Migrating from VB6 to C# with VB Migration Partner

Since its public launch, in May 2008, Code Architects' VB Migration Partner has proven to be the most complete VB6-to-VB.NET code converter on the market. The power of its code generation engine, which can be easily controller by over 80 different migration pragmas, and the completeness of the companion support library were the winning factors in this market segment.

VB Migration Partner version 1.50 is even more powerful than before, thanks to the support for VB6-to-C# code generation. This document outlines the many differences between Visual Basic (both classic VB6 and VB.NET) and C#, and how these differences may impact the migration process.

VB Migration Partner has no problem to convert virtually any VB6 keyword and feature to VB.NET, with just a handful exceptions such as the VarPtr, StrPtr, ObjPtr undocumented methods and a few others. This is possible also because the VB6 and VB.NET are more similar to each other than you might think at first. For example, both Visual Basic languages support error handling and late binding in the same way, a detail that greatly simplifies the job of VB Migration Partner.

When converting from VB6 to C#, on the other hand, VB Migration Partner has to account for many major and minor details that can make a ***big*** difference in some cases. This document explains what these differences are and how VB Migration Partner can fill the gap between VB6 and C# while preserving functional equivalent (and illustrates the few cases when obtaining functional equivalence isn't possible without some manual labor).

How to convert from VB6 to C#

There are two ways to indicate C# as the target language of a migration:

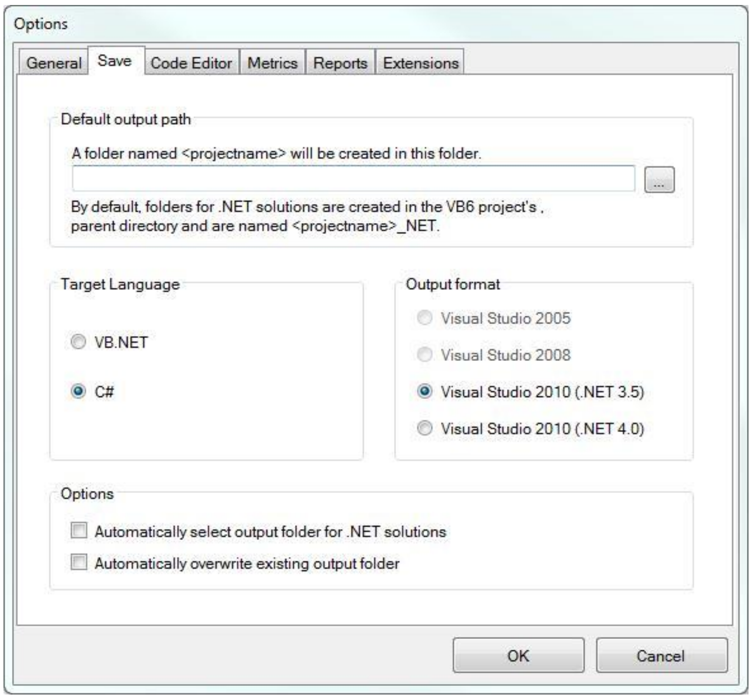
- You can use the radio buttons in the Save tab of the Tools-Options dialog box to select the target language and the target Visual Studio version
- You can use the **SetLanguage** pragma, which takes both the language name and the VS version:

```
' REM convert to C# for VS2010 and .NET 3.5

'## SetLanguage C#, 2010
```

```
' REM convert to C# for VS2010 and .NET 4.0

'## SetLanguage C#, 2010_40
```

When using the VBMP.EXE command-line version of VB Migration Partner, you can use the new **/language** option:

```
    VBMP myproject.vbp /language:c# /version:2010_40
```



Default output path

A folder named <projectname> will be created in this folder.

By default, folders for .NET solutions are created in the VB6 project's , parent directory and are named <projectname>_NET.

Target Language

- ( ) VB.NET
- (●) C#

Output format

- ( ) Visual Studio 2005
- ( ) Visual Studio 2008
- (●) Visual Studio 2010 (.NET 3.5)
- ( ) Visual Studio 2010 (.NET 4.0)

Options

- [ ] Automatically select output folder for .NET solutions
- [ ] Automatically overwrite existing output folder

C# doesn't support modules

All VB6 modules are converted into C# static classes; all calls to methods defined in a different module use the module's name as a prefix.

```
 ' this method is located in Functions.bas (a VB6 module)

Public Sub DoSomething()

    ' ...

End Sub


' the DoSomething method is invoked from inside a form

Private Sub Form_Load()

    DoSomething()

End Sub
```

Here's the C# translation of previous code:

```
public static class Functions

{

   public static void DoSomething()

   {

      // ...

   }

}


// the DoSomething method is invoked from inside a form
```

```
private void Form_Load()

{

    Functions.DoSomething();

}
```

C# uses "VB6Helpers" prefix to invoke methods in support library

A direct consequence of the fact that C# doesn't support modules is that all calls to methods defined in CodeArchitects.VBLibrary.dll require to be prefixed by the class name. To make the generated code appear more consistent, all the support methods in VBLibrary – including all methods with a "6" suffix such as Len6 or Abs6 – have been duplicated in a static class named **VB6Helpers**, as this code demonstrates:

```
' VB6

Public Sub DoSomething(ByVal i As Integer, ByRef s As String)

    s = Left(s, String(" ", i))

End Sub



// C#

public void DoSomething(short i, ref string s)

{
    s = VB6Helpers.Left(s, VB6Helpers.String(" ", i));
}
```

C# doesn't support static local variables

VB6 allows defining a static local variable, that is a variable that preserves its value between consecutive calls to the method where they are defined. Additionally, if the method itself is defined with the Static keyword, then all its local variables are implicitly static.

VB.NET supports static variables (but not the Static keyword applied to methods), but C# doesn't. For this reason, all static variables are converted into class-level variables. If there is no other class-level variable with same name, then the variable preserves its name; else, its name will be obtained by prefixing the variable name with the method name, as this code demonstrates:

```
' VB6
Private UserName As String     ' class-level field

Public Sub DoSomething()
  Static Password As String      ' static local variable
  Static UserName As String      ' variable with name collision
  Debug.Print UserName & " " & Password
End Sub

// C#
private string UserName;             // class-level field
private string Password;             // static local variable

private string DoSomething_UserName; // variable with name collision


public void DoSomething()

{

   VB6Helpers.DebugPrintLine(Password + " " + DoSomething_UserName);

}
```

C# doesn't create an implicit local variable named after the current method or property

Under VB6 you specify the value to be returned by a Function or a Property Get procedure by assigning a variable named after the function or the property. This technique is also supported by VB.NET, which additionally lets you specify the returned value by means of the Return keyword.

Under C# the **return** keyword is the only way you have to return a value from a function or a property get block. For each Function and Property Get, VB Migration Partner defines a local variable named **_retValue**, and correctly uses it in a property return statement.

```
' VB6

Function Evaluate(ByVal x As Double) As Double

    If x < 0 Then

        Evaluate = -1
    End If
    ' …
    Evaluate = Evaluate * 2
End Function


// C#
public double Evaluate(double x)
{
    double _retValue = 0;
    if (x < 0)
    {
       _retValue = -1;
    }
    // …
    _retValue *= 2;
    return _retValue;
}
```

In very simple cases – for example when returning a constant value – the _retValue variable is omitted and the returned value appears right in a return statement, as in:

```
' VB6
```

```
Function Evaluate2(ByVal x As Double) As Double

   If x < 0 Then

      Evaluate2 = 1 / x

   Else

      Evaluate2 = x * 2

   End If

End Function



// C#

public double Evaluate2(double x)

{

   if (x < 0)

   {

      return 1 / x;

   }

   else

   {

      return x * 2;

   }

}
```

In addition to what we've seen so far, the C# code generator has to account for another requirement: non-void methods and the get block of properties must explicitly return a value. This differs from VB6 and VB.NET, which implicitly return the default value for the method or property data type (zero, an empty string, or null).

You don't have to worry about this detail, because VB Migration Partner always generates an explicit return statement in all non-void methods. However, in some cases you might want to revise the generated C# code and possibly simplify it. (You can perform this kind of manual optimization at the end of the migration project.)

C# doesn't support implicitly-declared local variable (Option Explicit Off)

Unless the **Option Explicit** directive is defined, in VB6 you don't necessarily have to declare all variables: if a variable is referenced by not declared, it is assumed to be a Variant local variable. (More precisely, the type of undeclared variables is affected by directives such as DefInt or DefDbl). This is true also for VB.NET, but not for C#.

When converting to VB.NET, you can use the DeclareImplicitVariables pragma to force the declaration of all variables. Because C# doesn't support undeclared variables, this pragma is always assumed and variables are always declared when converting to C#. However, VB Migration Partner conveniently generates a migration warning for all implicitly-declared variables, so that you can check whether their type was assumed correctly.

C# requires that all local variables be initialized before use

The VB.NET compiler generates a warning if a variable is used before being initialized, but this warning can be ignored in most cases. The C# compiler is much stricter in that respect, and all variables must be correctly initialized before use else a compilation error occurs. This is also true for numeric, Date, and UDT variables, which don't cause a warning under VB.NET.

```
' VB6

Public Sub DoSomething(ByVal k As Integer)

   Dim i As Integer

   Dim s As String



   i = k

   '...

End Sub



// C#

public void DoSomething(short k)

{

   short i = k;

   string s = "";

}
```

C# can use the "out" keyword to define a parameter passed by reference

Both VB6 and VB.NET only recognize two mechanisms for passing an argument to a method parameter: by-value (the ByVal keyword) or by-reference (the ByRef keyword). Conversely, the C# language supports three value-passing mechanisms: by-value (no keyword), by-reference in/out (the **ref** keyword) and by-reference out-only (the **out** keyword).

For example, a method whose only purpose is to retrieve values from a database can (and should) use the **out** keyword for its parameters. In general, C# developers should use the **out** keyword whenever possible, because it lets the C# compiler optimize the generated code. Also, a variable that is passed to an **out** parameter doesn't need to be initialized before calling the method, which means that the compiler generates better code both inside the method and whenever the method is invoked.

By default, VB Migration Partner converts by-reference VB6 parameters into **ref** C# parameters, but you have the option to generate **out** parameters when possible, by means of the CSharpOption UseOut pragma. In this case, VB Migration Partner will use the **out** keyword where it's possible to do so, as this example demonstrates:

```
' VB6

'## CSharpOption UseOut

Public Sub DoSomething(ByRef x As Long, ByRef y As Long)

     x = x * 2    ' the incoming value of x is used

     y = x * 3    ' the value of y isn't used before assigning it

End Sub


// C#

public void DoSomething(ref int x, out int y)

{
    x *= 2;      // the x variable was rendered as "ref"

    y = x * 3;   // the y variable was rendered as "out"

}
```

Like all pragmas, the CSharpOptionUseOut pragma can be applied at the project, file, and method level.

As of this writing, no other VB6-to-C# conversion software supports the ability to generate "out" parameters.

The "ref" keyword can't be used with constants or expressions

This is an apparently minor detail that can severely affect the readability and maintainability of generated C# code.

First and foremost, VB Migration Partner always attempts to generate methods without **ref** parameters: if a parameter is never assigned inside a method, then it is rendered as a by-value parameter. This behavior has to be specifically activated when converting to VB.NET by means of the UseByVal pragma, but it is applied by default when converting to C#, on the assumption that a large number of by-reference parameters forces VB Migration Partner to generate ugly and inefficient code.

The main problem with by-reference parameters is that C# only permits to pass variables to them: if you pass a constant, a property, or an expression to such a parameter you get a compilation error under C#. Only class fields, local variables, and parameters can be passed to a C# **ref** parameter. Consider the following code:

```
' VB6

'## CSharpOption UseOut

Private Sub DoSomething(ByRef x As Long, ByRef y As Long, ByRef z As Long)
```

```
    y = x * x
    z = x * y
End Sub

Private Sub Test(ByRef k As Long)
    DoSomething 1, 2, k
End Sub

// C#
private void DoSomething(int x, ref int y, out int z)
{
    y = x * z;
    z = x * y;
}
private void Test(out int k)
{
    int _tempVar1 = 2;
    DoSomething(1, ref _tempVar1, out k)
}
```

Notice that the **x** parameter is never assigned inside the DoSomething method, therefore VB Migration Partner can safely convert it into a by-value parameter. Moreover, being a by-value parameter, it is OK to pass a constant to it, as it happens inside the Test method.

Conversely, the **y** parameter is assigned inside the method and must be rendered with the **ref** keyword. Being a by-reference parameter, it isn't legal to pass the "2" constant to it, as it happens inside the Test method. For this reason, VB Migration Partner has to generate a temporary variable whose only purpose is being able to use the **ref** keyword when invoking the method.

Finally, the **z** parameter is assigned inside the method and is never used before the assignment, therefore the CSharpOption UseOut pragma causes VB Migration Partner to use the **out** keyword for it. Notice that this keyword is also used inside the Test method and indirectly causes the **k** parameter to be defined with the **out** keyword as well.

The problem with this C# requirement is that sometimes forces VB Migration Partner to generate a lot of temporary variables when invoking a method with many by-reference parameters. Even worse, in some cases it is impossible to correctly generate such temporary variables, for example when the code invokes a Function inside an If, ElseIf, or Do statement. Consider the following code:

```
' VB6

Function Evaluate(x As Long, y As Long, z As Long) As Long

    x = x + 1

    y = y * x

    z = z * x * y

    Evaluate = x + y + z

End Function



Sub Test()

    Dim i As Long

    Do

        Debug.Print i

    Loop While Evaluate(i, 2, 4) < 100

End Sub
```

```
// C#
public int Evaluate(ref int x, ref int y, ref int z)
{
    x++;
    y *= x;
    z = z * x * y;
    return (short)(x + y + z);
}

public void Test()
{
   int i = 0;

   do
   {
       VB6Project.DebugPrintLine(i);
       // UPGRADE_ISSUE (#12D8): Unable to insert this code before next statement:
       // int _tempVar1 = 2; int _tempVar2 = 4;
   }
   while ( Evaluate(ref i, ref _tempVar1, ref _tempVar2) < 100 );
}
```

In the above examples, there is no "right" place where VB Migration Partner can insert the declaration of the **_tempVar1** and **_tempVar2** temporary variables, therefore it just emits an UPGRADE_ISSUE message that specifies the missing declarations. You just can't ignore this message, because the absence of variable declarations causes a compilation error, thus you *****must***** insert the declaration in an appropriate place before you can run the converted C# code.

This situation is very common in generated C# code, which therefore tends to be ugly, bloated, difficult to maintain, hard to evolve, and slightly less efficient than it should. Moreover, when the declaration of temporary variables can't be performed automatically, it's a sheer waste of time.

To work around these issues, VB Migration Partner supports the powerful CSharpOption OverloadsForByval pragma, which can be applied at the project, file, and method level. As its name suggests, this pragma forces VB Migration Partner to generate all the necessary overloads for a given method that has one or more **ref** parameters. For example, when this pragma is applied against the previous VB6 code, the following C# code is generated:

```
// UPGRADE_INFO (#0731): This overload was generated to account for optional

//                       or ref/out parameters.

public int Evaluate(ref int x, int y, int z)

{

  return Evaluate(ref x, ref y, ref z);

}



public int Evaluate(ref int x, ref int y, ref int z)

{

     x++;

     y *= x;

     z = z * x * y;

     return (x + y + z);

}
```

```
public void Test()

{

      int i = 0;

      do

      {

            VB6Project.DebugPrintLine(i);

      }

      while ( Evaluate(ref i, 2, 4) < 100 );

}
```

As you see, thanks to the CSharpOption OverloadsForByval pragma, the generated C# code is more readable, looks like the code that a developer would write, and – more important – doesn't require any manual fix after the migration.

The UPGRADE_INFO warning clearly marks all the overloads that were generated by VB Migration Partner to avoid temporary variables when a given method is invoked. Notice that VB Migration Partner only generates the overloads that are strictly needed in current program, as opposed as to all the possible combinations of by-value and by-reference parameters.

No other VB6-to-C# conversion software supports the ability to generate overloads for methods that expose by-reference parameters used with by-value semantics.

C# doesn't support ref/out optional parameters

C# language supports optional parameters, with the following restriction: only a by-value parameter can be defined as optional. This detail makes the conversion from VB6 to C# more difficult if the code contains many methods with optional parameters, even though the problem is somewhat less serious than it might appear because VB Migration Partner automatically generates by-value parameter if the parameter is never assigned inside the method.

Consider the following VB6 code and the corresponding C# that VB Migration Partner generates:

```
' VB6
Sub DoSomething(Optional x As Long = 1, Optional y As Long = 2, Optional z As Long = 3)
    y = x * z
End Sub

Sub Test()
   Dim x1 As Long
   DoSomething x1
End Sub

// C#
public void DoSomething(int x, ref int y, int z = 3)
{
    y = x * z;
}

public void Test()
{
    int x1 = 0;
    int _tempVar1 = 2;
    DoSomething(x1, ref _tempVar1);
}
```

The **x** and **z** parameters in the DoSomething method were rendered using the by-value semantics, but the **y** parameter wasn't (because it is assigned inside the method). For these reason, even if all three parameters are defined as optional in VB6, only the **z** parameter was actually rendered as an optional parameter in C#. (The **y** parameter is non-optional because it is a **ref** parameter, whereas the **x** parameter is non-optional because an optional parameter can precede a non-optional parameter in the method signature.)

Under VB6, the Test method invokes the DoSomething method and omits the values for the **y** and **z** parameters. Under C#, however, the first two parameters are non-optional, therefore VB Migration Partner automatically generates a temporary variable to account for the fact that y isn't optional after the conversion.

These automatically-generated temporary variables can negatively affect the readability and overall quality of the generated C# code, as it happens for temporary variables that must be generated to account for ref/out non-optional parameters. You can avoid such defect by applying the CSharpOption OverloadsForOptional pragma, at the project, file, or method level. When such a pragma is applied to the above VB6 code, this is what VB Migration Partner generates:

```
public void DoSomething()

{

   int y = 2;

   DoSomething(1, ref y, 3);

}



public void DoSomething(int x)

{

   int y = 2;

   DoSomething(x, ref y, 3);

}
```

```
public void DoSomething(int x, ref int y)

{

   DoSomething(x, ref y, 3);

}



public static void DoSomething(int x, ref int y, int z)

{

   y = x * z;

}

public void Test()

{

   int x1 = 0;

   DoSomething(x1);

}
```

As you can see, when the CSharpOption OverloadsForOptional pragma is used, VB Migration Partner generates a number of overloads equal to the number of optional parameters in the original method, but removes all optional parameters from the method itself. Thanks to these overloads, the code in the call method (Test) doesn't require temporary variables.

Notice that – even when this pragma is used – there is still one case when a temporary variable is generated. This happens when the calling code omits an argument that is passed to a ref/out parameter, but passes a value to another parameter after it, as in the following VB6 code and the corresponding C# code:

```
' VB6

Public Sub Test()
```

```
    Dim x1 As Long

    DoSomething , , x1

End Sub



// C#

public void Test()

{

    int _tempVar1 = 2;

    DoSomething(1, ref _tempVar1, x1);

}
```

In cases such as this, you can avoid the temporary variables by combining the CSharpOption OverloadsForOptional with the CSharpOption OverloadsForByval. When these two pragmas are used together, VB Migration Partner generates an additional overload for the DoSomething method, which in turn allows simplifying the code generated inside the Test method:

```
// UPGRADE_INFO (#0731): This overload was generated to account for optional

//                       or ref/out parameters.

public void DoSomething(int x, int y, int z)

{

    DoSomething(x, ref y, z);

}



// ...
```

```
public void Test()

{

    int x1 = 0;

    DoSomething(1, 2, x1);

}
```

C# doesn't support Date constants

VB6 Date variables and functions are converted to DateTime variables when converting to C#. While the correspondence appears to be perfect, the fact that Date isn't a native C# data type has a number of important implications.

First and foremost, you can't define a DateTime constant in C#. In fact, when converting a VB6 Date constant to C#, VB Migration Partner generates a readonly DateTime variable, as this code demonstrates:

```
' VB6

Public Const LASTDAY As Date = #12/31/2012#



// C#

public readonly DateTime LASTDAY = VB6Helpers.DateConst("12/31/2012");
```

Notice that VB Migration Partner generates a reference to the **VB6Helpers.DateConst** method rather than using the constructor of the DateTime type, for you to quickly locate all converted Date constants. Optionally, you can convert these occurrences into DateTime constructors by adding the following PostProcess pragma:

```
'## project:PostProcess "VB6Helpers\.DateConst\(""(?<mm>\d+)/(?<dd>\d+)/(?<yy>\d+)""\)",
        "new DateTime(${yy}, ${mm}, ${dd})"
```

Second, the fact that you can't define a C# DateTime constant means that you can't specify a default value for an optional DateTime parameter. As a matter of fact, C# doesn't support DateTime optional parameters: when converting a method that has one or more optional Date parameters, VB Migration Partner generates a C# method whose corresponding parameter isn't optional, as this code demonstrates:

```
' VB

Public Sub DoSomething(Optional ByVal x As Long, Optional ByVal d As Date, _

    Optional ByVal z As Long)

  ' ...

End Sub



// C#

public void DoSomething(int x, DateTime d, int z = 0)

{

    // ...

}
```

Notice in previous code that the **d** parameter is rendered as non-optional, as is the **x** parameter (because optional parameter can precede non-optional ones). You can somewhat reduce the impact of this C# language limitation by means of the CSharpOption OverloadsForOptional pragma.

C# doesn't support passing a scalar variable to a "ref" Object parameter

Under VB6 it is possible to pass a scalar variable - that is, a numeric, string, or Date variable - to a ByRef Variant parameter and expect that the called method modifies the value of the variable. This behavior was replicated under VB.NET, but can't be replicated under C#: when passing an argument to a **ref** parameter, C# requires that the argument be a variable of same type as the argument.

For this reason, in these circumstances VB Migration Partner has to generate temporary Object variables before invoking the method, and has to reassign the value of these temporary variables back to the original scalar variables when the method exits, as the following code demonstrates:

```
' VB6

Public Function Evaluate(v1 As Variant, v2 As Variant) As Long

    v1 = v1 * 2

    v2 = v2 * 4

    Evaluate = v1 * v2

End Function



Sub Test(x As Long, y As Long)

    Evaluate x, y

End Sub



// C#

public int Evaluate(ref dynamic v1, ref dynamic v2)

{

    v1 = VB6Helpers.CInt(v1) * 2;

    v2 = VB6Helpers.CInt(v2) * 4;

    return (int)(VB6Helpers.CDbl(v1) * VB6Helpers.CDbl(v2));

}
```

```
public void Test(ref int x, ref int y)

{

    object _tempVar1 = x; object _tempVar2 = y;

    Evaluate(ref _tempVar1, ref _tempVar2);

    x = VB6Helpers.CInt(_tempVar1); y = VB6Helpers.CInt(_tempVar2);

}
```

If the method calls occurs inside an If, ElseIf, Do, or For statement, VB Migration Partner is unable to correctly insert the assignments before or after the method call. In such cases, an UPGRADE_ISSUE is generated and you are expected to manually fix the problem, as in this code:

```
'  VB6

Sub Test(x As Long, y As Long)

    If Evaluate(x, y) > 10 Then

       Debug.Print "greater"

       ElseIf Evaluate(x, y) = 0 Then

       Debug.Print "zero"

    End If

End Sub



// C#

public void Test(ref int x, ref int y)

{

    object _tempVar1 = x; object _tempVar2 = y;
    if ( Evaluate(ref _tempVar1, ref _tempVar2) > 10 )

    {

    // UPGRADE_ISSUE (#12E8): Unable to insert this code after previous statement:

    // x = VB6Helpers.CInt(_tempVar1); y = VB6Helpers.CInt(_tempVar2);

        VB6Project.DebugPrintLine("greater");

        // UPGRADE_ISSUE (#12D8): Unable to insert this code before next statement:

        //    dynamic _tempVar3 = x; dynamic _tempVar4 = y;

    }

    else if ( Evaluate(ref _tempVar3, ref _tempVar4) == 0 )

    {

    // UPGRADE_ISSUE (#12E8): Unable to insert this code after previous statement:

    // x = VB6Helpers.CInt(_tempVar3); y = VB6Helpers.CInt(_tempVar4);

        VB6Project.DebugPrintLine("zero");

    }

}
```

Notice that VB Migration Partner generates two different kinds of messages: warning #12D8 refers to statements that should be inserted *before* the next statement, whereas warning #12E8 refers to statements that should be inserted *after* the next statement.

You might mistakenly believe that warning #12E8 is superfluous and that VB Migration Partner could just generate the following C# code:

```
dynamic  _tempVar1 = x; dynamik _tempVar2 = y;

if ( Evaluate(ref _tempVar1, ref _tempVar2) > 10 )

{
    // next line might not be executed!

    x = VB6Helpers.CInt(_tempVar1); y = VB6Helpers.CInt(_tempVar2);

    VB6Project.DebugPrintLine("greater");

    // UPGRADE_ISSUE (#12D8): Unable to insert this code before next statement:

    // dynamic _tempVar3 = x; dynamic _tempVar4 = y;

}

else if ( Evaluate(ref _tempVar3, ref _tempVar4) == 0 )

{

    // next line might not be executed!

    x = VB6Helpers.CInt(_tempVar3); y = VB6Helpers.CInt(_tempVar4);

    VB6Project.DebugPrintLine("zero");

}
```

However, a closer examination of this code reveals that there is no guarantee that the lines in boldface will be executed, because it would depend on whether the conditions of the **if** and **else if** statements are true or not.

As of this writing other VB6-to-C# conversion tools aren't aware of this detail, and therefore generate code that isn't equivalent to the original VB6 code and that might introduce hard-to-find subtle bugs in the generated C# code.

Notice that the need to generate extra statements before and after the method call only depends on the fact that the Object parameter uses by-reference semantics, therefore you can reduce the number of these extra lines by attempting to remove these **ref** parameters if they aren't strictly necessary, for example by means of the CSharpOption OverloadsForByval pragma.

Another way to work around this issue is manually creating an overload for the Evaluate method that takes int parameters, and then modify the generated code to use this new overload instead:

```
public int Evaluate(ref dynamic v1, ref dynamic v2)
```

```
{
     v1 = VB6Helpers.CInt(v1) * 2;
     v2 = VB6Helpers.CInt(v2) * 4;
     return (int)(VB6Helpers.CDbl(v1) * VB6Helpers.CDbl(v2));
}

public int Evaluate(ref int v1, ref int v2)
{
     v1 = v1 * 2;
     v2 = v2 * 4;
     return ( v1 * v2 );
}

public void Test(ref int x, ref int y)
{
     if ( Evaluate(ref x, ref y) > 10 )
     {
        VB6Project.DebugPrintLine("greater");
     }
     else if ( Evaluate(ref x, ref y) == 0 )
     {
        VB6Project.DebugPrintLine("zero");
     }
}
```

C# only supports "null" as the default value for an optional Object parameter

Under Visual Basic, an optional Variant parameter can have any default value, including a number, a string, or the special Null or Empty values. Conversely, under C# an Object optional parameter's default value can only be **null**. This minor detail forces VB Migration Partner to deal the parameter as if it were non-optional in most method calls, as the following code demonstrates:

```
' VB6
Sub Test()
   ' omit all four optional variant arguments
   DoSomething
End Sub

Sub DoSomething(Optional ByVal v As Variant = Null, _
    Optional ByVal v2 As Variant = Empty, Optional ByVal v2 As Variant _
    = "missing", Optional ByVal v4 As Variant)
    ' ...
End Sub

// C#
```

```
public void Test()

{

   // omit all four optional variant arguments

   DoSomething(VB6Helpers.Null, VB6Helpers.Empty, "missing", VB6Helpers.Missing);

}



public void DoSomething(dynamic v = null, dynamic v2 = null, dynamic v2 = null,

    dynamic v4 = null)

{

    // ...

}
```

This behavior is quite common when invoking methods defined in COM libraries, especially Microsoft Office libraries, which tend to expose many by-reference optional Variant parameters. In some cases, VB Migration Partner is forced to generate additional code to preserve functional equivalence:

```
' VB6

Sub Test(ByVal wordobj As Word.Application)

    ' show that default value of optional parameters is ok for COM methods

    wordObj.ActiveDocument.CheckSpelling IgnoreUppercase:=True

End Sub



//C#

public void Test(Word.Application wordobk)
```

```
{
  // show that default value of optional parameters is ok for COM methods
  dynamic _tempVar1 = true; dynamic _tempVar2 = VB6Helpers.Missing;
  dynamic _tempVar3 = VB6Helpers.Missing; dynamic _tempVar4 = VB6Helpers.Missing;
  dynamic _tempVar5 = VB6Helpers.Missing; dynamic _tempVar6 = VB6Helpers.Missing;
  dynamic _tempVar7 = VB6Helpers.Missing; dynamic _tempVar8 = VB6Helpers.Missing;
  dynamic _tempVar9 = VB6Helpers.Missing; dynamic _tempVar10 = VB6Helpers.Missing;
  dynamic _tempVar11 = VB6Helpers.Missing; dynamic _tempVar12 = VB6Helpers.Missing;
  objword.ActiveDocument.CheckSpelling(IgnoreUppercase: ref _tempVar1,
  CustomDictionary: ref _tempVar2, AlwaysSuggest: ref _tempVar3,
  CustomDictionary2: ref _tempVar4, CustomDictionary3: ref _tempVar5,
  CustomDictionary4: ref _tempVar6, CustomDictionary5: ref _tempVar7,
  CustomDictionary6: ref _tempVar8, CustomDictionary7: ref _tempVar9,
  CustomDictionary8: ref _tempVar10, CustomDictionary9: ref _tempVar11,
  CustomDictionary10: ref _tempVar12);
}
```

While VB Migration Partner always generate code that is functional equivalent to the original VB6 code, it should be noted that **ref** optional parameters can be omitted if the method belongs to a COM interface, as explained in the following MSDN article:

http://msdn.microsoft.com/en-us/library/ms178843%28v=VS.100%29.aspx#Y570

Therefore, the resulting C# code can be often be simplified if selecting a method that belongs to a COM interface. This optimization must be performed manually, once the migration process is over.

Implementing late-bound calls with Invoke helper methods

By default, VB Migration Partner converts Variant and Object variables using the dynamic data type (new in C# 2010); however, it is also possible to use the CSharpOption UseObject pragma to convert these variables using the standard object data type, in which case all late-bound calls are converted by means of helper methods. (This approach may be necessary if you want to keep the generated C# code compatible with previous Visual Studio versions.)

```
' VB6

'## CSharpOption UseUseObject

Public Sub Test(ByVal o As Object, ByVal v As Variant)

     ' a simple late-bound call

     o.DoSomething v

     ' late-bound call whose arg is the return value of another late-bound call

     o.DoSomething v.Evaluate

     ' a chain of late-bound calls

     o.DoSomething(v).Evaluate

End Sub



// C#

public void Test(object o, object v)

{

    // a simple late-bound call

    VB6Helpers.Invoke(o, "DoSomething", v);

    // late-bound call whose arg is the return value of another late-bound call

    VB6Helpers.Invoke(o, "DoSomething", VB6Helpers.Invoke(v, "Evaluate"));

    // a chain of late-bound calls
    VB6Helpers.Invoke(VB6Helpers.Invoke(o, "DoSomething(v)"), "Evaluate");

}
```

The actual late-bound call is performed by the **VB6Helpers.Invoke** helper method; if your original VB6 makes heavy use of late binding, expect to see tons of occurrences of calls to **VB6Helper.Invoke** method in the generated code, to the point where the code becomes slow, hard to maintain and evolve. In such circumstances you should consider the opportunity to redesign the original code or modify the generated C# code or, more simply, rely on the dynamic data type.

When the late-bound code is under the scope of a DefaultMemberSupport pragma, VB Migration Partner simplifies the code by merging the calls to **VB6Helpers.Invoke** and **VB6Helpers.GetDefaultMember** into a single call to the **VB6Helpers.InvokeGetDefault** method, as this code demonstrates:

```
' VB6

Sub Test(v As Variant, o As Object)

    '## DefaultMemberSupport

    v = o.Evaluate

End Sub



// C#

public void Test(ref object v, object o)

{

     v = VB6Helpers.InvokeGetDefault(o, "Evaluate");

}
```

C# supports limited form of late-binding by means of "dynamic" members

In an important breakthrough from previous version, C# 2010 supports a somewhat limited form of late-binding by means of the **dynamic** keyword, which is basically a synonym for **object** that additionally supports late-binding both in function calls and in math operators.

By default, VB Migration Partner automatically leverages this important C# feature, as in this example:

```
' VB6

Public Sub Test(ByVal o As Object, ByVal v As Variant)

   ' a simple late-bound call

   o.DoSomething v

   ' late-bound call whose arg is the return value of another late-bound call

   o.DoSomething v.Evaluate()

   ' a chain of late-bound call

   o.DoSomething(v).Evaluate

End Sub


// C#

public void Test(dynamic o, dynamic v)

{

  // a simple late-bound call

   o.DoSomething(v);
  // late-bound call whose arg is the return value of another late-bound call
   o.DoSomething(v.Evaluate());
  // a chain of late-bound call
   o.DoSomething(v).Evaluate;
}
```

The **dynamic** keyword almost perfectly mimics the behavior of late-binding under VB6, with the following exceptions:

a) Late-bound member names are case-sensitive (as are all names in C#).
b) Late-bound method names with no arguments must end with an empty parenthesis pair - e.g. DoSomething() - to distinguish them from late-bound properties.
c) if two dynamic variables appear as operands of a "+" operator, it is considered to be a string concatenation operation if either variable contains a string. (Conversely, under VB6 it is considered an addition if either Variant operand is numeric.)
d) if a dynamic variable appears as operand of any math operator other than "+", the operation fails if the variable doesn't contain a numeric value. (Conversely, under VB6 this operation succeeds if the Variant operand contains a string that can be converted to a number.)

If a late-bound method uses the wrong casing (see point A), you should fix the original VB6 code. Likewise, if the original VB6 code invokes a late-bound method with no parameters, you should ensure that the code includes

```
res = v.Evaluate()  ' empty parenthesis are OK because Evaluate returns a value
v.Evaluate()        ' this statement doesn't compile under VB6 because Evaluate
                    ' doesn't return a value
```

By default, VB Migration Partner appends a pair of empty parenthesis only if the pair appears in the original VB6 code. This is intentional, because it allows you to distinguish between late-bound properties and late-bound methods. However, you need a ParseReplace pragma if the original VB6 statement rejects the empty parenthesis pair, as in this example:

```
'## ParseReplace v.Evaluate()
v.Evaluate
```

By default, when using dynamic variables, VB Migration Partner assumes that all variables contain data that don't require further conversion before they can be used in math operations or as arguments to methods call, as in this example:

```
' VB6
'## UseDynamic
Public Function Evaluate(ByVal v1 As Variant, ByVal v2 As Variant) As Variant
   Evaluate = v1 * v2
End Sub


// C#
public dynamic Evaluate(dynamic v1, dynamic v2)
{
   return v1 * v2;
}
```

In some cases, however, this assumption is incorrect and you might want to explicitly convert **dynamic** values into the expected data type. You can force VB Migration Partner to generate all needed conversion code by means of the CSharpOption ForceConversions pragma:

```
' VB6
'## CSharpOption ForceConversions
Public Function Evaluate(ByVal v1 As Variant, ByVal v2 As Variant) As Variant
   Evaluate = v1 * v2
End Sub
```

```
// C#

public dynamic Evaluate(dynamic v1, dynamic v2)

{

   return VB6Helpers.CDbl(v1) * VB6Helpers.CDbl(v2);

}
```

The CSharpOption ForceConversions pragma can have project, file, or method scope. If applied at the variable-level scope, the pragma has no effect on the way the variable is used inside expressions.

C# doesn't support properties with parameters

Under both VB6 and VB.NET it is legal to define any number of properties with arguments, even though only one of them can be marked with the **Default** keyword. C# doesn't support properties with arguments, even though it supports indexers (i.e. the special member named **this**).

When converting a class that contains one or more properties with arguments, VB Migration Partner generates an indexer only for the default property, and a pair of **get_propname** and **set_propname** methods for all the other properties. If none of the properties is the default property, no index is generated. Consider the following VB6 class named TestClass:

```
Private m_Items(100) As String

Private m_Names(100) As String



' Item is the default property

Public Property Get Item(ByVal index As Long) As String

    Item = m_Items(index)

End Property
```

```
Public Property Let Item(ByVal index As Long, ByVal value As String)
    m_Items(index) = value
End Property

' Name is a non-default property
Public Property Get Name(ByVal index As Long) As String
    Name = m_Names(index)
End Property

Public Property Let Name(ByVal index As Long, ByVal value As String)
    m_Names(index) = value
End Property

Sub Test()
    Dim o As New TestClass
    ' two ways for accessing the default property
    o(1) = o(2)
    o.Item(1) = o.Item(2)
    ' only one way for accessing a non-default property
    o.Name(1) = o.Name(2)
End Sub
```

Here's the corresponding C# code generated by VB Migration Partner:

```
private string[] m_Items = new string[101];

private string[] m_Names = new string[101];


// Item is the default property

public string this[int index]

{

    get

    {

       return m_Items[index];

    }

    set

    {

       m_Items[index] = value;

    }

}


// Name is a non-default property

public string get_Name(int index)

{

    return m_Names[index];

}
```

```
public void set_Name(int index, string value)

{

    m_Names[index] = value;

}



public void Test()

{

    TestClass o = new TestClass();

    // two ways for accessing the default property

    o[1] = o[2];

    o[1] = o[2];

    // only one way for accessing a non-default property

    o.set_Name(1, o.get_Name(2));

}
```

You have a special case when a class with a default Variant property with argument is under the scope of a PreservePropertyAssignmentKind pragma. In these circumstances, a Variant property that is defined by means of Property Get, Property Let, and Property Set procedures generates one more method called **setobj_propname,** that contains the code that was originally included in the Property Set procedure. For example, consider this VB6 code:

```
'## PreservePropertyAssignmentKind



' Item is the default Variant property, with distinct Get/Let/Set procedures

Public Property Get Item(ByVal index As Long) As Variant

     Item = m_Items(index)
```

```
End Property

Public Property Let Item(ByVal index As Long, ByVal value As Variant)
     m_Items(index) = value
End Property

Public Property Set Item(ByVal index As Long, ByVal value As Variant)
     Set m_Items(index) = value
End Property

Sub Test()
    Dim o As New DefaultProperty
    ' two ways for accessing the default property
    o(1) = o(2)
    Set o(1) = o(2)
End Sub
```

You can see how the PreservePropertyAssignmentKind pragma affects both the rendering of the property under C# and the client that uses the Set keyword to assign the property:

```
public object this[int index]
{
    get
     {
            return m_Items[index];
         }
        set
         {
            m_Items[index] = value;
         }
}

public void setobj_Item(int index, object value)
{
     m_Items[index] = value;
}

public void Test()
{
     TestClass o = new TestClass();
     // two ways for accessing the default property
     o[1] = o[2];
     o.setobj_Item(1, o[2]);
}
```

No other VB6-to-C# conversion software supports the ability to discern between simple assignments and Set assignments to properties that expose both the Property Let and the Property Set procedures.

There is one minor difference between Visual Basic and C# worth mentioning: C# doesn't support optional parameters in properties. When your VB6 code include a property with optional parameters, VB Migration Partner converts it into a property with all non-optional parameters, and modifies call references to that property accordingly.

C# doesn't support a keyword equivalent to ReDim Preserve

There are several differences in how Visual Basic (both VB6 and VB.NET) and C# declare arrays, the most important of which are:

- C# requires that you specify the number of elements in the array, whereas Visual Basic requires that you specify the index of the last element
- C# doesn't support the ReDim Preserve keyword

VB Migration Partner automatically compensates for the different value to be used when declaring or ReDim-ming an array, as the following code demonstrates:

```
' VB6
Public Sub DoSomething(ByVal count As Integer)
   Dim arr(10) As String
   Dim arr2() As Double
   ' ...
   ReDim arr2(count)
End Sub


// C#
public void DoSomething(short count)
```

```
{
    string[] arr = new string[11];
    double[] arr2 = null;
    // ...
    arr2 = new double[count + 1];
}
```

The ReDim Preserve keyword is supported via the **VB6Helpers.RedimPreserve** method in the support library. In this case, the last argument is the index of the last item, as it happens in VB.NET:

```
' VB6
Public Sub DoSomething(ByVal count As Integer, arr() As String)
   ReDim Preserve arr(10) As String
End Sub


// C#
public void DoSomething(short count, ref string[] arr)
{
   VB6Helpers.RedimPreserve(ref arr, 0, count);
}
```

C# doesn't support WithEvents variables

Unlike both VB6 and VB.NET, the C# language doesn't support WithEvents variables. This limitation has two important consequences. First, VB Migration Partner has to generate the code that associates a given method as the handler for a control's event, in the code-behind portion of all forms, as in the following example:

```
//

// Text1

//

this.Text1.Name = "Text1";

this.Text1.Size = new System.Drawing.Size(121, 41);

this.Text1.Location = new System.Drawing.Point(48, 24);

this.Text1.TabIndex = 0;

this.Text1.Text = "Text1";

this.Text1.Change += new CodeArchitects.VB6Library.Events.VB6EventHandler

                                                 (this.Text1_Change);

this.Text1.GotFocus += new CodeArchitects.VB6Library.Events.VB6EventHandler

                                                 (this.Text1_GotFocus);

this.Text1.KeyPress += new CodeArchitects.VB6Library.Events.VB6KeyPressEventHandler

                                                 (this.Text1_KeyPress);
```

Second, VB Migration Partner converts a WithEvent variables into a property that wraps a private variable, and the property's setter contains the code that associates the object's events with the methods that handle that event. Here's an example:

```
' VB6

Dim WithEvents txtField As TextBox
```

```
' event handlers

Private Sub txtField_Change()

     Debug.Print "Field has changed"

End Sub



Private Sub txtField_KeyPress(KeyAscii As Integer)

     Debug.Print "A key has been pressed"

End Sub



// C#

private VB6TextBox txtField_InnerField;



private VB6TextBox txtField

{

    get

    {

        return txtField_InnerField;

    }

    set

    {

        if (object.Equals(txtField_InnerField, value))

           return;
        if (txtField_InnerField != null)
        {
            txtField_InnerField.Change -= txtField_Change;
            txtField_InnerField.KeyPress -= txtField_KeyPress;
        }
        txtField_InnerField = value;
        if (txtField_InnerField != null)
        {
             txtField_InnerField.Change += txtField_Change;
             txtField_InnerField.KeyPress += txtField_KeyPress;
        }
     }
 }

 // event handlers
private void txtField_Change()
{
    VB6Project.DebugPrintLine("Field has changed");
}

private void txtField_KeyPress(ref short KeyAscii)
{
    VB6Project.DebugPrintLine("A key has been pressed");
}
```

The C# implementation of a WithEvents variable has exactly the same behavior as in VB6, with a caveat: if you later extend/improve the application by adding another event handler for that variable, it's up to you to manually modify the wrapper property to account for the new handlers. (In VB.NET this manual step is never necessary.)

C# has a different syntax to define and raise events

Under VB6 and VB.NET you define an event by means of the Event keyword, and trigger the event by means of the RaiseEvent keyword:

```
' inside a class or user control
Public Event Change(ByVal oldText As String, ByVal newText As String)
    ' …
Private Sub OnChange(ByVal oldText As String, ByVal newText As String)
    RaiseEvent Change(oldText, newText)
End Sub
```

C# supports the **event** keyword, however it requires that you define a delegate type that matches the event signature; also, C# doesn't directly support the RaiseEvent keyword and requires that you manually invoke the delegate. This is the code that VB Migration Partner produces:

```
#region Project-level delegate classes
     public delegate void ChangeEventHandler(string oldText, string newText);
#endregion


public event ChangeEventHandler Change;
```

```
    // …

private void OnChange(string oldText, string newText)

{

    if (Change != null)

    {

        Change(oldText, newText);

    }

}
```

Interestingly, VB Migration Partner never generates a delegate type unless it's strictly necessary to do so. For example, if the event signature matches a delegate that is already defined in CodeArchitects.VBLibrary.dll, then VB Migration Partner uses the delegate in the library instead of defining a delegate type inside the current project.

All the delegate types defined for the current project are gathered in a region named **Project-level delegate classes**, which is inserted in the first generated C# file.

C# can implement an interface either implicitly or explicitly

Under VB6 you specify that a method in a class is implementing an interface member by means of a naming convention, i.e. the method must be named *interfacename_membername*. Under VB.NET you get the same result by using the Implements keyword immediately after the method declaration.

C# provides two ways to reach the same goal: you can implement the interface member implicitly (by defining a public method that has the same name as the interface member) or implicitly (by creating a private method named *interfacename.membername*).

Consider the following VB6 classes, and notice that we've used the ClassRenderMode pragma to tell VB Migration Partner that we are only interested in using IWidget.cls to generate a .NET interface:

```
' --- the IWidget.cls class is actually used to define an interface
```

```
'## ClassRenderMode Interface
Public TestField As Double

Sub TestSub(ByVal x As Integer)
End Sub

Property Get TestProperty() As Integer
End Property

' --- the TestWidget.cls class implements the IWidget interface
Implements IWidget

Private Property Let IWidget_TestField(ByVal RHS As Double)
   ' ...
End Property

Private Property Get IWidget_TestField() As Double
   ' ...
End Property

Private Sub IWidget_TestSub(ByVal x As Integer)
   ' ...
```

```
End Sub


Private Property Get IWidget_TestProperty() As Integer
   ' ...
End Property
```

By default, VB Migration Partner uses the implicit implementation, and this is the C# that it produces:

```
internal interface IWidget
{
     double TestField {get; set;}
     void TestSub(short x);
     short TestProperty {get; set;}
}


internal class TestWidget : IWidget
{
     public double TestField
     {
        get
        {
           // ...
        }
        set
        {
           // ...
        }
     }

     public void TestSub(short x)
     {
          // ...
    }

    public short TestProperty
    {
        get
        {
            // ...
        }
    }
}
```

In some cases, you might want to have VB Migration Partner implement the interface explicitly. For example, this arrangement might be necessary to avoid name collisions, or desirable to meet your company's coding guidelines. You can obtain the explicit interface implementation by simply adding a CSharpOption ExplicitInterfaceImplementation pragma. If this pragma is added to the TestWidget class, the generated C# code becomes:

```
internal class TestWidget : IWidget
{
    double IWidget.TestField
    {
        get
        {
           // ...
        }
        set
        {
           // ...
        }
    }

    void IWidget.TestSub(short x)
    {
       // ...
    }

    short IWidget.TestProperty
    {
       get
```

```
        {
                // ...
        }
    }
}
```

You can CSharpOption ExplicitInterfaceImplementation pragma at the project level (to affect all interface implementations), at the file level (to affect the way a single class implements one or more interfaces), or the method level (to specify the only interface members that should be implemented explicitly).

In general, you should implement an interface implicitly if possible, because it helps VB Migration Partner to generate more linear code when the interface member is invoked, but there are cases when explicit implementation is the only viable choice.

As of this writing, no other VB6-to-C# conversion software supports the ability to explicitly implement the members of an interface.

C# doesn't support the On Error Goto keyword

VB Migration Partner automatically attempts to convert On Error Goto statements into **try-catch** blocks, as if a UseTryCatch pragma were specified at the project level.

In some cases, however, it is impossible to generate a try-catch block that is equivalent to the original VB6 code. When this happens, VB Migration Partner emits a migration message. For example, this is what happens if the execution "flows" into the error handler, as in this code:

```
' VB6
Sub Test(ByVal x As Long)
On Error GoTo ErrorHandler

    Dim z As Long
    z = x - 1

    x = x / z

' flows into the error handler, which makes the try-catch not 100% equivalent

ErrorHandler:

    Dim y As Long

    y = 12

    x = y

End Sub



// C#

// UPGRADE_ISSUE (#04E8): Unable to use Try-Catch in current method.

public void Test(int x)

{

    int z = x - 1;

    x /= z;

// flows into the error handler, which makes the try-catch not 100% equivalent

ErrorHandler:

    int y = 12;

    x = y;

}
```

The UPGRADE_ISSUE #04E8 message is also emitted when the method contains multiple On Error Goto statements, or one On Error Goto 0 statement that disables a previous On Error Goto statement, or a Resume keyword.

If you see the #04E8 migration warning you can still force VB Migration Partner to generate a try-catch block, even if the generated code isn't functionally equivalent to the original VB6 code, by applying a project-, class- or method-level CSharpOption ConvertOnErrorGoto pragma. For example, if this pragma is used in previous VB6 code, this is the resulting C# code:

```
public void NotEquivalentTryCatch(int x)
{
   int z = 0;
   int y = 0;

   try
   {
     // UPGRADE_ISSUE (#04A8): On Error Goto statement has been converted to a try-catch
     //                        block that isn't functionally equivalent
     // IGNORED: On Error GoTo ErrorHandler
             z = x - 1;
             x /= z;
     // flows into the error handler, which makes the try-catch not 100% equivalent
   }
   catch (Exception _ex)
   {
        // IGNORED: ErrorHandler:
        VB6Helpers.SetError(_ex);
        y = 12;
        x = y;
    }
}
```

Notice that VB Migration Partner has generated a different UPGRADE_ISSUE message this time, and you have the responsibility to manually fix the code after the migration. Also notice that **Resume <label>** statements are converted into **goto <label>** C# statements, which nearly always cause a compilation error, because the Resume statement and the target label are located in different blocks. **Resume Next** statements are always remarked and a migration warning is issued.

As of this writing, no other VB6-to-C# conversion programs emits a warning if the try-catch block isn't functionally equivalent to the original VB6 code.

C# doesn't support the On Error Resume Next keyword

C# doesn't support the ability to just ignore errors, as it happens with VB6 and VB.NET when you specify the On Error Resume Next keyword. By default, VB Migration Partner ignores this keyword and just emits an UPGRADE_ISSUE #04E8 message when one is found:

```
' VB6
Function Test(ByVal x As Long) As Long
    On Error Resume Next
    Test = 100 / x
End Function


// C#
// UPGRADE_ISSUE (#04E8): Unable to use Try-Catch in current method.
```

```
public int Test(int x)

{

     VB6Helpers.ClearError();

     return 100 / x;

}
```

(Notice that a call to **VB6Helpers.ClearError** method is inserted, to mimic the fact that the On Error Resume Next keyword resets the Err object.)

You can force VB Migration Partner to recognize the On Error Resume Next keyword by inserting the CSharpOption ConvertOnErrorResumeNext pragma, at the project, file, or method level. When this pragma is applied to previous VB6 code, the generated C# code becomes:

```
public int Test(int x)

{

    int _retValue = 0;

    VB6Helpers.ClearError();

    VB6Helpers.IgnoreError( () => {_retValue = 100 / x;} );

    return _retValue;

}
```

As you see, individual statements under the scope of the On Error Resume Next keyword are rendered under C# as anonymous lambda methods (i.e. the **() =>** syntax). Lambda methods have several limitations and shortcomings, one of which is the fact that they reduce the readability and maintainability of the generated C# code.

VB Migration Partner attempts to reduce code cluttering by gathering multiple anonymous lambdas inside a single call to the **VB6Helpers.IgnoreError** helper method:

```
VB6Helpers.IgnoreError(

    () => { x = y + 10;},
    () => { y = z ^ 3} );
```

An important detail: the **VB6Helpers.IgnoreError** helper method executes all the anonymous lambda methods passed to it as arguments, and internally invokes the **VB6Helpers.SetError** method if any of those lambda calls throw an exception. This behavior perfectly mimics what VB6 does.

Things become more complex if the method contains If, Do, or While blocks. In this case, VB Migration Partner generates a call to the **VB6Helpers.IgnoreErrorBool** helper method:

```
' VB6

Public Function CaptionIsEmptyOrMissing(ByVal o As Object) As Boolean

    On Error Resume Next

    ' next statement may fail if “o” doesn’t expose a property named Caption

    ' in which case the “next” statement assigns True to the return value

    If o.Caption = "" Then

        CaptionIsEmptyOrMissing = True

    End If

End Function


// C#

public bool CaptionIsEmptyOrMissing(object o)

{

    bool _retValue = false;

    VB6Helpers.ClearError();

  // next statement may fail if "o" doesn't expose a property named Caption

  // in which case the "next" statement assigns True to the return value
```

```
    if (VB6Helpers.IgnoreErrorBool(()=>
       {return VB6Helpers.Invoke(o,"Caption")== "";}))
    {
       _retValue = true;
    }
    return _retValue;
}
```

The call to **VB6Helpers.IgnoreErrorBool** helper method is generated only if the expression may throw, and is omitted for simple comparison expression such as **x <> 0** or similar. Also, the helper method returns the value of the expression passed to it, or true if the expression threw an exception. Returning true when an error occurs ensures that the "next" statement is executed, if the helper method is invoked from inside an **if**, **elseif** or **while** block.

A serious limitation of anonymous lambda methods is that they can't reference any ref/out parameter, because such a reference would cause a C# compilation error. When this happens, VB Migration Partner generates a one-line **try-catch** block that wraps only the statement in question, as shown in this example:

```
' VB6
Public Sub Test(x As Long, ByVal y As Long)
   On Error Resume Next
     y = y + 1
     ' next statement uses a by-reference parameter
     x = x * 2
End Sub
```

```
// C#
public void Test(ref int x, int y)
{
    VB6Helpers.ClearError();
    VB6Helpers.IgnoreError( () => {y++;} );
    // next statement uses a by-reference parameter
    try { x *= 2;  }
    catch (Exception _ex) { VB6Helpers.SetError(_ex); }
}
```

If the original code contains an If, ElseIf, or Do block that uses an expression that references a ref/out parameter, the code isn't wrapped inside a call to VB6Helpers.IgnoreErrorBool, but VB Migration Partner emits an UPDATE_ISSUE warning. It is your responsibility to check whether the expression can throw and, if this is the case, manually fix the generated code.

Other VB6-to-C# migration tools may use a similar technique based on anonymous lambda methods to convert On Error Resume Next statements; however, only VB Migration Partner correctly accounts for special cases such as ref/out parameters in lambda methods and typically generates fewer C# compilation errors than its competitors.

C# doesn't expose the Err object

Unlike Visual Basic, C# doesn't keep track of the most error by means of the **Err** object. VB Migration Partner remedies by defining the **VB6Helpers.Err** object, which exposes the same properties and methods as the VB6 Err object:

```
' VB6
Err.Clear
Err.Raise 123, , "this is a description"
```

```
Debug.Print Err.Description


// C#
VB6Helpers.Err.Clear();
VB6Helpers.Err.Raise(123, null, "this is a description");
VB6Project.DebugPrintLine(VB6Helpers.Err.Description);
```

The generated C# code initializes the Err “pseudo-object” whenever an exception is thrown, by means of a call to the **VB6Helpers.SetError** method that VB Migration Partner generates in a **try-catch** block:

```
' VB6
Sub Test(ByVal x As Long, ByVal y As Long, ByVal z As Long)
    On Error GoTo ErrorHandler
    x = x / y
    Exit Sub

    ErrorHandler:
    x = 123
End Sub


// C#
public void Test(int x, int y, int z)
{
    try
    {
      // IGNORED: On Error GoTo ErrorHandler
      x /= y;
      return;
    }
    catch (Exception _ex)
    {
      // IGNORED: ErrorHandler:
      VB6Helpers.SetError(_ex);
      x = 123;
    }
}
```

The **VB6Helpers.SetError** method is also invoked behind the scenes by the **VB6Helpers.IgnoreError** and **VB6Helpers.IgnoreErrorBool** methods, which VB Migration Partner generates for code portions that contain an On Error Resume Next statement and that are under the scope of a CSharpOption ConvertOnErrorResumeNext pragma.

C# requires strict typing (Option Strict On)

C# is a "very strongly-typed" language, and requires that you explicitly convert between different data types any time an overflow error (or an error of another nature) might otherwise occur. (In this respect, C# is pickier than VB.NET when the Option Strict On directive is used.) VB Migration can automatically insert all the conversion and cast methods that are required

```
' VB6
Public Sub Test(ByVal x As Integer, ByVal y As Long, ByVal s As String)
```

```
    x = y
    y = s
    ' ...
End Sub


// C#
public void Test(short x, int y)
{
    x = (short)y;
    y = VB6Helpers.CInt(s);
    // ...
}
```

A minor annoyance caused by C# strict typing is that all integer math operators return a 32-bit integer, even if both operators are 8-bit or 16-bit values. This means that assigning the result of a sum, a subtraction, a multiplication, etc. to a variable that was declared as a VB6 Integer variable always generates a cast to **short**, as this code demonstrates:

```
' VB6
Public Sub DoubleIt(ByRef x As Integer)
     x = x * 2
End Sub


// C#
public void DoubleIt(ref short x)
```

```
{
    x = (short)(x * 2);
}
```

You can reduce the number of such cast operations by ensuring that all local variables be declared as Long rather than Byte or Integer. This can be easily obtained by means of a PreProcess pragma, for example:

```
'## project:PreProcess "\b(?<dec>(Dim|Private|Public) \w+) As Integer\b",
    "${dec} As Long"
```

C# has both bitwise and logical And/Or operators

VB6 offers only bitwise And/Or operators, but the fact that True and False Boolean values are represented by -1 (all bits set) and 0 (all bits cleared) means that, in practice, you can use these bitwise operators as if they were logical operators.

VB.NET offers both bitwise and logical operators, where the logical AndAlso and OrElse operator provide short-circuit evaluation and can avoid the evaluation of subexpressions whose value wouldn't alter the overall True/False value of the entire expression. However, when converting from VB6 to VB.NET VB Migration Partner doesn't attempt to convert And/Or into AndAlso/OrElse, because the added short-circuit evaluation feature might alter the code behavior. (You can force the conversion to AndAlso/OrElse by means of the LogicalOps pragma.)

When converting to C#, VB Migration Partner detects whether the And/Or bitwise operators were used as logical operators, and correctly emits the **&&** and **||** operators if this is the case. We decided to change the conversion default because C# developers would find it quite unnatural using bitwise operators where a logical operator would be more appropriate. This behavior is by default and can't be changed (in other words, the LogicalOps pragma has no effect when converting to C#).

In the vast majority of cases replacing the original bitwise operator with its logical counterpart makes your code more efficient, especially if you also specify the MergeIfs pragma to merge nested IF blocks in a single expression.

However, there are a few cases when this behavior might introduce subtle bugs in your application. Consider the following code:

```
' VB6
If count <> 0 And Increment(count) > 10 Then DoSomething(count)
'...

Function Increment(x As Long) As Long
   x = x + 1
   Increment = x
End Function

// C#
If (count != 0 && Increment(ref count) > 0)
   DoSomething(count);

// ...
public int Increment(ref int x)
{
      x++
      return x;
}
```

The problem in previous code occurs if count is zero, because in such a case the VB6 code invokes the Increment method anyway and therefore increments the variable, whereas the C# code doesn't.

The bottom line: you should always check that complex Boolean expressions don't invoke methods with ByRef parameters or that have other undesired side effect, such as altering global variables. (Such checks aren't necessary if the Boolean expresson was the result of the MergeIfs pragma.)

C# supports neither the <, <=, >, and >= string operators nor the Option Compare Text directive

A big surprise for VB developers switching to C# is that this latter language only supports the equal (==) and nonequal (!=) operators to compare strings. When any other comparison operator is used between strings, VB Migration Partner generates a call to the **VB6Helpers.StrComp** helper methods:

```
' VB6

Sub TestStringComparisons(x As String, y As String)

    If x = y Then

        Debug.Print "Equal"

    ElseIf x <> y Then

        Debug.Print "Different"

    ElseIf x > y Then

        Debug.Print "Greater"

    ElseIf x >= y Then

        Debug.Print "Greater or Equal"

    ElseIf x < y Then

        Debug.Print "Less"

    ElseIf x <= y Then
```

```
        Debug.Print "Less or Equal"
    End If
End Sub


// C#
public void TestStringComparisons(ref string x, ref string y)
{
     if ( x == y )
     {
          VB6Project.DebugPrintLine("Equal");
     }
     else if ( x != y )
     {
          VB6Project.DebugPrintLine("Different");
     }
     else if ( VB6Helpers.StrComp(x, y) > 0 )
     {
          VB6Project.DebugPrintLine("Greater");
     }
     else if ( VB6Helpers.StrComp(x, y) >= 0 )
     {
          VB6Project.DebugPrintLine("Greater or Equal");
     }
     else if ( VB6Helpers.StrComp(x, y) < 0 )
     {
          VB6Project.DebugPrintLine("Less");
     }
     else if ( VB6Helpers.StrComp(x, y) <= 0 )
     {
          VB6Project.DebugPrintLine("Less or Equal");
     }
}
```

To further complicate things, both VB6 and VB.NET support the Option Compare Text directive, which forces all string comparisons to be case-insensitive. C# lacks this feature, therefore it has to generate slightly different code for files where this directive is specified. For example, if the previous VB6 code snippet is defined in a file containing the Option Compare Text directive, this is the C# code that VB Migration Partner generates:

```
public void TestStringComparisons(ref string x, ref string y)
{
    if ( VB6Helpers.StrComp(x, y, CompareMethod.Text) == 0 )
    {
        VB6Project.DebugPrintLine("Equal");
    }
    else if ( VB6Helpers.StrComp(x, y, CompareMethod.Text) != 0 )
    {
        VB6Project.DebugPrintLine("Different");
    }
    else if ( VB6Helpers.StrComp(x, y, CompareMethod.Text) > 0 )
    {
        VB6Project.DebugPrintLine("Greater");
    }
    else if ( VB6Helpers.StrComp(x, y, CompareMethod.Text) >= 0 )
    {
        VB6Project.DebugPrintLine("Greater or Equal");
    }
    else if ( VB6Helpers.StrComp(x, y, CompareMethod.Text) < 0 )
    {
        VB6Project.DebugPrintLine("Less");
    }
    else if ( VB6Helpers.StrComp(x, y, CompareMethod.Text) <= 0 )
    {
        VB6Project.DebugPrintLine("Less or Equal");
    }
}
```

The Option Compare Text directive also affects a few string methods, namely InStr, InstrRev, and Replace. VB Migration Partner generates the correct C# code when these methods are used in files under the scope of an Option Compare Text directive.

As of this writing, no other VB6-to-C# code converter honors the Option Compare Text directive.

C# uses the "+" operator for both addition and string concatenation

When translating VB6 code that uses the "+" operator and the dynamic data type is not used (that is, the CSharpOption UseObject is present), it is essential to understand whether the operator stands for "addition" or "concatenation", because it might be necessary to adjust the type of one or both operands.

Understanding the correct type is very simple when both operands are numbers or both operands are strings, but the rules becomes confused if one or both arguments are object variables. To make a correct guess of the correct operator VB Migration Partner looks at several clues, including the type of the variable/parameter the result is assigned or passed to.

If it is impossible to unambiguously detect the operator type, however, VB Migration Partner tends to opt for the "addition" operator, and consequently converts both operands to double, using the **VB6Helpers.CDbl** helper method, as in this example:

```
' VB6
Public Sub Test(ByVal v1 As Variant, ByVal v2 As Variant, ByVal n As Double, _
       ByVal s As String)
    n = v1 + v2
    s = v1 + v2
    v = v1 + v2
End Sub


// C#
public void Test(object v1, object v2, double n, string s)
{
    n = VB6Helpers.CDbl(v1) + VB6Helpers.CDbl(v2);   // addition is assumed

    s = VB6Helpers.CStr(v1) + VB6Helpers.CStr(v2);   // concatenation is assumed

    o = VB6Helpers.CDbl(v1) + VB6Helpers.CDbl(v2);   // addition is assumed

}
```

If VB Migration Partner's assumption isn't correct, you should edit the original VB6 code to let the code analyzer better understand your code. For example, in above code you can force the last assignment to use string concatenation by using a CStr method in original VB6 code:

```
      v = v1 + CStr(v2)
```

Alternatively, you can use the CSharpOption AssumeConcatenation pragma to force VB Migration Partner to use string concatenation when the available information is ambiguous or non-conclusive. This pragma can have a project-, file-, or method-level scope, therefore different portions of the VB6 project being migrated can use different default behaviors.

Notice that this problem doesn't manifest when the dynamic data type is used, in which case, the following C# code is generated and the operator type is determined dynamically at runtime:

```
public void Test(dynamic v1, dynamic v2, double n, string s)

{

    n = v1 + v2;

    s = v1 + v2;

    o = v1 + v2;

}
```

C# doesn't support comparisons between Object variables

Being a strongly-typed language, C# doesn't support comparison operators between Object variables (which might be generated by the conversion of VB6 Variant variables), therefore VB Migration Partner has to generate code that converts one or both the operands of a comparison.

Notice that the following considerations only applies if the CSharpOption UseObject pragma is used, which inhibits the generation of dynamic variables.

When only one operand is an Object variable, VB Migration Partner assumes that such variable is of the same type as the other, non-Object operand. If the non-Object variable is numeric, VB Migration Partner assumes that you are comparing two numbers and converts the Object variable to a numeric type; if the non-Object variable is a string, VB Migration Partner assumes that you are comparing two strings, therefore converts the Object variable to a string and generates the most appropriate string operator or method. (For example, if the code is under the scope of an Option Compare Text directive then all comparison operators are converted to calls to **VB6Helpers.StrComp** method.)

```
' VB6 (assumes no Option Compare Text directive is present)
Public Sub Test(ByVal v1 As Variant, ByVal v2 As Variant, _
           ByVal n As Double, ByVal s As String, ByVal res As Boolean)
    res = (v1 > n)
    res = (v1 > s)
    res = (v1 > v2)
End Sub


// C#
public void Test(object v1, object v2, double n, string s, bool res)
{
    res = ( VB6Helpers.CDbl(v1) > n );
    res = ( VB6Helpers.StrComp(VB6Helpers.CStr(v1), s) > 0);
    res = ( VB6Helpers.CDbl(v1) > VB6Helpers.CDbl(v2) );
}
```

As you see in previous code, when both operands of a comparison are Object variable, by default VB Migration Partner assumes that both of them are numbers and therefore converts them using the **VB6Helpers.CDbl** method.

You can change this default behavior by inserting a CSharpOption AssumeStringComparison pragma. Like all CSharpOption pragmas, you can specify it at the project, file, and method level, therefore you have all the necessary granularity in how you affect the code generator behavior.

Another way to avoid or reduce this problem is using the UseDynamic pragma.

No other VB6-to-C# code converter allows you to decide whether a late-bound comparison is actually a numeric or a string comparison, and therefore force you to either fix the original VB6 code or the generated C# code.

C# doesn't support comparison operators and the TO keyword in a switch block

The **switch** C# keyword broadly corresponds to Visual Basic's Select Case keyword, however once you look a bit more closely you see that there are many differences and that the **switch** statement is far less flexible than VB's Select Case. For example, C#'s **case** blocks can only check for equality, don't support comparison operators such as <>, <, <=, > and >=, and don't support ranges (i.e. the To keyword). In addition, the **switch** keyword only supports bool, int, string, and enum expressions.

VB Migration Partner always attempts to convert a Select Case into a **switch** block, because this keyword can be optimized by the C# compiler into a jump table and in general delivers better performance. If using a **switch** block isn't possible, VB Migration Partner reverts to a more flexible **if...elseif** block, as the following code demonstrates:

```
' VB6

Public Sub Test(ByVal x As Integer, ByVal s As Single)

    Select Case x

       Case 1, 2

            Debug.Print "First case"
       Case Is = 3
            Debug.Print "Second case"
       Case Else
            Debug.Print "Third case"
    End Select

    Select Case s
       Case 1, 2
            Debug.Print "One or Two"
       Case < 0
            Debug.Print "Negative"
    End Select
End Sub

// C#
public void Test(int x, float s)
{
    switch ( x )
    {
      case 1: case 2:
          VB6Project.DebugPrintLine("First case");
          break;
      case 3:
          VB6Project.DebugPrintLine("Second case");
          break;
          default:
          VB6Project.DebugPrintLine("Third case");
          break;
    }

    if ( s == 1 || s == 2 )
    {
        VB6Project.DebugPrintLine("One or Two");
    }
    else if ( s < 0 )
    {
       VB6Project.DebugPrintLine("Negative");
    }
}
```

The main problem with **if…elseif** blocks is that the expression should be repeatedly evaluated in each **else if** block, a detail that can slow down your code or – worse – can lead to subtle bugs if the expression invokes one or more functions that modify their parameters or have other side effects (e.g. modify global variables). In converting a Select Case that tests an expression (as opposed to a simple variable), VB Migration Partner assigns the expression to a temporary variable and then uses this temporary variable in the **if…elseif** block, as this code demonstrates:

```
' VB6
```

```
Select Case Rnd * 100

    Case Is > 1

       Debug.Print "One"

    Case 2 To 9

       Debug.Print "Two"

    Case 10 To 20, 30 To 40

       Debug.Print "Three or more"

End Select


// C#

float _switchVar1 = VB6Helpers.Rnd() * 100;

if ( _switchVar1 > 1 )

{

    VB6Project.DebugPrintLine("One");

}

else if ( _switchVar1 >= 2 && _switchVar1 <= 9 )

{

    VB6Project.DebugPrintLine("Two");

}

else if((_switchVar1>=10 && _switchVar1<=20)||(_switchVar1>=30 && _switchVar1<=40)

{

    VB6Project.DebugPrintLine("Three or more");
}
```

C# foreach loops variables can't be passed to a ref/out parameter

Here are two minor difference between Visual Basic (both VB6 and VB.NET) and C#: first, the controlling variable of a **foreach** loop must be declared at the top of the foreach loop; second, the controlling variable cannot be passed to a by-reference parameter when invoking a method. (Passing it to a by-value parameter is ok.) In fact, when passing a foreach controlling variable to a by-reference parameter the C# compiler generates the following compilation error:

Cannot pass 'varname' as a ref or out argument because it is a 'foreach iteration variable

VB Migration Partner works around the first problem by creating a new variable as the **foreach** controlling variable. Interestingly, this technique indirectly solves also the second problem, as this code demonstrates:

```
' VB6

Public Sub Test(ByVal col As Collection)

    Dim v As Variant

    For Each v In col

       ' here we assume that DoSomething takes a by-ref argument

       DoSomething v

    Next

End Sub



// C#
```

```
public void Test(Collection6 col)

{

    object v = null;

    foreach (object v_var in col)

    {

        v = v_var;

        DoSomething(ref s);

    }

    v = null;

}
```

Notice in previous code that VB Migration Partner has created an aliased variable named **v_var** as the loop controlling variable, and assigns **v = v_var** so that all the code inside the loop works as expected; it is now possible to pass the **v** variable to the first parameter of the DoSomething method without causing a compilation error, because **v** isn't the loop controlling variable.

Also notice that the **v** variable is set to **null** when the loop is exited, because this is the value that it is assigned when a VB6 For Each loop terminates.

Optionally, you can use the CSharpOption OptimizeForEach to have VB Migration Partner generate foreach loops that don't use the aliased **name_var** variable. More precisely, this optimization can be applied if the variable in question is used only inside the foreach loop. This pragma, however, can generate one or more compilation errors if the loop variable is passed to a by-reference parameter. Consider the following code:

```
' VB6

'## CSharpOption OptimizeForEach

Public Sub Test(ByVal col As Collection)

   Dim v As Variant, v2 As Variant
   For Each v In col

       DoSomething v

   Next

   v2 = "success"


   For Each v2 In col

       DoSomething v2

   Next

End Sub


// C#

public void Test(Collection6 col)

{

    object v2 = null;

    foreach (object v in col)

    {

        DoSomething(ref v);

    }

    v2 = "success";


    foreach (object v2_var in col)

    {
        v2 = v2_var;

        DoSomething(ref v2);

    }

    v2 = null;

}
```

Notice that VB Migration Partner could avoid using an aliased variable the first foreach loop, because the **v** variable only appears inside the loop, whereas it couldn't apply this optimization to the second loop because the **v2** variable appears outside the loop.

However, in this particular case the CSharpOption OptimizeForEach pragma causes a compilation error, because the loop variable is passed to a by-reference parameter of the DoSomething method.

If you see this compilation error you have two options. First, if all the loops in the method suffer from this problem, override the file-level pragma with a method-level pragma that disables the optimization, e.g.:

```
'## CSharpOption OptimizeForEach



Public Sub Test(ByVal col As Collection)

  '## CSharpOption OptimizeForEach, False

    ' …

End Sub
```

Alternatively, you can decide which foreach loops should be optimized by adding a "dummy" VB6 statement outside the loop. In the above code snippet, you might add the following statement to make VB Migration Partner believe that the **v** variable is used outside the loop:

```
Public Sub Test(ByVal col As Collection)

   Dim v As Variant, v2 As Variant

   Set v = Nothing
   For Each v In col
         DoSomething v
   Next
   ' …
End Sub
```

C# doesn’t support multi-level "break" keywords

In Visual Basic (both VB6 and VB.NET) you must use the appropriate Exit keyword to exit loops; in other words you must use Exit For to exit a For or For Next loop, Exit Do to exit a Do loop, or Exit While to exit a while loop. Conversely, in C# you use the **break** statement to exit all sorts of loops.

While the C# approach seems to be more concise and elegant, it has a serious limitation. In Visual Basic you can use an Exit keyword to exit deeply nested loops, as this code demonstrates:

```
' VB6
Sub Test(ByVal x As Long, ByVal y As Long)
    For x = 1 To 100
       y = x
       Do While y < 100
           y = y + 1
           ' next Exit For keyword exits both Do and For loops
           If x * y > 10000 Then Exit For
             ' ...
       Loop
    Next
    ' ...
End Sub
```

VB Migration Partner correctly handles these cases, by adding a **goto** statement:

```
// C#
public void Test(int x, int y)
{
     for (x = 1; x <= 100; x++)
     {
       y = x;
       while ( y < 100 )
       {
          y++;
          if ( x * y > 10000 )
          {
             goto ExitLoop_1;
          }
          // ...
       }
     }
  ExitLoop_1:
    // ...
}
```

No other VB6-to-C# conversion programs accounts for Exit keywords inside nested loops and generate code that compiles correctly but is severely bugged.

C# doesn't support With...End With blocks

Both VB6 and VB.NET support With…End With blocks, whereas C# doesn't. VB Migration Partner works around this limitation in two ways: if the With expression is a simple variable, then the variable is repeated for each statements generated inside the With block. Conversely, if the With expression is an expression, then VB Migration Partner generates a temporary variable that is assigned the expression, and then uses the temporary variable for statements inside the With block:

```
' VB6

Dim wid As New Widget

With wid

   .Name = "test"

   .ID = 1234

   ' CreateNewOrder returns a WidgetOrder object

   With .CreateNewOrder()

      .ID = 99

      .Qty = 12

   End With

End With



// C#
```

```
Widget wid = new Widget();

wid.Name = "test";

wid.ID = 1234;


WidgetOrder _withVar1 = wid.CreateNewOrder();

_withVar1.ID = 99;

_withVar1.Qty = 12;
```

Notice that VB Migration Partner detected that the variable for the nested With block is a function call, therefore correctly generates the **_withVar1** temporary variable to ensure that the function is invoked only once.

C# doesn't support the Declare statement

Visual Basic uses the Declare statement to define methods defined in an external DLL, whereas C# requires that these methods be defined as **static extern** methods flagged with the DllExport attribute.

VB Migration Partner takes care of all the necessary conversion details, as shown by following example:

```
' VB6

Public Declare Function Beep Lib "kernel32.dll" (ByVal dwFreq As Long, _
        ByVal dwDuration As Long) As Long
Public Declare Function SendMessage Lib "user32.dll" Alias "SendMessageA" _
        (ByVal hwnd As Long, ByVal wMsg As Long, ByVal wParam As Long, _
        ByRef lParam As Any) As Long
Public Declare Function GetWindowText Lib "user32.dll" Alias "GetWindowTextA" _
        (ByVal hwnd As Long, ByVal lpString As String, ByVal cch As Long) As Long

Sub Test()
    ' display the caption of frmMain form
    Dim wincaption As String
    wincaption = Space(255)
    wincaption = Left(wincaption, GetWindowText(frmMain.hWnd, wincaption, 255))
    Debug.Print wincaption
End Sub

' C#
// List of delegates used for callback methods
public delegate int EnumWindows_CBK(int handle, int lParam);
// UPGRADE_INFO (#01F1): References to the 'Beep' were replaced by calls to the
// 'Console.Beep' method. Check that this Declare is useless and then delete it.
[DllImport("kernel32.dll")]
public static extern int Beep(int dwFreq, int dwDuration);
```

```
[DllImport("user32.dll", EntryPoint="SendMessageA")]

public static extern int SendMessage(int hwnd, int wMsg, int wParam, ref int lParam);

[DllImport("user32.dll", EntryPoint="SendMessageA")]

public static extern int SendMessage(int hwnd, int wMsg, int wParam, int lParam);

[DllImport("user32.dll", EntryPoint="SendMessageA")]

public static extern int SendMessage(int hwnd, int wMsg, int wParam, string Param);


[DllImport("user32.dll", EntryPoint="GetWindowTextA")]

public static extern int GetWindowText(int hwnd, [MarshalAs_

                      (UnmanagedType.VBByRefStr)] ref string lpString, int cch);

[DllImport("user32.dll")]

public static extern int EnumWindows(int lpEnumFunc, int lParam);

[DllImport("user32.dll", EntryPoint="EnumWindows")]

public static extern int EnumWindows_Private(EnumWindows_CBK lpEnumFunc, int lParam);


public void Test()

{

    ' display the caption of frmMain form

    string wincaption = VB6Helpers.Space(255)

    // UPGRADE_WARNING (#0384): Using the 'GetWindowText' Windows API method as an

    //             argument to the 'Left' function might result in a string filled
    //            with spaces. Please split the next line in two statements.

    wincaption = VB6Helpers.Left(wincaption, GetWindowText(frmMain.hWnd,

                 ref wincaption, 255));

    VB6Project.DebugPrintLine(wincaption);

}
```

VB Migration Partner has performed several tasks in addition to the mere translation of the Declare keyword, namely:

- detects whether a Windows API call (Beep in this case) can be safely replaced by a native .NET method;
- detects that an external method (EnumWindows in this case) uses a callback, and correctly generates the delegate that defined the callback method's signature;
- generates one or more overloads for all Declare statements that contain an **As Any** parameter (SendMessage in this case)
- correctly adds the **MarshalAs(UnmanagedType.VBByRefStr)]** attribute for ByVal string parameters that are actually passed by reference
- generates a warning if an API method that takes a by-reference string is used incorrectly and might generate a string filled with spaces (see the Test method)

As of this writing, no other VB6-to-C# conversion program is able to correctly handle all the above mentioned cases.

VB Migration Partner manages an internal list of the most common Windows API methods, and generates the appropriate **MarshalAs** attribute. However, your VB6 project might be using an external method that takes a ByVal string with by-reference semantics and that isn't included in VB Migration Partner's internal list.

In such a case you can tell VB Migration Partner that a given string parameter uses by-reference semantics by using the special **VB6ByrefString** data type. Since you can't use this fictitious data type in VB6 code, the best approach is using a ParseReplace pragma to modify the VB6 Declare statement. For example, assume that you have the following code:

```
' the buffer parameter uses by-reference semantics

Declare Function GetStringValue Lib "mylibrary.dll" (ByVal buffer As String, _

                                                     ByVal size As Long) As Long
```

Here's how you can use the ParseReplace pragma to tell VB Migration Partner that the first parameter uses by-reference semantics, and the C# code that is generated:

```
' VB6
'## ParseReplace Declare Function GetStringValue Lib "mylibrary.dll" (ByVal buffer As _
                                          VB6ByrefString, ByVal size As Long) As Long
Declare Function GetStringValue Lib "mylibrary.dll" (ByVal buffer As String, _
                                          ByVal size As Long) As Long


// C#
[DllImport("mylibrary.dll")]
public static extern int GetStringValue([MarshalAs(UnmanagedType.VBByRefStr)]
                                          ref string buffer, int size);
```

C# doesn't support math operators and function in #define and #if directives

Unlike Visual Basic, C# doesn't support math operators and method calls inside #const, #if and #elif compiler directives. If necessary, VB Migration Partner generates custom **#define** or **#undef** statements containing derivate compilation constants, with remarks that show the original expression each constant corresponds to. If possible, these constants are given self-explanatory names, eg. VALUE_EQ_1. Here's an example:

```
' VB6
#Const CONST_ZERO = 0
#Const CONST_ONE = 1
#Const CONST_TWO = 2
```

```
' simple test for equality
#If CONST_ONE = 1 Then
   Debug.Print "One"
#ElseIf CONST_ONE = 2 Then
   Debug.Print "Two"
#End If

' compound tests
#If CONST_ONE > 1 Then
    Debug.Print "Zero"
#ElseIf CONST_ONE > 1 Or CONST_TWO >= 2 Then
    Debug.Print "Win32"
#End If

' math operators
#If Sgn(CONST_ONE) > 0 Then
    Debug.Print "One"
#ElseIf CONST_ONE <= CONST_TWO Then
    Debug.Print "Two"
#ElseIf Not CONST_ONE Then
    Debug.Print "None"
```

```
#End If

// C#
#undef CONST_ZERO                 // original value: 0
#define CONST_ONE                 // original value: 1
#define CONST_TWO                 // original value: 2

// IMPORTANT: if preceding #define/#undef statements are modified then ensure_
                                  that following statements are changed accordingly
#define CONST_ONE_EQ_1            // original value: CONST_ONE = 1
#undef CONST_ONE_EQ_2             // original value: CONST_ONE = 2
#undef CONST_ONE_GT_1             // original value: CONST_ONE > 1
#define DEFINE_EXPR_1             // original value: CONST_ONE_GT_1 or CONST_TWO >=
#define DEFINE_EXPR_2             // original value: Sgn(CONST_ONE) > 0
#define CONST_ONE_LE_CONST_TWO    // original value: CONST_ONE <= CONST_TWO

// simple test for equality
#if CONST_ONE_EQ_1
   VB6Project.DebugPrintLine("One");
#elif CONST_ONE_EQ_2
   // UPGRADE_WARNING (#0244): Code in #If ,#ElseIf, or #Else hasn't been upgraded.
   Debug.Print "Two"
```

```
#endif


// compound tests

#if CONST_ONE_GT_1

   // UPGRADE_WARNING (#0244): Code in #If ,#ElseIf, or #Else hasn't been upgraded.

   Debug.Print "Zero"

#elif DEFINE_EXPR_1

   VB6Project.DebugPrintLine("Win32");

#endif


// math operations

#if DEFINE_EXPR_2

   VB6Project.DebugPrintLine("One");

#elif CONST_ONE_LE_CONST_TWO

   // UPGRADE_WARNING (#0244): Code in #If ,#ElseIf, or #Else hasn't been upgraded.

   Debug.Print "Two"

#elif !CONST_ONE

   // UPGRADE_WARNING (#0244): Code in #If ,#ElseIf, or #Else hasn't been upgraded.

#endif
```

To ensure that the converted C# code is functional equivalent to the original VB6 code, VB Migration Partner generates several **#define** and **#undef** directives, depending on whether the corresponding value is nonzero or zero. Additionally, it generates a remark to the right of each "derivate" **#define** or **#undef** directive, reporting the original value or expression the compilation constant comes from.

It is crucial that you don't delete these remarks and that you manually change **#define** into **#undef**, or vice versa, if the value of any of the "primitive" compilation constants changes in the future.

As of this writing, other VB6-to-C# conversion tools are unable to handle all the subtleties of VB6 #If compiler directive and tend to generate incorrect code that generates either compilation errors or, worse, that include/exclude the wrong set of C# statements.

C# doesn't support initializing a class-level field by using another class-level field

In VB.NET it is legal to initialize a class-level field with an expression that contains a reference to another class-level field. When migrating from VB6 to VB.NET, such code can be the result of a refactoring technique that moves initialization from inside a Class_Initialize method, as in this example:

```
' VB6

Public FirstName As String

Public LastName As String



Private Sub Class_Initialize()

   FirstName = "Joe"

   LastName = FirstName & " Doe"

       ' ...

End Sub



' VB.NET
```

```
Public FirstName As String = "Joe"

Public LastName As String = FirstName & " Doe"



Private Sub Class_Initialize_VB6()

      ' ...

End Sub
```

When converting the previous code to C#, however, you get a compilation error, because the LastName field is assigned an expression that contains a reference to the FirstName field:

```
// C#

public string FirstName = "Joe";

public string LastName = FirstName + " Doe";



private void Class_Initialize_VB6()

{

     // ...

}
```

You can get rid of these compilation errors by means of the SetOption **DisableInitializeOptimization**, result you get:

```
// C#

public string FirstName;

public string LastName;
```

```
private void Class_Initialize_VB6()

{

      FirstName = "Joe";

      LastName = FirstName + " Doe";

      // ...

}
```

You can disable the refactoring technique at the project- or the file-level, so in practice you can and should use it only inside the files where a compilation error appears.

C# gotos can only jump to labels that are defined in the same syntax block

In C#, the statements included in a pair of { and } curly brackets define a scope block, a detail that may make the conversion from VB6 to C# very complex in some cases.

In fact, if a **goto** keyword in one block targets a label defined in a different block, a compilation error occurs. This problem becomes apparent when VB Migration Partner converts a Goto keyword that is located inside an If, Do, For, or Select Case block. The compilation error message that you get in such cases is:

No such label 'xxx' within the scope of the goto statement

In a few cases you can get rid of this error by applying the right pragma. For example, if the error is caused by a Gosub keyword that is being translated into a "push+goto" pair, you can try using the ConvertGosubs pragma to convert the Gosub into a method call instead.

In most cases, however, there is very little you can do, short of manually fixing the generated C# code. Better yet, you should consider modifying the original VB6 code to get rid of the Goto statement.

C# doesn't support default form instances

Unlike VB6 and VB.NET, the C# language doesn't define a "default instance" for the forms defined in the current project. To work around this limitation, VB Migration Partner generates a static property named **DefInstance** in each form class in the project, and adjusts form references as needed:

```
' VB6

Form1.BackColor = Form1.ForeColor



' C#

Form1.DefInstance.BackColor = Form1.DefInstance.ForeColor;
```

In most cases, the generated C# code is functionally equivalent to the original VB6 code, with one exception. In VB6 and VB.NET, you can close/unload and later reuse the default instance, and the "new" reference that you get is always the same form instance; most important, all fields defined at the form level retain their previous value. Conversely, when you close/unload or set a C# form's default instance to null, the form instance is actually destroyed. If you later reference again the default instance, another form is re-created, and all the fields in the new instance will be initialized to their default value (zero, empty string, or null).

In some rare cases, this behavior can introduce minor bugs in your converted project. Unfortunately, it is impossible to reproduce the VB6/VB.NET behavior in C#, therefore you might need to manually fix either the original VB6 code or the generated C# code.

C# doesn't support the VB.NET application framework

The migration from VB6 to VB.NET is simplified by the fact that the latter language supports an application framework that lets you decide whether the project has a startup form or a splash screen, and whether the application closes when the main form closes or when all forms close.

C# doesn't support such an application framework: all C# programs must have a static **void Main** method where the execution begins. Such a method typically contains an **Application.Run** statement that displays a given form modally; when this form closes, the entire application terminates.

If the VB6 project being converted doesn't have a Sub Main method, VB Migration Partner generates a class named **Program** and a method Main inside that class, which in turn contains an Application.Run statement that shows the default instance of the startup form.

```
static class Program

{

   [STAThread()]

   static void Main()

   {

       Application.EnableVisualStyles();

       Application.Run(Form1.DefInstance);

   }

}
```

This code works fine in most circumstances; however, it doesn't run as intended if the VB6 application has a splash screen which in turn loads the "real" main form and then unloads itself. In such a case, when converting from VB6 to VB.NET it is possible to use the EnableAppFramework pragma to select the splash screen; however this pragma is ignored when converting to C# because this language doesn't come with application framework.

In simpler scenarios you can work around this issue by using the **VB6Helpers.InitializeFormChaning** method and manually changing the Application.Run statement into a statement that loads the splash screen, as in this example:

```
static class Program

{

   [STAThread()]
```

```
    static void Main()
    {
        if ( VB6Helpers.InitializeFormChaining() )
            return;
        frmSplash.DefInstance = VB6Helpers.LoadForm(frmSplash.DefInstance);
        frmSplash.DefInstance.Show();
    }
}
```

In more complex, scenarios it might be necessary to either modify the original VB6 project or the generated C# code to get rid of the splash screen form.

C# can't work with the special VB6Variant data type

An important limitation is that the special **VB6Variant** data type isn't supported when converting to C#. The problem is, the VB6Variant data type relies on several VB.NET features that C# lacks.

For these reasons, you should avoid using the ChangeType or SetType pragmas to render a Variant variable as a VB6Variant variable. If you do so you don't receive any migration warning, yet the pragma will be ignored.

Likewise, VB Migration Partner doesn't support the VariantConversionSupport pragma, which is only useful with VB6Variant variables.

NOTE: starting with version 1.52, the VB6Variant class is not officially supported.

C# developers may prefer using native .NET methods rather than methods defined in the VB6Helpers class

VB Migration Partner maps all VB6 library methods – e.g. Left or DoEvents – to methods defined in the **VB6Helpers** class, which is part of the support library used by all migrated projects. This approach is necessary because only these helper methods ensure that the generated C# code works exactly like the original VB6 code:

```
' VB6

Sub Test(ByVal x As Integer, ByVal s As String)

' pad a string with blanks

    s = Left(s + Space(x), x)

End Sub



public void Test(short x, string s)

{

     // pad a string with blanks

     s = VB6Helpers.Left(s + VB6Helpers.Space(x), x);

}
```

You can reduce the calls to helper methods by means of the UseNetMethods pragma, which can be applied at the project-, file-, or method-level scope. If the previous VB6 code is under the effect of the UseNetMethods pragma, the generated C# code is different:

```
public static void Test(short x, string s)

{

   // pad a string with blanks

   s = (s + new string(' ' x)).Substring(0, x);

}
```

Please notice that, in general, replacing a VB6Helpers method with its native .NET equivalent seldom delivers C# code that is 100% functionally equivalent to the original VB6 code. For example, the **VB6Helpers.Left(s, n)** method works fine even if n is greater than the length of the s string (as it happens with the VB6 Left function), whereas the **s.SubString(0, n)** method throws an ArgumentOutOfRange exception in this case.

IMPORTANT NOTE #1: if you use the **UseNetMethods** pragma, it is your responsibility to ensure that no bugs are introduced in the converted C# code.

IMPORTANT NOTE #2: Not all VB6Helpers methods can be mapped to a native .NET method

The UseNetMethods pragma can take two optional arguments, which affect the number of cases when VB Migration Partner is allowed to use them. The first parameter is a regular expression that contains the list of VB6Helpers methods that VB Migration Partner is allowed to replace with .NET methods. For example, the following pragma tells VB Migration Partner to replace only the Left and Space functions inside the current Test method:

```
Sub Test(ByVal x As Integer, ByVal s As String)

   '## UseNetMethods "(Left|Space)"

   ' ...

End Sub
```

Whereas the following pragma prevents VB Migration Partner from converting the Split and Join functions found in the entire converted project:

```
      ' next regex matches anything except the words "Split" and "Join"

      '## project:UseNetMethods "(?!(Split|Join)\b).+"
```

The second optional argument is a Boolean that specifies how "aggressive" the pragma will be and if it is OK to repeat subexpressions in the generated code. An example will make this concept clearer. Consider the following VB6 code:

```
' VB6

Sub Test(ByVal x As Integer, ByVal s As String)
```

```
    '## UseNetMethods
    ' a simple call to Right function
    s = Right(s, x)
    ' a call to Right function that takes a string expression
    s = Right(Space(x) + s, x)
End Sub


// C#
public static void Test(short x, string s)
{
    // a simple call to Right function
    s = s.Substring(s.Length - x);
    // a call to Right function that takes a string expression
    s = VB6Helpers.Right(new string(' ', x) + s, x);
}
```

As you may notice, only the first occurrence of **VB6Helpers.Right** was converted into the SubString .NET method; the second occurrence was not converted because it would have produced unnaturally bloated and inefficient code that contains a repetition of the **new string(' ', x) + s** subexpression. If you really wish to get rid of the VB6Helpers.Right call, however, you can force VB Migration Partner to do so, by specifying True in the second argument of the UseNetMethods pragma:

```
' VB6
Sub Test(ByVal x As Integer, ByVal s As String)
    '## UseNetMethods , True
    ' a call to Right function that takes a string expression
```

```
      s = Right(Space(x) + s, x)
End Sub


// C#
public static void Test(short x, string s)
{
    // a call to Right function that takes a string expression
    s = (new string(' ', x) + s).Substring((new string(' ', x) + s).Length - x);
}
```

It's evident that the generated code is less efficient than it is when using the VB6Helpers.Right method. However, you might wish to generate it anyway, if you later plan to optimize it by hand, when the migration is complete. In this specific case, the statement would be simplified as follows:

```
     s = (new string(' ', x) + s).Substring(s.Length);
```

C# requires special helper methods for files longer than 2^31 bytes

When converting to VB.NET, VB Migration Partner converts the Seek#, Loc# and LOF# VB6 functions into FileSeek6, Loc6, and LOF6 functions, respectively. These functions return a 64-bit integer, which means that they are ready to handle files of any size.

When designing a similar set of file methods to be exposed by the VB6Helpers class for C# applications, we realized that having to deal with file functions that return a **long** value would have forced VB Migration Partner to generate a lot of casts in generated code (because in most cases the returned value is assigned to a 32-bit integer variable):

```
int filesize = (int) VB6Helpers.LOF(filename)
```

For this reason, the **FileSeek**, **Loc** and **LOF** functions defined in VB6Helpers class return a 32-bit value, which means that they work fine with files up to 2^31 bytes (about 2GB). However, the class also exposes three additional methods named **FileSeek64**, **Loc64**, and **LOF64**, which return a 64-bit value.

```
'## project:PostProcess "VB6Helpers\.(?FileSeek|Loc|LOF)", "VB6Helpers.${fn}64"
```

However, applying the above pragma is surely going to cause several casting compilation errors in the generated code, and you have to fix such compilation errors by hand.

Other minor differences between Visual Basic and C#

This section outlines a few other minor differences between Visual Basic and C#. In general, VB Migration Partner automatically accounts for all these differences, which we describe here only for reference purposes:

- **C# is case-sensitive:** VB Migration Partner translates all VB symbols so that they use the same casing, even if they appear with different cases in the VB6 code. An exception to this rule occurs when you apply the UseDynamic pragma to have VB Migration Partner generate late-bound method calls (in which case the casing found in the original VB6 code is used).
- **C# has a different set of keywords and other reserved names (e.g. "Properties"):** VB Migration Partner ensures that VB6 variables and methods whose name would be invalid in C# are correctly renamed to avoid compilation errors.
- **C# doesn't support project-level namespace imports:** All the namespaces that would be imported at the project-level are rendered as a **using** statement inside each and every source file. VB Migration Partner supports the AddImports pragma to add either project-level or file-level using directive, but it is recommended that you only use this pragma only to add file-level using statements, to avoid unnecessary clutter in other files.
- **C# doesn't support Octal constants (eg &O1234567):** VB Migration Partner converts these constants into hex values.
- C# requires that non-integer numeric literals have a trailing "F" or "M" suffix when assigned to a float or decimal variable: this is a special case of strict typing and VB Migration Partner automatically accounts for it.

- **C# uses the "\" (backslash) character in string literals as an escape character:** If a string literal contains one or more backslashes, VB Migration Partner converts it into a "@" (verbatim) string.
- **C# can't implement the Mid method to assign a substring:** this happens because .NET doesn't support properties that take a by-reference parameter, and in fact the VB.NET compiler generates some hidden code to work around this limitation. When converting to C#, VB Migration Partner converts assignments that use the Mid method into calls to the VB6Helpers.MidSet method.
- **C# requires that the parameter for property setters be named "value":** VB Migration Partner correctly changes the name of this parameter if necessary.
- **C# doesn't support the Collection object:** VB Migration Partner converts all Collection references to the Collection6 object defined in the support library.
- **C# issues a warning if a class or method lacks an XML comment:** VB Migration Partner avoids a large number of compilation warnings of this type by disabling it in the Properties-Build page.

Pragmas that aren't supported when converting to C#

The following migration pragmas aren't supported or have a different behavior when converting to C#:

UseTryCatch: this pragma is always assumed to be present, and VB Migration Partner always attempts to convert On Error statements to try-catch blocks.

DeclareImplicitVariables: this pragma is always assumed to be present, because C# requires that all variables be declared.

LogicalOps: this pragma is ignored, because VB Migration Partner automatically detects when an And/Or operators should be translated into a logical && or || operator.

FixRemarks: this pragma is ignored because it is useless when converting to C#.

NullSupport: this pragma is ignored and most methods defined in the VB6Helpers class can accept and return null values in all cases - even though in most cases this ability won't prevent exceptions from appearing at runtime, if your code isn't prepared to handle the DBNull value returned by these methods.

BinaryCompatibility: this pragma is ignored, as current version of VB Migration Partner is unable to generate binary-compatible DLLs when converting to C#.

EnableAppFramework: this pragma is ignored, because C# doesn't support the kind of application framework that VB.NET supports.

VariantConversionSupport: this pragma is ignored because it is only useful with VB6Variant variables, which aren't supported in this version.

LateBoundProperties, LateBoundMethods, and LateBoundVariantMembers: all these pragmas are ignored when converting to C#.