WHITE PAPER<

A smart approach to 3rd-party ActiveX control conversion

The vast majority of VB6 business applications use one or more 3rd-party (non-Microsoft) ActiveX controls. It is therefore essential that the VB6 conversion software of your choice be able to correctly convert these controls to .NET with a cost-effective approach.

This document illustrates how VB Migration Partner can perform this task using an innovative approach based on **wrapper classes**. It intends to prove that - unlike more traditional approaches based on code transformations and mapping techniques - only wrapper classes can account for all minor and minor differences between ActiveX and .NET controls, can generate concise and efficient code, and don't cause bottlenecks in the migration project.

Table of Contents

ActiveX control migration and replacement

In describing how VB Migration Partner and other similar tools work it is essential to agree on what "*converting an ActiveX control to .NET*" really means. In fact, there are two distinct levels for this conversion. For the sake of clarity, we will use the terms ***ActiveX migration*** and ***ActiveX replacement***, even though in a somewhat arbitrary fashion. In the context of this document:

- **ActiveX control migration** is the process of generating VB6 forms into .NET forms that still reference the same ActiveX control, via COM Interop.

- **ActiveX control replacement** is the process of substituting the migrated ActiveX control with a pure .NET control, be it a native Microsoft .NET control, a 3rd-party .NET control, or a custom-made .NET control.

ActiveX migration is made possible by COM Interop, the portion of .NET Framework that allows .NET apps to communicate with COM components and ActiveX controls. COM Interop is a great technology that works unbelievably well considering that COM and .NET are completely different worlds. However, COM Interop has a few issues and shortcomings, which explain why ActiveX replacement is nearly always necessary.

In describing the ActiveX conversion process, it is useful to distinguish between two different kinds of ActiveX controls and components, depending on whether the ActiveX control is part of the user interface:

- **Invisible ActiveX controls** are controls that you place on VB6 forms, so that you can set their attributes in the Properties window, yet they don't produce a visible user interface at runtime. Examples of these controls are MSComm (for serial communications), WinSock (to TCP and UDP communications), or Microsoft Script Control.
- **Standard (visible) ActiveX** controls are the usual controls that you used to enrich your VB6 apps with a better user interface, for example MSDataGrid, MSChart, MSCalendar. The majority of 3rd-party controls also belong to this categories, for example the VideoSoft VSFlexGrid, Apex TrueDBGrid, or FarPoint Spread.

COM Interop works very well with COM type libraries and ActiveX controls that have no user interface, therefore it is usually considered acceptable for migrated apps to preserve all references to type libraries and invisible ActiveX controls.

Conversely, COM Interop may have problems with more complex ActiveX controls, such as grids and charting controls. These controls may behave errantly or crash the application when hosted on a .NET form, therefore it is highly recommended that the migrated .NET application have no reference to the original ActiveX controls and that you replace all ActiveX controls with equivalent, fully managed .NET controls.

Even if your ActiveX controls don't crash the application, many 3rd-party controls don't display their UI elements correctly under recent versions of Microsoft Windows.

Another reason for replacing ActiveX controls is that ***COM and ActiveX are inherently 32-bit technologies*** that don't work well (or at all) under 64-bit platforms. While you are facing the task of converting your code to .NET, you should consider it as your best opportunity to get rid of any dependency from older 32-bit CPUs.

Databinding is another reason for replacing all ActiveX controls with their .NET equivalent. If the original VB6 control uses databinding to display and modify data in an ADODB.Recordset object, this control won't work any longer if you replace the Recordset with the ADO.NET DataSet object. The same consideration holds true if your VB6 application uses other COM-based data access technologies. In general, if you are switching from ADODB, DAO, or RDO to ADO.NET, then you have to replace all your databound ActiveX controls with equivalent ADO.NET controls that are bound to the DataSet object.

Finally, ActiveX controls require special care in installation and deployment. If your .NET application depends on one or more ActiveX controls, then it will be subject to a form of the DLL Hell problem that has plagued VB6 developers for years. If another application that uses a different (minor) version of the same control is installed after yours, odds are that your application stops working or behaves errantly. For this reason, your ***final goal should be a .NET application that has no dependency on COM and ActiveX components***.

Challenges in ActiveX replacement

Regardless of how similar a VB6 control and its corresponding .NET control can be, there will always differ in tons of major and minor details, including

- **Different member name:** for example, the SetFocus method is replaced by the Focus method under .NET, hWnd corresponds to Handle property, and so forth.
- **Different syntax:** all events and many methods have a different syntax under .NET.
- **Different behavior:** for example, the ValidateControls VB6 method can be translated with the Validate method under .NET, but the former raises an error whereas the latter just returns False if validation fails.
- **Different appearance:** there can be minor differences in how the VB6 and .NET controls display themselves, and sometimes these differences cannot be ignored. For example, System.Windows.Forms.Button .NET control of same size and same caption as a VB6 CommandButton has a smaller client area and therefore often truncates its inner caption.
- **Different event order:** .NET controls don't necessarily fire the same events as VB6 controls, or they fire events in a different order; for example, the order in which the GotFocus and LostFocus events fire under VB6 and .NET is different, a detail that may lead to many subtle bugs.
- **Missing members:** for example, .NET controls expose neither graphic methods (e.g. PSet, Line, and Circle) nor drag-and-drop members that are comparable to those you can find in VB6 controls.
- **Missing features:** .NET controls don't expose DDE-related members; position and dimension properties are always expressed in pixels and .NET offers no support for user-defined coordinate systems; also, no .NET property can replace the AutoRedraw property, just to mention a few key missing features.
- **Late binding:** the migrated control should work fine even if the name or syntax of one or more members has changed from VB6 to .NET
- **Split and combined members:** some VB6 members have been split in two different .NET members; for example, the ZOrder VB6 method maps to either BringToFront or SendToBack .NET methods. Likewise, VB6 members may have been combined into a single .NET property, method, or event, as is the case with the QueryUnload and Unload VB6 events that have been merged into the FormClosing .NET event.

- **Databinding:** VB6 controls can be bound to ADODB, DAO, or RDO data source – for example, the ADODB.Recordset object or the ADODC control - whereas .NET controls can be typically bound only to ADO.NET sources (e.g. DataReader, DataView, or DataSet objects).

NOTE: Please read this whitepaper for a more complete list of the differences between VB6 and .NET built-in controls. The majority of these differences also affect ActiveX controls.

When you replace an ActiveX control with an approximately equivalent .NET, you must find a way to account for all these (and other) differences, and you are eventually going to write plumbing code to force the .NET control to behave like the original VB6 code, so that you preserve functional equivalence and the application's look-and-feel.

Unfortunately, no conversion software on the market can write this plumbing code for you, because there are hundreds of ActiveX controls around and each of them supports hundreds of properties, methods, and events. It is simply impossible to account for all members of all existing ActiveX controls out there.

Therefore, when evaluating VB Migration Partner or a similar conversion tool from another vendor, the question to ask should ***not*** be "*Does your software support the XYZ ActiveX control?*" Instead, a more appropriate question is *"How much extra code am I expected to write to support the XYZ ActiveX control?"*

In the remainder of this document we will describe how VB Migration Partner answers this question and compare its approach with the solutions offered by other VB6 conversion tools.

How Upgrade Wizard deals with ActiveX controls

Before describing the approach that VB Migration Partner adopts to migrate ActiveX controls and eventually replace them with .NET controls, let's quickly revise how most VB6 conversion tools work in this respect, including the Upgrade Wizard – included in Visual Studio 2008 and previous versions, but not in Visual Studio 2010 – and other conversion tools based on the same conversion engine.

Behind the scenes, Upgrade Wizard runs the **AxImp** utility to generate a DLL that contains the binary wrapper for that control. (AxImp is a command-line tool that is part of .NET Framework SDK). Next, Upgrade Wizard converts individual VB6 forms and generates .NET forms that use this binary wrapper.

Technically speaking, Upgrade Wizard creates .NET applications that indirectly reference the original ActiveX control through a wrapper class that is exposed by the AxImp-generated DLL. For all practice purposes, generated .NET applications use the original ActiveX controls, with all the issues that this approach brings with it. It is worth noticing that the AxImp-generated wrapper class is compiled in the DLL and you can't see nor modify its source code.

As explained previously, it is highly recommended that you replace these ActiveX references with references to native .NET controls. Unfortunately, Upgrade Wizard offers no support for this replacement stage. In practice, you have to manually replace all ActiveX controls with the .NET control of your choice.

Worse, manual replacement usually introduces tons of syntax and compilation errors – depending on how different the original ActiveX control and the selected .NET control are. You have to manually fix all these errors before proceeding; again, Upgrade Wizard doesn't offer any kind of support for this lengthy and error-prone job.

In summary, Upgrade Wizard does an acceptable job as far as ActiveX control migration is concerned, but offers no support for ActiveX control replacement. Given that replacement is the final stage of all real-world migration projects, it is safe to conclude that Upgrade Wizard isn't a serious candidate tool for dealing with ActiveX controls.

How VB Migration Partner deals with ActiveX controls

VB Migration Partner fully supports over 60+ controls right out-of-the-box; including all controls that come with VB6 and many others that you could download from Microsoft web site. This group includes the Threed library (SSButton, SSCheck, etc.), the MSWLess library (WLText, WLCombo, etc.), and the Microsoft Script Control.

NOTE: *the only Microsoft controls that VB Migration Partner doesn't support are: the OLE container control, the DataRepeater, and the Coolbar.*

All controls that aren't part of the abovementioned group should be considered as not directly supported by VB Migration Partner.

Immediately after you install VB Migration Partner, it knows how to neither migrate nor replace these unsupported ActiveX controls: if you convert a VB6 form that contains unsupported controls, VB Migration Partner replaces those controls with generic **VB6Placeholder** controls, which look like red rectangles on the converted .NET form.

The inability to migrate third-party controls can be disappointing at first, especially because even the lowly Upgrade Wizard can perform this task without any manual labor. On the other hand, as you'll see shortly, VB Migration Partner's approach is far more flexible, allows you to save time and effort, doesn't cause bottlenecks in the migration process, and greatly simplifies the debug and test phases.

ActiveX control migration

A smart approach to 3rd-party ActiveX control conversion

Here's a practical example: let's say that you want to add support for the MSCalendar control. (This example is for illustration purposes only, because MSCalendar is the list of controls that VB Migration Partner natively supports.)

The first step towards ActiveX control migration is launching **AxWrapperGen**, a utility that is installed with VB Migration Partner. This command-line tool takes the name of the *.ocx file that contains the ActiveX, the name of the VB.NET project to be created (/project option), and the path of the folder where the VB.NET project will be created. For example, this command

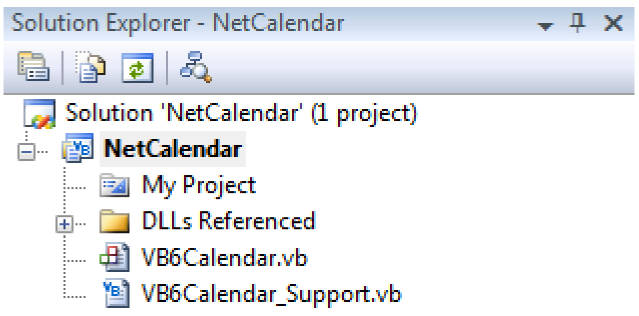
```
AxWrapperGen c:\windows\system32\mscal.ocx /out:c:\myapp /project:NetCalendar
```

creates a wrapper class for the mscal.ocx ActiveX control (the MSCalendar control), as part of a VB.NET project named NetCalendar stored in the c:\myapp folder.

The NetCalendar.vbproj project contains two classes, **VB6Calendar** and **VB6Calendar_Support**. The former inherits from AxMSACAL.AxCalendar and wraps the ActiveX control; the latter is an auxiliary class that is only used during the migration step to control how VB Migration Partner generates code. (This support class isn't relevant for our discussion and won't be analyzed in this article.)

NOTE: *AxWrapperGen generates a pair of classes for each ActiveX control defined in the OCX file. In this example only a class pair was generated, because the mscal.ocx file contains only the MSCalendar ActiveX control.*

Like the Upgrade Wizard, AxWrapperGen runs the AxImp utility behind the scenes. The resulting NetCalendar VB.NET references the two DLLs that were generated by AxImp, MSACAL.dll and AxMSACAL.dll:



This is how the VB6Calendar class looks like:

```
<VB6Object("MSACAL.Calendar", _
DependsOnAssemblies:="CodeArchitects.AxVBLibraryOCX.dll,MSACAL.dll")> _
Public Class VB6Calendar
    Inherits AxMSACAL.AxCalendar
    ...
End Class
```

For the most part, the code inside the VB6Calendar class consists of simple wrapper properties and methods that delegate to the corresponding property or method exposed by the inner ActiveX control (the AxMSACAL.AxCalendar class). For example, this is how AxWrapperGen generates the code for the FirstDay property:

```
Public Property FirstDay() As Short
    Get
        Return MyBase.FirstDay
    End Get
    Set(ByVal value As Short)
        MyBase.FirstDay = value
    End Set
End Property
```

AxWrapperGen knows that some standard VB6 properties and methods correspond to .NET members with different name:

```
' The VB6 SetFocus method maps to .NET Focus method
Public Sub SetFocus()
    MyBase.Focus()
End Sub
```

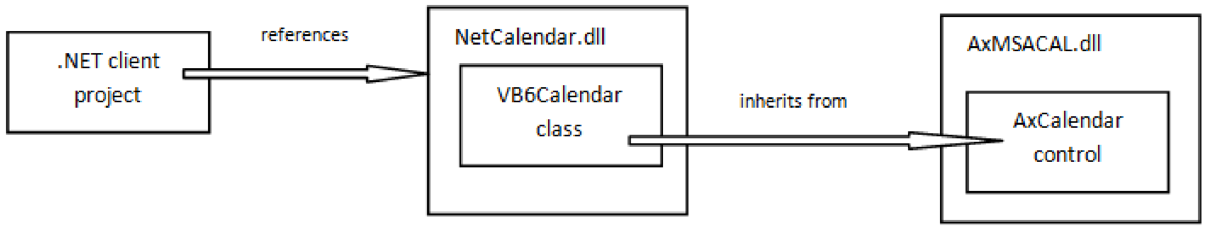
AxWrapperGen is also aware that VB6 members that have to do with position and size can be affected by the current value of the ScaleMode property of the parent form; the wrapper property or method for these elements calls methods in VB Migration Partner's support library:

```
Public Shadows Property Left() As Single
    Get
        Return VB6Utils.FromPixelX(Me, MyBase.Left, False)
    End Get
    Set(ByVal Value As Single)
        MyBase.Left = VB6Utils.ToPixelX(Me, Value, False)
    End Set
End Property
```

In many cases, the code that AxWrapperGen generates compiles correctly and without errors. For more complex controls – most notably, grids and charting controls – you may need to manually fix some statements to compile or run correctly. (For more details, see the remarks in the "Instructions" region, at the top of the generated wrapper class and read this whitepaper.)

When all fixes are in place, you can compile the project, generate the NetCalendar.dll assembly, and manually copy this DLL to VB Migration Partner's setup folder.

When VB Migration Partner runs, it scans all the DLLs in this folder, looks for classes that are marked with the **VB6Object** attribute, and takes note of the first argument of this attribute. From now on, VB Migration Partner knows that all MSACAL.Calendar ActiveX controls must be rendered as instances of the NetCalendar.VB6Calendar class, which in turn inherits from (and work as a wrapper for) the AxMSACAL.AxCalendar ActiveX control:



Please notice that you only need to run AxWrapperGen only once for each given ActiveX control used in your application. These wrapper classes are just VB.NET classes, therefore you can then group all wrappers in a single DLL, or you can compile them as part of the main VB.NET application if you don't want to distribute any additional DLL.

VB Migration Partner's approach to replacing ActiveX control might sound a bit contorted when described in abstract terms, but in practice it works like a charm, as this feedback can attest:

One of the major obstacles we found during the migration was the presence of Infragistics' DataGrid 3.1 in virtually all forms. Code Architects's support team helped us in authoring a .NET custom control (based on DevExpress' XtraGrid) that implements all the properties, methods, and events that were used by ProdWare (among the many exposed by DataGrid) and that faithfully reproduces their behavior so that the resulting .NET code can be considered as functionally equivalent to the original VB6 code. By applying the right attribute to our control, VB Migration Partner was able to automatically replace all occurrences of the Infragistics' grid with our control.

Dr. Italo Lunati - Elabora srl, makers of ProdWare

Here are the words from another user who fought (and won) against a large number of ActiveX controls:

In 1995 we realized a client/server application in VB3 using several third party controls, such as Crystal Report, Formula 1, and Crescent QuickPak. Over the years the software was greatly enhanced with new features and was ported to VB5 and then to VB6. [....] the included AxWrapperGen tool allowed us to create wrapper classes that guarantee the correct working of a few VB6 controls that aren't directly supported under VB.NET. We estimate that this tool alone spared us several man months of hard work.

Marco Meneo – ProgeSoftware

ActiveX control replacement

At this point VB Migration Partner knows how to migrate the ActiveX control, yet the result of the migration is a .NET application that still depends on the original ActiveX control. In fact, the original ActiveX control must be installed with the proper design-time or runtime license for migrated apps to work correctly.

Let's see now how to instruct VB Migration Partner to generate a .NET application that has no dependency on legacy ActiveX components. The wrapper class that we generated in the previous step greatly reduces the effort that is needed to complete the replacement step. In fact, the wrapper class works as a mediator between the .NET client project and the AxCalendar class (the actual ActiveX control), as shown in previous diagram. We can therefore modify the wrapper class without touching the .NET client project.

The first thing to do is replacing the VB6Calendar's base class so that VB6Calendar inherits from a .NET control that we selected. In our example, we can make VB6Calendar inherit from System.Windows.Forms.MonthCalendar control, thus we need to change the Inherits statement as follows:

```
<VB6Object("MSACAL.Calendar")> _
Public Class VB6Calendar
    Inherits System.Windows.Forms.MonthCalendar
    Implements IVB6Control
    ...
End Class
```

Also notice that we have dropped the DependsOnAssemblies property of the VB6Object, because our wrapper will have no dependency on any ActiveX control or DLL.

The change in the Inherits statement causes several compilation errors, because the MyBase object now points to a MonthCalendar control instead of AxCalendar, and these two controls have a different programming interface.

For example, the MonthCalendar control doesn't expose a property named FirstDay (the weekday shown in leftmost column), even though its FirstDayOfWeek property performs the same job. The FirstDayOfWeek .NET property takes a System.Windows.Forms.Day enumerated value, whereas the original FirstDay VB6 property takes a 16-bit integer, therefore we need to perform a type conversion (edits are in boldface):

```
Public Property FirstDay() As Short
    Get
        Return CInt(MyBase.FirstDayOfWeek)
    End Get
    Set(ByVal value As Short)
        MyBase.FirstDayOfWeek = CType(value, System.Windows.Forms.Day)
    End Set
End Property
```

If the ActiveX control and the .NET control are similar enough, all necessary code changes are as simple as the one we have just seen.

Another important detail: you don't strictly need to edit each and every member of the wrapper class to account for the different base class. For example, if your client .NET project doesn't use the NextDay method, you can just remark out its inner statement(s):

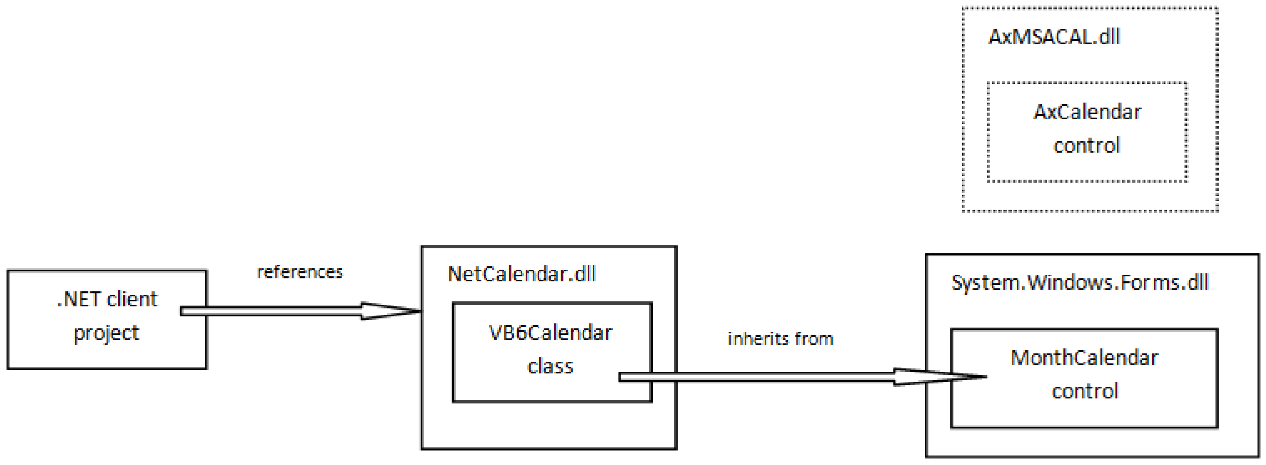
```
Public Sub NextDay()
    ' MyBase.NextDay()
End Sub
```

Better yet, you can add a Throw statement and an Obsolete attribute:

```
<Obsolete("MSCalendar.NextDay method isn't supported")> _
Public Sub NextDay()
    Throw New System.NotImplementedException()
End Sub
```

The Obsolete attribute forces Visual Studio to treat all calls to unsupported members as compilation warnings, whereas the Throw statement ensures that all calls to these members cause a runtime exception. These two elements ensure that a call to NextDay from the .NET client application won't go unnoticed.

You can now remove the project references to AxMSACAL.dll and MSACAL.dll files, so that the NetCalendar.dll assembly has no dependency on ActiveX legacy components and is ready to be compiled for 64-bit platform. The dependency diagram becomes:



For all practical purposes, all migrated .NET forms are now using the MonthCalendar control, via the VB6Calendar class. In case you haven't noticed it, let's emphasize that ***we can replace the ActiveX control without changing any single statement in the migrated .NET project!***

The wrapper class approach has other advantages, some of which might not be obvious:

- The wrapper class is fully independent of the main application, therefore ***ActiveX control replacement can be carried out by a different team of developers***, in parallel with the developers that are working on the main application.
- The number of edits on the wrapper class that are necessary to complete ActiveX control replacement depends only on the number and complexity of the members your main app actually uses; it doesn't depend on how many times the ActiveX control is used in the original VB6 project. In real-world projects of medium or large size, ***this factor dramatically reduces the overall necessary effort***.
- The developers working on the wrapper class need to know nothing about the main application nor do they need to see its source code, therefore ***you can safely outsource this job to another company*** if you don't have enough in-house resources, regardless of how strict your security policies are. (Code Architects offers this kind of service, by the way.)
- Being standard VB.NET classes, you can easily include these modified wrapper classes in your main project, if you don't want to distribute them as additional DLLs. This arrangement can sometime give you the additional flexibility you might need to work around special cases, undocumented behaviors, and so forth.

For more information about AxWrapper and how to create and refine wrapper classes, please read the ActiveX Controls and wrapper classes whitepaper.

How other VB6 conversion tools deal with ActiveX controls

A smart approach to 3rd-party ActiveX control conversion

Unlike VB Migration Partner and its wrapper classes, all other VB6 conversion tools on the market attempt to convert ActiveX controls using a code transformation technique known as *member mapping*.

Code transformations are used to solve some of the differences between an ActiveX control and its .NET equivalent control. This approach works well when resolving an ActiveX member if .NET offers a property, method or event with *exactly* the same behavior, but often brings to clumsy results in other cases. For example, consider this simple handler for the KeyPress event:

```
' VB6
Private Sub Text1_KeyPress(KeyAscii As Short)
    ' convert the pressed key to uppercase, but ignore spaces
    If KeyAscii = 32 Then KeyAscii = 0: Exit Sub
    KeyAscii = Asc(Chr(KeyAscii))
End Sub
```

A VB6 conversion tool from one of our competitors generates this VB.NET code:

```
Private Sub Text1_KeyPress(ByVal sender As Object, _
       ByVal e As KeyPressEventArgs) Handles Text1.KeyPress
    Dim KeyAscii As Integer = Asc(e.KeyChar)
    ' convert the pressed key to uppercase, but ignore spaces
    If KeyAscii = 32 Then
        KeyAscii = 0
        If KeyAscii = 0 Then
            e.Handled = True
        End If
        Exit Sub
    End If

    KeyAscii = Asc(Chr(KeyAscii).ToString()(0))
    If KeyAscii = 0 Then
        e.Handled = True
    End If
    e.KeyChar = Chr(KeyAscii)
End Sub
```

which is obviously more difficult to read and maintain than the original VB6 code.

In addition to delivering more verbose and obscure code, ActiveX conversion based solely on code transformation don't work well with VB6 members that have a significantly different behavior in .NET.

Comparison of ActiveX replacement techniques

The following table compares VB Migration Partner's wrapper classes with the "traditional" approach based on code transformation and member mapping adopted by all other VB6 conversion tools.

difference between VB6 and .NET	**wrapper classes**	**code transformation**
member name	supported	supported
member syntax	supported	supported
member behavior	supported – wrapper classes give you the ability to change the behavior of the .NET control so that it perfectly mimics the original ActiveX control. (*)	supported only if the difference can be solved by means of basic code transformations.
control appearance	supported	not supported
event order	supported - by editing the code in the wrapper class you can fire events in any order as well as fire events that the .NET control wouldn't normally fire. (**)	not supported
missing property, method or event	supported	supported only in some cases, typically by means of helper methods in a separate support library.
missing features	supported - if necessary, you can add missing features by extending the wrapper class as you see fit. (*)	not supported
method call via late-binding	supported - .NET controls expose members with same name and syntax as the original ActiveX control	not supported
spit members (ie VB6 methods that correspond to 2+ .NET methods)	supported	can be supported only in very simple cases
combined members (ie multiple VB6 method that correspond to a single .NET method)	supported	can be supported only in very simple cases
databinding	supported	can be supported only in very simple cases

(*) It's easy for us to prove this claim. All controls in VB Migration Partner support library are actually wrapper classes of standard .NET controls in System.Windows.Forms namespace. Our controls support virtually all VB6 features – including graphics, drag-and-drop, DDE, double-buffering, user-defined coordinate systems. Any feature that we implemented in our standard support library can be implemented in user-defined wrapper classes, too. None of our competitors – who rely on code transformations exclusively – can support these features.