Support library and code maintainability

VB6 and VB.NET are similar languages that differ for a myriad of major and minor details. Even keywords, methods, and controls that have the same name in both environments may have a completely different behavior. The problem is even bigger when converting from VB6 to C#.

A tool that aims at preserving functional equivalence with the original VB6 code should account for the majority of these differences. All the VB6 conversion products on the market, including VB Migration Partner, fill this gap with a combination of these two elements:

- code transformation techniques, to generate VB.NET or C# code that behaves like the original VB6 code
- support library that expose methods and controls that aren't found in the .NET Framework

One of VB Migration Partner's strengths is its comprehensive support library. Its dozens of classes and hundreds of methods ensure that the generated VB.NET or C# code performs like the original VB6 app, an important factor to dramatically reduce the time and cost of the migration process. No other conversion tool comes with such a complete library and in fact no other conversion tool can compete with VB Migration Partner in its support for VB6 features.

Our competitors are aware of the many advantages of this approach, therefore some of them tend to downgrade the importance of support library in migration scenarios.

The purpose of this document is to confute these and other “facts” that can be found in marketing literature. In our quest for objectivity, we will back up our affirmation with concrete code samples or other technical documentation.

All VB6 conversion software use a support library

For example, the product from one of ***our competitors uses a support library that consists of over 30,000 lines***, gathered in helper classes that perform a variety of tasks that are essential for the migrated .NET code to work correctly. If your VB6 code references any property or method that isn’t directly supported by a .NET control (or that behaves in a different manner), the generated code references one of these helper classes and therefore depends on them.

The truth is, **all migration tools are based on a support library**, the main difference being that VB Migration Partner’s library is more powerful and provides support for more VB6 features, including DDE, drag-and-drop, graphic methods, etc. All our competitors, including those who have been in this market for longer than us, fail to support these and many other language features.

VB Migration Partner translates VB6 controls into references to .NET controls defined in the support library. For example, the TextBox control maps to the VB6TextBox class, the TreeView control maps to the VB6TreeView class, etc. Thanks to these control classes, ***VB Migration Partner can preserve 100% functional equivalence*** with the original VB6 program, ***with few or no manual fixes to the generated code***.

This important feature dramatically reduces the duration and cost of even the most complex migration project. Just as important, ***developers working on the migration don't need to be familiar with the intricacies of .NET Framework***, because the support library takes care of all the differences between the VB6 and the .NET worlds. This detail means that even VB6 developers can be immediately productive with .NET, with no need for in-depth .NET Framework training.

Only a support library guarantees full functional equivalence

For the most part, classes in VB Migration Partner's library are thin wrappers for native .NET controls that ensure that each property, method, and event behaves exactly as in VB6. For example, let's consider the following VB6 code:

```
' move the cursor at the top of the txtEditor TextBox control
txtEditor.SelStart = 0
```

A "traditional" VB6 converter – such as Upgrade Wizard (formerly included in Visual Studio) or similar products – converts the above code into:

```
' move the cursor at the top of the txtEditor TextBox control
txtEditor.SelectionStart = 0
```

No warning is emitted, thus a developer working at the migration is led into thinking that the result is fully equivalent to the original code, which is not the case. However, assignments to the VB6 SelStart property have two side effects that the .NET SelectionStart property doesn't have: they clear the selection and scroll the field contents to ensure that the text cursor is in the visible area of the control. This behavior might or might not affect the way your application works, but in the former case it might take you hours to realize why the migrated code isn't working as expected.

The SelStart property of the VB6TextBox class in VB Migration Partner's library reproduces both side effects, so that the developer doesn't have to worry about them. Here is the source code of the relevant portion of the library:

```
Public Property SelStart() As Integer
    Get
        Return MyBase.SelectionStart
    End Get
    Set(ByVal value As Integer)
        MyBase.SelectionStart = value
        ' in VB6 setting SelStart brings the cursor into the visible window
        ' and resets the selection length
        MyBase.ScrollToCaret()
        MyBase.SelectionLength = 0
    End Set
End Property
```

In addition to perfectly mimicking VB6 behavior, defining a property named SelStart (as in VB6) ensures that the code works correctly even if the TextBox control is accessed via late-binding. For example, only VB Migration Partner correctly deals with the following VB6 code:

```
' reset cursor position in TextBox, ComboBox, and RichTextBox controls
Sub ResetCursor(ctrl As Object)
    ctrl.SelStart = 0
End Sub
```

All properties and methods in VB Migration Partner are conceptually similar to SelStart, and account for major and minor differences between VB6 and the .NET Framework. For example, the code behind Left, Top, Width, and Height properties ensures that they work correctly with all possible ScaleMode settings. Here's a code snippet behind the VB6TextBox and other controls in the 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
```

(VB6Utils is a helper class defined elsewhere in the library). Thanks to its support library, VB Migration Partner is the only VB6 conversion software that works correctly with any VB6 coordinate system, including user-defined ScaleModes.

False myths about support libraries

Some developers would prefer not to distribute an additional DLL with their migrated apps. These concerns are understandable, yet we believe that they are negligible if compared to the many advantages that a well-written, comprehensive library gives you.

Such benefits stand out so clearly that even our competitors don't deny that an extensive library can speed up a migration project. To counter this evidence, however, they may insist that a library has many shortcomings. We can easily prove that all these "issues" are groundless, as we do in our 17 reasons for using a support library in migration scenarios whitepaper.

The point is, a comprehensive support library is the key factor in achieving 100% functional equivalence ***and*** in keeping migration time and cost as low as possible. See what one of our customers has to say:

"An initial migration compared migration tools from six vendors. It showed superior results for VB Migration Partner, which delivered fewer compilation and runtime errors than all its competitors… ***It took 2.5 hours to***

get a compilable and runnable VB.NET project with VB Migration Partner, and 13 hours with its closest competitor."

The higher productivity that a complete support library gives you is a sure fact, and no competitor has yet published different results in similar rigorous productivity tests. In the absence of objective measurements, however, they can attempt to scare potential customers away from VB Migration Partner by making vague statements about its supposed limitations.

The following list summarizes the kind of claims we found in our competitors' documentation. It isn't clear which product they had in mind when they wrote these sentences, however we can easily prove that all of them are false when applied to VB Migration Partner.

a. *…other vendors charge a runtime fee for their support library.*

   FALSE! VB Migration Partner users can freely distribute its library with their apps.

b. *…other vendors may charge a subscription for using the library in the future.*

   FALSE! VB Migration Partner's EULA states that users will be allowed to download any future version of the library at no additional cost.

c. *…other vendors don't make their library's source code available to customers.*

   FALSE! VB Migration Partner users can license the library's code.

d. *… apps that rely on a support library might not work on future versions of the operating system.*

   FALSE! VB Migration Partner's library is written in pure VB.NET and uses only documented features of the .NET Framework. As such, the library is guaranteed to work on all future Windows versions that support .NET Framework 3.5 and 4.x binaries.

   A closer examination reveals that this line of reasoning is illogical. Just for the sake of discussion, let's assume that Microsoft releases a Windows or .NET Framework version breaks compatibility with existing .NET 4.x programs. In such a highly unlikely case, your own .NET applications will stop working correctly and the support library will be the least of your problems.

   A support library actually helps to preserve compatibility. Should a future version of Windows have minor problems in running .NET 4.x apps, Code Architects will promptly fix the problem release a new support library that all users can download free of charge. You might need to solve compatibility problems of your main app, but at least you don't have to worry about the support library.

e. *… if your migrated apps depend on a 3rd-party library, you might be in trouble if the vendor decides to stop supporting it in the future*

FALSE! VB Migration Partner's EULA states that – should Code Architects stop supporting the product – all customers will receive the library's source code at no additional cost. In this respect, choosing Code Architects is as safe as, or safer than, choosing any 3rd-party control such as a grid or a charting component.

f. *… a support library can slow down your code*

FALSE! VB Migration Partner's library has been authored by a team of expert developers, including two Microsoft Regional Directors who wrote 7 top-selling Microsoft Press books on VB and .NET programming, authored over one hundred articles on technical magazines routinely give lectures in US and Europe, and consult for Microsoft and its largest customers.

VB Migration Partner's support library is carefully optimized and often runs much faster than the code that most VB.NET or C# developers usually write, especially if they are put under pressure by near deadlines. For example, our VB6Collection class runs many times faster than the standard VB.NET Collection object, and our StringBuilder6 object allows you to automatically speed up string concatenations by a factor of 100x without having you modify the generated .NET code.

g. *…other migration tools generate inefficient code that retains its VB6 flavor*

FALSE! VB Migration Partner's conversion engine uses very sophisticated refactoring techniques and generates code that takes full advantage of VB.NET and C# features, including Try/Catch blocks, short-circuiting (the AndAlso or && operators), the IDisposable interface, variable initializers, compound operators (e.g. x += 1 or x++), and much more.

Interestingly, VB Migration Partner can apply many refactoring techniques that no other VB6 conversion tool currently supports, e.g. Gosub refactoring, Declare overloads, using StringBuilder to speed up string concatenations, plus a number of C#-only features such as using dynamic variables for late-bound method calls, using of "out" parameters when possible, optionally generation of overloads for ref/out parameter to simplify method calls.

Speaking of refactoring, VB Migration Partner can optionally rename all members to comply with .NET naming rules. If you don't like our rules, you can customize the rename engine by modifying an XML file. No other conversion software supports customizable rename rules.

h. *…projects produced by other VB6 converters are difficult to evolve*

FALSE! VB Migration Partner library doesn't limit you in any way, because all its classes inherit from native .NET classes and you can therefore leverage the .NET Framework full potential.

i. *…you can't mix .NET native forms and controls with forms and controls defined in a support library*

FALSE! After you convert a VB6 project to.NET using VB Migration Partner, you can extend the application by adding .NET native forms, and you can drop .NET native controls onto a migrated form.

j. *…the code produced by other VB6 converters that adopt a support library is hard to maintain*

FALSE! A support library can make your migrated code *more* readable and *easier to maintain*, as demonstrated in the next section.

Readability and maintainability of generated code

Proving that an extensive support library can generate more readable and maintainable code is easy. We just need to compare the code generated by VB Migration Partner with the code coming out from a "traditional" VB6 conversion tool that has a less extensive support library. Consider the following KeyPress handler written in VB6:

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

VB Migration Partner converts it as follows:

```
' VB.NET
Private Sub Text1_KeyPress(ByRef KeyAscii As Short) Handles Text1.KeyPress
    ' convert the pressed key to uppercase, but ignore spaces
    If KeyAscii = 32 Then KeyAscii = 0 : Exit Sub
    KeyAscii = Asc(Chr(KeyAscii))
End Sub

// C#
private void Text1_KeyPress(ref short KeyAscii)
{
   // convert the pressed key to uppercase, but ignore spaces
   if (KeyAscii == 32 )
   {
     KeyAscii = 0;
     return;
   }
   KeyAscii = VB6Helpers.Asc(VB6Helpers.Chr(KeyAscii));
}
```

The resulting .NET code is basically identical to the original code, and many VB6 developers can understand how to read and maintain this code. Obviously there is no maintainability problem here.

Let's see now the code produced by another conversion tool that doesn't rely on an extensive support library. The name of the product we used to generate the following lines doesn't really matter, because the same reasoning applies to any conversion tool that attempts to fill the gap between VB6 and VB.NET exclusively by means of code transformation techniques:

```
' VB.NET
```

```
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

    'UPGRADE_ISSUE: (1058) Assignment not supported: KeyAscii to a non-positive constant
    'More Information: http://www.vbtonet.com/ewis/ewi1058.aspx
    KeyAscii = Strings.Asc(Strings.Chr(KeyAscii).ToString()(0))
    If KeyAscii = 0 Then
        e.Handled = True
    End If
    e.KeyChar = Chr(KeyAscii)
End Sub

 // C#
private void Text1_KeyPress(Object eventSender, KeyPressEventArgs eventArgs)
{
    int KeyAscii = Strings.Asc(eventArgs.KeyChar);
    // convert the pressed key to uppercase, but ignore spaces
    if (KeyAscii == 32)
    {
        KeyAscii = 0;
        if (KeyAscii == 0)
        {
            eventArgs.Handled = true;
        }
        return;
    }
    //UPGRADE_ISSUE: (1058) Assignment not supported: KeyAscii to a non-positive c
    //More Information: http://www.vbtonet.com/ewis/ewi1058.aspx
    KeyAscii = Strings.Asc(Strings.Chr(KeyAscii).ToString()[0]);
    if (KeyAscii == 0)
    {
        eventArgs.Handled = true;
    }
    eventArgs.KeyChar = Convert.ToChar(KeyAscii);
 }
```

In the attempt to preserve functional equivalence of the original 3 statements inside the method, the tool generated as many as 13 (thirteen!) statements, a 4x increase in size.

You might believe that the KeyPress event is a special and unique case, so let's see another code snippet, a simple VB6 method that defines two optional parameters:

```
Public Sub TestOptional(Optional x As Variant, Optional y As Variant)
    If IsMissing(x) Then
        If IsMissing(y) Then
            x = 10
            y = 20
        End If
    End If
    MsgBox(x * y)
End Sub
```

This is how VB Migration Partner correctly translates it to .NET:

```
'VB.NET
Public Sub TestOptional(ByRef Optional x As Object = MissingValue6, _
  ByRef Optional y As Object = MissingValue6)
    If IsMissing6(x) AndAlso IsMissing6(y) Then
        x = 10
        y = 20
    End If
    MsgBox6(x * y)
End Sub

// C#
public void TestOptional(ref object x, ref object y)
{
    if ( VB6Helpers.IsMissing(x) && VB6Helpers.IsMissing(y) )
    {
        x = 10;
        y = 20;
    }
    VB6Helpers.MsgBox(VB6Helpers.CStr(VB6Helpers.CDbl(x) * VB6Helpers.CDbl(y)));
}
```

Thanks to its support library, VB Migration Partner can generate code that is as readable and maintainable as the original VB6 method, even after conversion to C#. Well, the generated code is actually ***more readable***, because our software merged the two nested IFs, however this improvement is achieved by means of its sophisticated conversion engine and isn't a consequence of using a library.

Let's now look at the code that another tool generates for the same method:

```
Public Sub TestOptional(ByRef opt_x As Object, ByRef opt_y As Object)
    Dim y As Object = Nothing
    If opt_y Is Nothing Or Not opt_y.Equals(Type.Missing) Then _
       y = TryCast(opt_y, Object)
    Dim x As Object = Nothing
    If opt_x Is Nothing Or Not opt_x.Equals(Type.Missing) Then _
       x = TryCast(opt_x, Object)
    Try
       If Not (opt_x Is Nothing) AndAlso opt_x.Equals(Type.Missing) Then
          If Not (opt_y Is Nothing) AndAlso opt_y.Equals(Type.Missing) Then
             y = 20
```

```
            End If
        End If
        MessageBox.Show(CStr(CDbl(x) * CDbl(y)), Application.ProductName)
    Finally
        opt_y = y
        opt_x = x
    End Try
End Sub

Public Sub TestOptional(ByRef opt_x As Object)
    Dim tempRefParam As Object = Type.Missing
    TestOptional(opt_x, tempRefParam)
End Sub

Public Sub TestOptional()
    Dim tempRefParam2 As Object = Type.Missing
    Dim tempRefParam3 As Object = Type.Missing
    TestOptional(tempRefParam2, tempRefParam3)
End Sub
```

Notice that this code introduced two new methods and two new variables inside the main method, yet it doesn't contain a single remark that explains why this extra code was generated. It should be noted that this extra code doesn't guarantee functional equivalence with the original code. In fact, this approach generates a compilation error in any call that specifies only the second argument, as in:

```
TestOptional(, 1)
```

Now ask yourself: ***Which code would you like to maintain in the future?*** The concise and readable 7-line method produced by VB Migration Partner or the 3 methods and 28 statements produced by the other tool?

We might add many more examples of how **a *support library can make your code more concise, faster, easier to read, and easier to maintain in the future***, but we don't think we really need to: just pay a visit to our code sample section and see the code that VB Migration Partner generates for real-world VB6 apps.