the OcxState property, so that the control can restore all its properties when it is instantiated on a .NET form.

In some (rare) cases, VB Migration Partner fails to generate a correct value for the OcxState property, often because the specific control uses a nonstandard mechanism to store its properties. If you realize that the control's design-time properties aren't initialized correctly or if the form fails to run correctly, you should attempt to use Microsoft Upgrade Wizard (included in Visual Studio 2005 and 2008) to migrate that form and generate a valid OcxState, which you can later reuse with VB Migration Partner by means of the **ReuseResxFiles** pragma.

For more information, please read this KB article: [PRB] ActiveX controls aren't migrated correctly.

The generated .NET form cannot be displayed in Visual Studio designer

There can be many causes for this issue. The most common ones are:

- VB Migration Partner has generated one or more properties in the *.Designer.vb that shouldn't be there. For example, we have met this problem with a property named LicenseKey, which refers to COM licensing mechanism that can't be used from inside .NET. Such spurious properties typically cause an exception when you display the form at design-time or when you run the VB.NET project. In such cases you must prevent VB Migration Partner from generating a property with that name, by means of the IgnoreMembers property of the VB6Object attribute. (See paragraph 3 of previous section.)

- The wrapper class exposes one or more overloaded properties. (See paragraph 8 of previous section.)
- The name of a property used in the hidden portion of a VB6 .frm file differs from the property name that appears in VB6 object browser. (See paragraph 4 of previous section.)
- One or more transient properties should be marked with a DesignerSerializationVisibility attribute but aren't. (See paragraph 11 of previous section.)

The ActiveX control is migrated, but its size (or the size/position of its UI elements) is incorrect, either at design-time or runtime

There can be two causes for this issue:

- The control exposes properties or methods that take values that are affected by the parent form's ScaleMode. In this case you should fix the wrapper class as described in paragraph 10 of previous section.
- If the control's size appears different at design-time – usually much larger than it should – then you should edit the support class, as described in paragraph 13 of previous section.

The code in client form contains compilation errors

The most common causes for this issue are:

- The name of a property used in the hidden portion of a VB6 .frm file differs from the property name that appears in VB6 object browser. (See paragraph 4 of previous section.)
- You changed the name of an event to prevent name collisions but forgot to edit the TranslateEvent property of the VB6Object attribute. (See paragraph 7 of previous section.)
- AxImp changed the name of one or more members of the ActiveX control (See paragraph 9 of previous section.)

The client application crashes with COM error 80040154

You are likely to run the client as a 64-bit application. Read this KB article for more information.

Databinding doesn’t work

If the converted control appears to work correctly except that data-binding features throw an exception or silently fail, odds are that you are using a control that binds to a DAO or RDO Data control. These controls can’t be migrated correctly, either by Upgrade Wizard or VB Migration Partner. (See note at the end of “Running AxWrapperGen” section.)

The ActiveX control can’t be instantiated dynamically with the Controls.Add method

Read paragraph 2 in “Fixing and polishing the wrapper class” section.

Inheriting from a .NET control

Once your wrapper class works well, you can use VB Migration Partner to convert all the VB6 projects that use that ActiveX control. However, keep in mind that the resulting .NET project will still depend on COM and ActiveX legacy technologies, which is usually undesirable in the long run.

To get rid of this dependency you must edit the wrapper class so that it inherits from a “pure” .NET control rather than the ActiveX control. This section describes all the steps you must take to do the replacement.

1. Select the most suitable .NET control

The first thing to do is deciding which .NET control you want to use to replace the legacy ActiveX control. In general, you should select a .NET control that has all the features that you need (among those exposed by the ActiveX control) and, preferably, a control that exposes an object model that is similar to the programming interface of the ActiveX control.

If the vendor of the original ActiveX control is still in business and if it released a .NET version of the original control, then the decision is obvious. This is the case, for example, of the majority of the controls that were sold by companies such as Sheridan (which changed its name into Infragistics), or Apex and VideoSoft (which joined into the company named ComponentOne).

Many VB6 developers used 3rd-party ActiveX controls only because the corresponding VB6 built-in control wasn’t powerful enough. This is the case, for example, of the many “super textbox” controls offered by companies such as Crescent and MicroHelp. Most of these ActiveX controls can be conveniently replaced by the controls that are included in the System.Windows.Forms namespace, in which case you don't have to purchase any additional control.

2. Change the base class

Next you must edit the **Inherits** statement, so that the wrapper class now inherits from the selected control. For example, let's suppose that you found out that the fictitious XGrid control can be replaced by the DataGridView control. The Inherits clause therefore becomes:

```
<VB6Object("XLib.XGrid", _
   TypeLibrary:="XLib", TypeLibraryNamespace:="XLibCtrl", _
   IgnoreMembers:="")> _
Public Class VB6XGrid
   Inherits System.Windows.Forms.DataGridView
```

Also notice that you should delete the DependsOnAssemblies property of the VB6Object attribute, to inform VB Migration Partner that this class doesn't depend on the ActiveX DLLs any longer.

3. Remark out non-essential members

As soon as you change the Inherits class, many compilation errors will appear in the wrapper class when the **MyBase** keyword is used, because the new .NET control has programming interface that differs from the ActiveX control.

You can temporarily solve all these compilation errors by remarking out all the properties and methods of the wrapper class, except the ones listed below:

- The **Name**, **Index**, **Parent**, and **Container** properties are part of the IVB6Control interface, which must be exposed by all controls that are converted by VB Migration Partner. The code that AxWrapperGen emits for these properties is correct, thus you don't need to remark them out.
- The **Left, Top, Width**, and **Height** properties are always error-free and are mandatory, therefore you don't need to remark them.
- The **ZOrder**, **Move**, and **Focus** methods are always common to all controls and don't need to be commented.
- The list of members that you can safely leave uncommented includes **ToolTipText**, **MousePointer**, **MouseIcon**, **Object**, and all **Font*Xxxx*** properties.

4. Clean up the support class

Now that the ActiveX control inherits from a .NET control, the support class that AxWrapperGen generated must be modified accordingly, by removing all references to the ActiveX control. After cleaning up the support class should look like the following code:

```
 <VB6ControlSupport(GetType(VB6XGrid))> _
Public Class VB6XGrid_Support
    Inherits VB6ObjectSupportBase

    Public Overrides Function ParseProperties(ByVal objInfo As Object) As Object
        On Error Resume Next

        Dim info As ControlSupportInfo = DirectCast(objInfo, ControlSupportInfo)
        Dim comp As VBComponent = info.Component

        ' TODO: extract all binary properties here

        ' don't add any new component
        Return Nothing
    End Function

    Public Overrides Function  GeneratePropertyCode(ByVal objInfo As Object) As String
        Dim info As ControlSupportInfo = DirectCast(objInfo, ControlSupportInfo)

        ' TODO: Append here more code if necessary. Each property rendered  manually
        '       should be added to the ExcludedProperties  collection, as in:
        '       info.Component.ExcludedProperties.Add("propertyname")
        Return ""
    End Function

End Class
```

NOTE: *as modified in this example, the support class does nothing and might be completely removed from the DLL that contains the wrapper class. Don't do that now, however, because you might later find out that some nonstandard actions must be necessary during the migration process.*

5. Determine the list of used members

An ActiveX control – especially complex controls such as grids and charting controls – expose literally hundreds of properties, methods, and events. In general, you don't want the wrapper class implement all of them, because it would be a huge waste of time in most cases. The members that you really want to implement are only those that your VB6 client projects actually use.

You therefore need to determine what are the members that you really need to implement. There are at least three ways to obtain this list:

- You compile the wrapper class (with most members still remarked out – see previous point), use VB Migration Partner to migrate the VB6 project(s) and take note of all compilation errors caused by the members that the wrapper class should expose but doesn't.
- You use a code analysis tool to analyze the VB6 code and automatically create this list for you. One of the best VB6 code analysis tool we know is VBDepend, so you can download a demo version and check how well it can create this list for you.
- You just browse the VB6 source code and manually take note of all the ActiveX members that the wrapper class needs to implement. (This approach is ok only if the ActiveX control is used sparingly.)

Now that you know which properties and methods the wrapper class must implement, you can start uncommenting them, one at a time, while fixing the statement that delegates the call to the inner base class. In completing this task, you can face many different situations, which are described in following numbered paragraphs.

6. Delete members with same name, syntax, and behavior

In the simplest case, a member has exactly the same name, syntax, and behavior in the ActiveX control and in the selected .NET control. For example, Boolean properties named Enabled, Visible, and ReadOnly are likely to work in the same way in the two controls:

```
Public Property Enabled() As Boolean
    Get
        Return MyBase.Enabled
    End Get
    Set(ByVal value As Boolean)
        MyBase.Enabled = value
    End Set
End Property
```

In this very favorable case, you can just delete the entire property or method, because it will be correctly inherited from the base .NET control.

7. Implement members with different name or syntax, but same behavior

In most cases, the .NET control implements the same functionality of the original control by means of a property or method that has a different name and/or syntax, but that behaves exactly like in the original ActiveX control.

For example, an ActiveX control that implements a "super TextBox" control usually exposes the SelText, SelStart, and SelLength properties, which correspond to the SelectedText, SelectionStart, and

SelectionLength properties of the .NET TextBox control. You can therefore uncomment the code for these properties and simply replace the name of the property:

```
Public Class VB6SuperText
    Inherits System.Windows.Forms.TextBox

Public Property SelText() As String
    Get
        Return MyBase.SelectedText
    End Get
    Set(ByVal value As String)
        MyBase.SelectedText = value
    End Set
End Property

' similar code for SelStart and SelLenght properties
' ...
```

In other cases the property has a slightly different syntax. For example, the VB6 RightToLeft property takes a Boolean value, whereas the .NET RightToLeft property takes an enumerated value:

```
Public Shadows Property RightToLeft() As Boolean
    Get
        ' return True if inner .NET control returns "Yes", else return False
        Return (MyBase.RightToLeft = System.Windows.Forms.RightToLeft.Yes)
    End Get
    Set(ByVal value As Boolean)
        If value Then
            MyBase.RightToLeft = System.Windows.Forms.RightToLeft.Yes)
        Else
            MyBase.RightToLeft = System.Windows.Forms.RightToLeft.No)
        End If
    End Set
End Property
```

8. Implement events

The implementation of events greatly differs between similar VB6 and .NET controls. For example, all .NET events take two arguments – the *sender* Object reference and the *e* object, of type EventArgs or a type that inherits from EventArgs. For example, this is the code that AxWrapperGen generates for the Click event of most ActiveX controls:

```
Public Shadows Event Click As VB6EventHandler

Sub Control_Click(ByVal sender As Object, ByVal e As EventArgs) Handles Control.Click
   VB6EventDispatcher.Raise(Me, "Click")    ' no additional arguments
End Sub
```

As you see, this code intercepts the Click event coming from the inner .NET control and forwards it to all control's clients that have registered this event. This latter forwarding action ***must*** be performed by means of the VB6EventDispatcher.Raise method, else the control might fire events at the wrong time (for example, while the form is being loaded).

In most cases, these code sections can be uncommented and they will work correctly. In other cases you might need to edit the code to account for event arguments (which aren't needed in this specific case).

9. Implement remaining members

The cases discussed in paragraphs 6, 7, and 8 above account for the vast majority of members, but not for all of them. In remaining cases you must perform the "mapping" between the outer member (that has the original ActiveX name and syntax) and the inner member (as exposed by the .NET base control).

There are so many cases that it is virtually impossible to describe how to handle each of them. However, the key point is that the wrapper class mechanism is powerful enough to allow you to account for things such as different member names, syntax, and behavior. For example, you can alter the order in which events fire or can even prevent an event from firing.

10. Re-deploy the wrapper class DLL

You can now compile the ".NET-only" wrapper class to a DLL and deploy the DLL as you did with the old DLL, as described in the "Deploying the wrapper class" section.

In most cases all migrated projects will work exactly as before. In some cases, however, it might be necessary that you manually adjust projects references, to ensure that they point to the new DLL.

In some rare circumstances you might also need to convert again the client project(s). This is necessary, for example, if you modified the TranslateProperties, IgnoreMembers, or TranslateMembers properties of the VB6Object attribute.

Conclusion

The wrapper class approach to ActiveX control conversion and replacement is powerful enough to allow you to face even the most intricate situations.

Writing a bullet-proof wrapper class is simple, because most of the job is performed by the AxWrapperGen utility, yet you must be prepared to handle some special cases yourself. This document – especially its Troubleshooting section - should be your first reference when something doesn't work as expected.

If you meet an issue that isn't covered by this article, please send us a note: we will help you in migrating your specific control and will extend this document so that other VB Migration Partner users can find a solution to similar problems.