Visual Basic Upgrade Companion vs. Code Advisor

When planning a migration project with the Visual Basic Upgrade Companion (VBUC), many users ask if they should still execute the Visual Basic 6.0 Code Advisor, since VBUC automatically takes care of many of the tasks flagged by Code Advisor. Which of these tasks can be delegated to VBUC, and which ones have to be addressed manually? This article provides a practical view focused on minimizing manual work and taking advantage of the enhanced features of VBUC.

By Juan Fernando Peña

One of the tasks usually recommended before beginning a Visual Basic 6.0 to .Visual Basic .NET migration project is the analysis of the code to be migrated with Code Advisor. Code Advisor, provided at no cost by Microsoft, is a useful tool that evaluates and recommends modifications to the VB6 source code to prepare for and simplify the upgrade process and reduce the post-migration manual effort that would otherwise be required to complete the upgrade. This advice is presented as a series of comments added to the source code, all of them beginning with the “FIXIT” prefix. For this reason, we will refer to Code Advisor recommendations as “FixIts” in this article. Code Advisor, for example, would flag the following line of Visual Basic 6.0 code

```
Dim z
```

resulting in the following output

```
'FIXIT: Declare 'z' with an early-bound data type…
Dim z
```

because--as we will see in more detail later in this article--variables that are not declared with a specific type result in migration issues. Code Advisor can be downloaded from http://www.microsoft.com/downloads/

The Visual Basic Upgrade Companion (VBUC), ArtinSoft’s improved version of the Visual Basic Upgrade Wizard migration tool, is enhanced to produce a higher level of automatic migration. Some of the issues commonly identified by Code Advisor are already addressed by VBUC, resulting in a lower percentage of the manual effort necessary to complete any given migration process. Therefore, it is important for developers to be aware of and differentiate between the instances in which the source code needs to be modified in Visual Basic 6.0 according to Code Advisor and instances where such work is already covered by VBUC.

The difference depends greatly on the characteristics of the specific source code to be migrated. As a rule of thumb, you should always execute Code Advisor--doing so will help you identify potential migration issues that will prepare you to successfully tackle the upgrade process. The following sections list the FixIts that are addressed automatically by VBUC and that require no manual work in Visual Basic 6.0.

<'object'> not upgraded to Visual Basic .NET by the Upgrade Wizard

Several objects that were used in VB6, such as the Printer Object and the Printers Collection, are not directly available in .NET, and are therefore not converted by the Upgrade Wizard. However, the

following objects are supported by VBUC by using helper classes implemented in .NET or by using special classes from the .NET Framework:

- Printer Object
- Printers Collection
- Clipboard Object
- Forms Collection

Portions of code that use any other unsupported object will have to be rewritten in VB.NET.

Declare <'variable'> with an early-bound data type

Variables that were either not declared with a specific type or declared As Variant in VB6 are declared As Object in VB.NET. Since the actual type of a Variant (or late-bound) variable is not known until runtime, the Upgrade Wizard will not be able to determine which operations to apply to that variable, resulting in migration errors. For instance, since the type of the variable is not known at migration time, default properties will not be expanded, leading to issues and crashes when the migrated application is executed.

The Visual Basic Upgrade Companion features an Artificial Intelligence-based type inference engine, which in most situations will be able to infer the type of a variable based on its usage in the source code, given that its runtime type remains consistent and that it interacts with literals and variables with known types. For example, the following code snippet

```
Private Sub Sum()
a = 1
b = 2 + 3
End Sub
```

would be converted by VBUC to

```
Private Sub Sum()
Dim a As Byte = 1
Dim b As Byte = 2 + 3
End Sub
```

Even when VBUC may not be able to infer the type of all the late-bound variables in the source code, it's a good idea to let VBUC do as much work as it can do for you. Variables that still do not have a specific type after migration has taken place can be identified if you search for the following late binding Upgrade Warning

```
'UPGRADE_WARNING: Couldn't resolve default property of object <object name>.
```

Replace <'variant'> function with <'string'> function

In Visual Basic 6.0, text functions such as Mid, Trim and Right have both a "String" version and a "Variant" version, where the first one returns a String type and the latter returns a Variant type. String versions of these functions are preferable since they avoid late binding. These strongly-typed functions can be recognized because their names end with a dollar sign ("$"), which is also used in

VB6 to denote string types. For instance, Mid has a Variant return type and Mid$ has a String return type.

Many VB6 programmers were not aware of this difference and tended to use the Variant version of the function instead of the early-bound, String version. From a migration standpoint, using the Variant function increases the amount of late-bound variables, generates more Upgrade Warnings, and decreases the quality of the converted code. In the majority of cases, the Visual Basic Upgrade Companion will successfully infer the type of the variables and will automatically add a call to the appropriate function. For instance, the following Visual Basic 6.0 code fragment

```
'FIXIT: Declare 's' with an early-bound data type…
Dim s
s = "Hello World"
'FIXIT: Replace 'Mid' function with 'Mid$' function…
s1 = Mid(s, 1, 4)
'FIXIT: Replace 'Right' function with 'Right$' function…
s3 = Right(s, 4)
'FIXIT: Replace 'Trim' function with 'Trim$' function…
s4 = Trim(s)
'FIXIT: Replace 'LTrim' function with 'LTrim$' function…
s5 = LTrim(s)
'FIXIT: Replace 'RTrim' function with 'RTrim$' function…
s6 = RTrim(s)
```

would be converted to the following Visual Basic .NET code

```
'FIXIT: Declare 's' with an early-bound data type…
Dim s As String = "Hello World"
'FIXIT: Replace 'Mid' function with 'Mid$' function…
Dim s1 As String = s.Substring(0, 4)
'FIXIT: Replace 'Right' function with 'Right$' function…
Dim s3 As String = s.Substring(s.Length - (Math.Min(s.Length, 4)))
'FIXIT: Replace 'Trim' function with 'Trim$' function…
Dim s4 As String = s.Trim()
'FIXIT: Replace 'LTrim' function with 'LTrim$' function…
Dim s5 As String = s.TrimStart()
'FIXIT: Replace 'RTrim' function with 'RTrim$' function…
Dim s6 As String = s.TrimEnd()
```

In this case, VBUC is able to infer the type of the variables, making the replacement of Variant functions with String functions unnecessary. Also, note how the conversion tool uses equivalent methods from the String class instead of backwards compatibility functions to perform the same operations.

The use of <'enumeration'> is not valid for the property being assigned

This FixIt usually appears when invalid enumeration values are being set for a property. In Visual Basic 6.0, you can assign invalid enumeration values to properties and still obtain the expected results if the underlying integer values are the same. For instance, “vbNormal” has been commonly used to set the MousePointer property of a window to its default, when the correct value is

“vbDefault”. Since both of them share a common underlying value of zero, the same results are observed at runtime even when the assignment is incorrect.

In Visual Basic .NET, only the values defined for each property are allowed. However, the Visual Basic Upgrade Companion has extended coverage for enumerations, which allow it to recognize incorrect enumeration values and constants and replace them with the appropriate ones. For instance, the following VB6 code uses an incorrect enumeration value and the literal “2” to set the MousePointer property of a form and the Alignment property of a TextBox, respectively.

```
Me.MousePointer = vbDefault

'FIXIT: The use of 'vbNormal' is not valid for the property…

Me.MousePointer = vbNormal
Me.Text1.AlignMent = vbLeftJustify
Me.Text1.AlignMent = vbCenter
Me.Text1.AlignMent = vbRightJustify
Me.Text1.AlignMent = 2
```

The Visual Basic Upgrade Companion is able to recognize the values being assigned and translate them to their correct equivalents in .NET:

```
Me.Cursor = Cursors.Default

'FIXIT: The use of 'vbNormal' is not valid for the property…

Me.Cursor = Cursors.Default
Me.Text1.TextAlign = HorizontalAlignMent.Left
Me.Text1.TextAlign = HorizontalAlignMent.Center
Me.Text1.TextAlign = HorizontalAlignMent.Right
Me.Text1.TextAlign = HorizontalAlignMent.Center
```

Use Option Explicit to avoid implicitly creating variables of type Variant

Whether it is a good practice to always use Option Explicit in Visual Basic 6.0, the type inference capabilities of VBUC can solve most of the late binding and make this revision unnecessary in VB6. Types that could not be inferred during migration will be marked with the following late binding Upgrade Warning

```
'UPGRADE_WARNING: Couldn't resolve default property of object <object name>
```

and should be reviewed after migration.

Case Study

To test the effectiveness of the Visual Basic Upgrade Companion, three real-world applications were processed by Code Advisor and then migrated to .NET by VBUC. Table 1 shows the size of each application, measured in Effective Lines of Code, and the number of FixIts that were added by Code Advisor to each one. In this context, Effective Lines of Code represent all the VB6 lines in classes,

forms or modules, and all the design lines in forms or user controls. Comment or blank lines are not included in the measure.

Application	Effective Lines of Code	Number of FixIts
Application 1 (Windows EXE)	20,051	91
Application 2 (Windows EXE)	16,485	308
Application 3 (Windows DLL)	17,395	372
Total	**53,931**	**771**

Table 1. Size of the migrated applications and number of “FixIts” added by Code Advisor

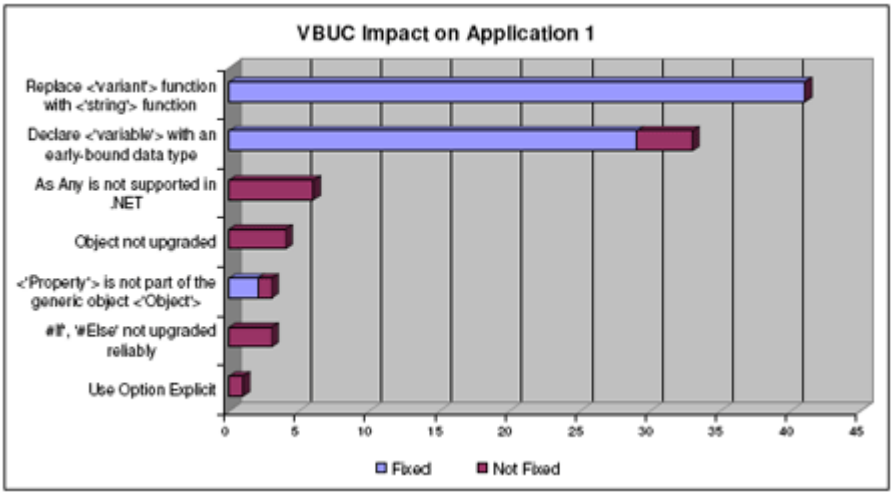
A total of 771 FixIts were added to the applications by Code Advisor. The following paragraphs explain how these FixIts are classified for each application and show the impact that migrating with VBUC had on each one.

Table 2 shows a list of the generated FixIts in the Visual Basic 6.0 source code for Application 1. Most of the FixIts in this application refer to late binding (“Replace <'variant'> function with <'string'> function” and “Declare <'variable'> with an early-bound data type”).

FixIts generated on Application 1		
FixIt Description	**Occurrences fixed by VBUC**	**Total Occurrences**
Replace <'variant'> function with <'string'> function	41	41
Declare <'variable'> with an early-bound data type	29	33
As Any is not supported in .NET	0	6
Object not upgraded	0	4
<'Property'> is not part of the generic object <'Object'>	2	3
#If', '#Else' not upgraded reliably	0	3
Use Option Explicit'	0	1
Total	**72**	**91**

Table 2. FixIts generated on Application 1

The impact of migrating Application 1 with VBUC was remarkable, as 72 out of 91 generated FixIts were corrected by the migration tool, the highest impact dealing with issues related to late binding. Figure 1 shows the impact of VBUC on each type of FixIt generated by Code Advisor on Application 1.



[Click to Enlarge]

Figure 1. Automatic coverage for each FixIt in Application 1

All the 41 occurrences of the “Replace <'variant'> function with <'string'> function” FixIt were corrected by VBUC. 29 out of 33 occurrences of “Declare <'variable'> with an early-bound data type” were corrected, which can be attributed to the type inference engine. Also, thanks to the same type inference mechanism, 2 out of 3 occurrences of “<'Property'> is not part of the generic object <'Object'>” were fixed. Figure 2 shows a representation of the impact that VBUC had on the FixIts found in Application 1.

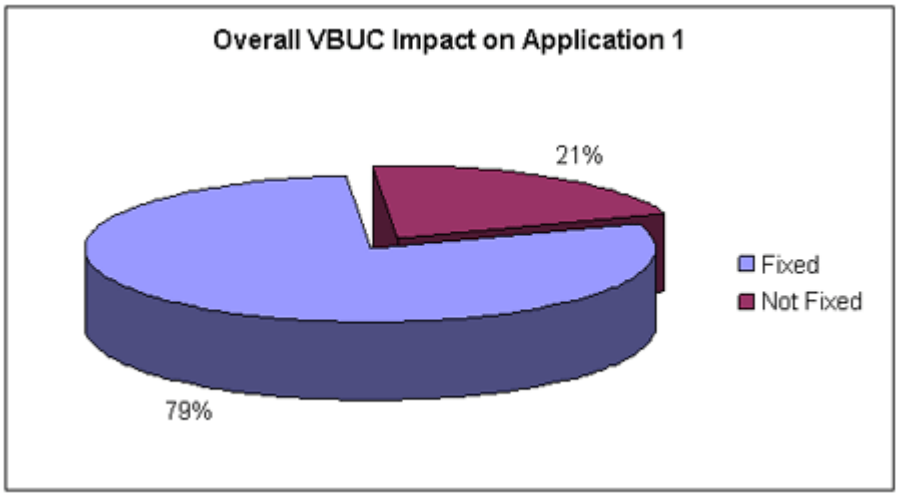


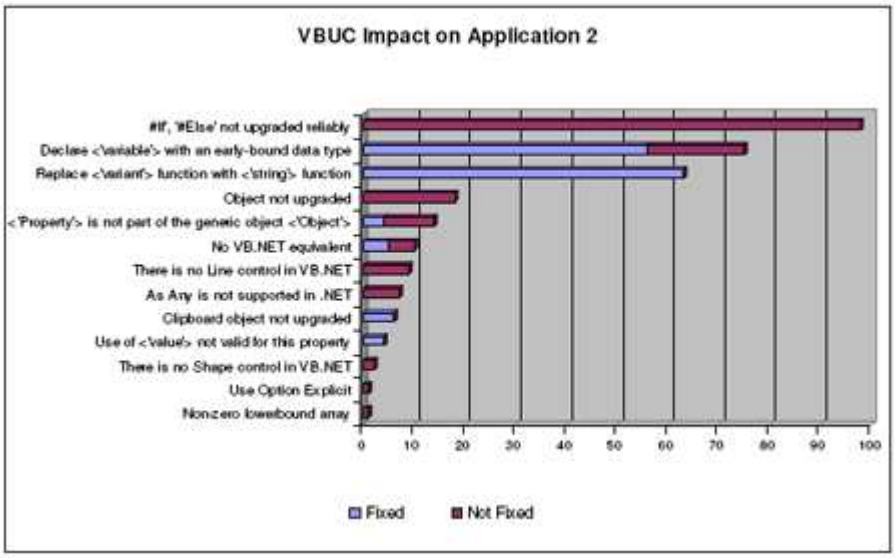
Figure 2. Overall VBUC Impact on Application 1

VBUC encountered a different scenario with Application 2,, where the predominant FixIt generated by Code Advisor was “#If', '#Else' not upgraded reliably”. The Visual Basic Upgrade Companion cannot deal with conditional compile directives, and only the portion of the ‘#If’ block that evaluated to True was upgraded, resulting in 98 FixIts that VBUC was unable to correct. Table 3 shows how the FixIts are distributed in Application 2.

FixIts generated on Application 2		
FixIt Description	**Occurrences Fixed by VBUC**	**Total Occurrences**
#If, '#Else' not upgraded reliably	0	98
Declare <'variable'> with an early-bound data type	56	75
Replace <'variant'> function with <'string'> function	63	63
Object not upgraded	0	18
<'Property'> is not part of the generic object <'Object'>	4	14
No VB.NET equivalent	5	10
There is no Line control in VB.NET	0	9
As Any is not supported in .NET	0	7
Clipboard object not upgraded	6	6
Use of <'value'> not valid for this property	4	4
There is no Shape control in VB.NET	0	2
Use Option Explicit	0	1
Non-zero lower bound array	0	1
Total	**138**	**308**

Table 3. FixIts generated on Application 2

Figure 3 shows the impact that VBUC had on each FixIt category in Application 2. As can be observed, the predominant FixIt was not impacted, but a substantial contribution was made to fix "Declare <'variable'> with an early-bound data type" and "Replace <'variant'> function with <'string'> function".



[Click to Enlarge]

Figure 3. Automatic coverage for each FixIt on Application 2

Figure 4 shows the overall contribution made by VBUC to address the FixIts found in Application 2.

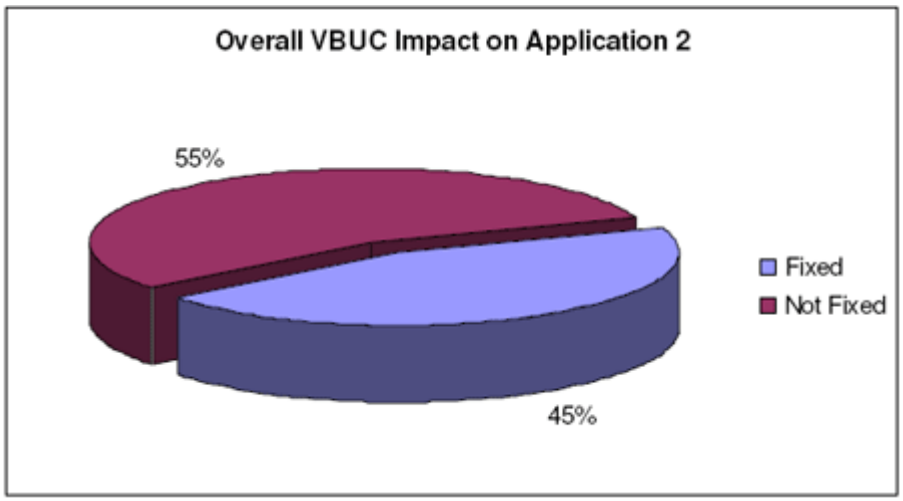


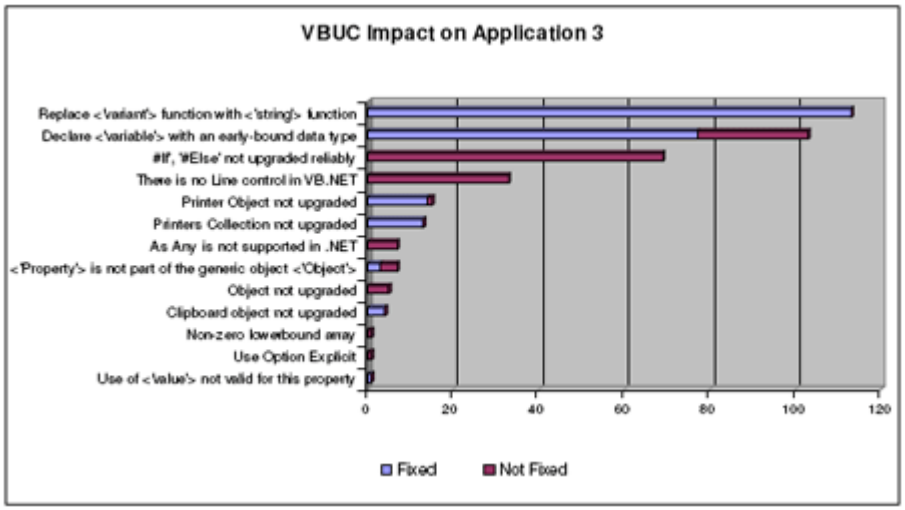
Figure 4. Overall VBUC Impact on Application 2

VBUC had a more significant impact on Application 3 than on Application 2, as 225 out of 372 FixIts were handled automatically. Even though there was a significant number of FixIts that weren't covered ("#If', '#Else' not upgraded reliably" and "There is no Line control in VB.NET"), the most important FixIts were successfully addressed. Table 4 shows the distribution of the FixIts in Application 3.

FixIts generated on Application 3		
FixIt Description	**Occurrences Fixed by VBUC**	**Total Occurrences**
Replace <'variant'> function with <'string'> function	113	113
Declare <'variable'> with an early-bound data type	77	103
#If', '#Else' not upgraded reliably	0	69
There is no Line control in VB.NET	0	33
Printer Object not upgraded	14	15
Printers Collection not upgraded	13	13
As Any is not supported in .NET	0	7
<'Property'> is not part of the generic object <'Object'>	3	7
Object not upgraded	0	5
Clipboard object not upgraded	4	4
Non-zero lowerbound array	0	1
Use Option Explicit	0	1
Use of <'value'> not valid for this property	1	1
Total	**225**	**372**

Table 4. FixIts generated on Application 3

Figure 5 shows the impact that migrating with VBUC produced on each category of FixIts. As well as fixing most of the late-binding related issues, VBUC had an important impact on several cases of "<'object'> not upgraded to Visual Basic .NET by the Upgrade Wizard", such as "Printer Object not upgraded", "Printers Collection not upgraded" and "Clipboard object not upgraded".



[Click to Enlarge]

Figure 5. Automatic coverage for each FixIt in Application 3

Figure 6 shows the overall VBUC Impact on Application 3.

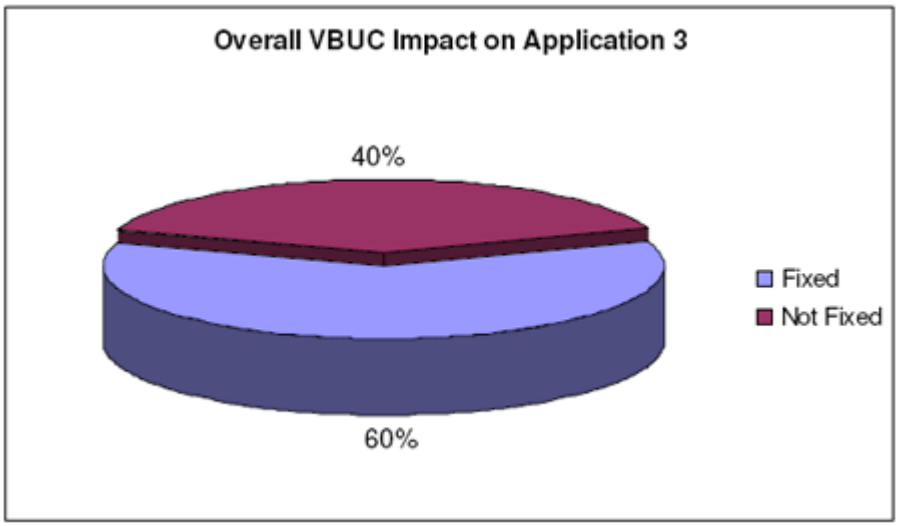


Figure 6. Overall VBUC Impact on Application 3

Conclusions

In this article, we have covered the list of Code Advisor FixIts that are automatically handled by the Visual Basic Upgrade Companion. Also, we evaluated the impact that VBUC had on the FixIts of three different real-world Visual Basic 6.0 applications. The percentage of FixIts of these applications that were automatically addressed by VBUC ranged from 45% to 79%--the biggest impact occurred with the “Replace <'variant'> function with <'string'> function” and “Declare <'variable'> with an early-bound data type” FixIts.

While the Visual Basic Upgrade Companion significantly reduces the work that would otherwise be carried out manually, it is always recommendable to execute Code Advisor and analyze its output to

help foresee potential migration issues. If the migration is to be done with VBUC, correcting the resulting FixIts should be focused only on those not covered by VBUC; while those FixIts that are automatically addressed can be reviewed in Visual Basic .NET, after the automatic migration is completed.

References

Microsoft Corporation (2006). Code Advisor 1.1 for Visual Basic 6 FixIt Reference.

Microsoft Corporation (2006). Upgrading Visual Basic 6.0 Applications to Visual Basic .NET and Visual Basic 2005. Microsoft Patterns and Practices.

http://msdn.microsoft.com/library...

About the Author

Juan Fernando Peña is a Microsoft Certified Professional (MCP) and works as a Senior Consultant in Visual Basic application migrations at ArtinSoft, the company that developed the Visual Basic Upgrade Wizard and the Visual Basic Upgrade Companion. He has been involved in many application conversion projects around the world and currently manages several migration initiatives at ArtinSoft involving enhancements to VBUC and the conversion of Visual Basic 6.0 code to .NET.